READ MODE · ANSWER FIRST

How LLM applications actually work

Build a practical mental model of context, instructions, sampling, reasoning, streaming, schemas, tools, retries and token cost.

CHEAT SHEET · 10 ESSENTIAL IDEAS

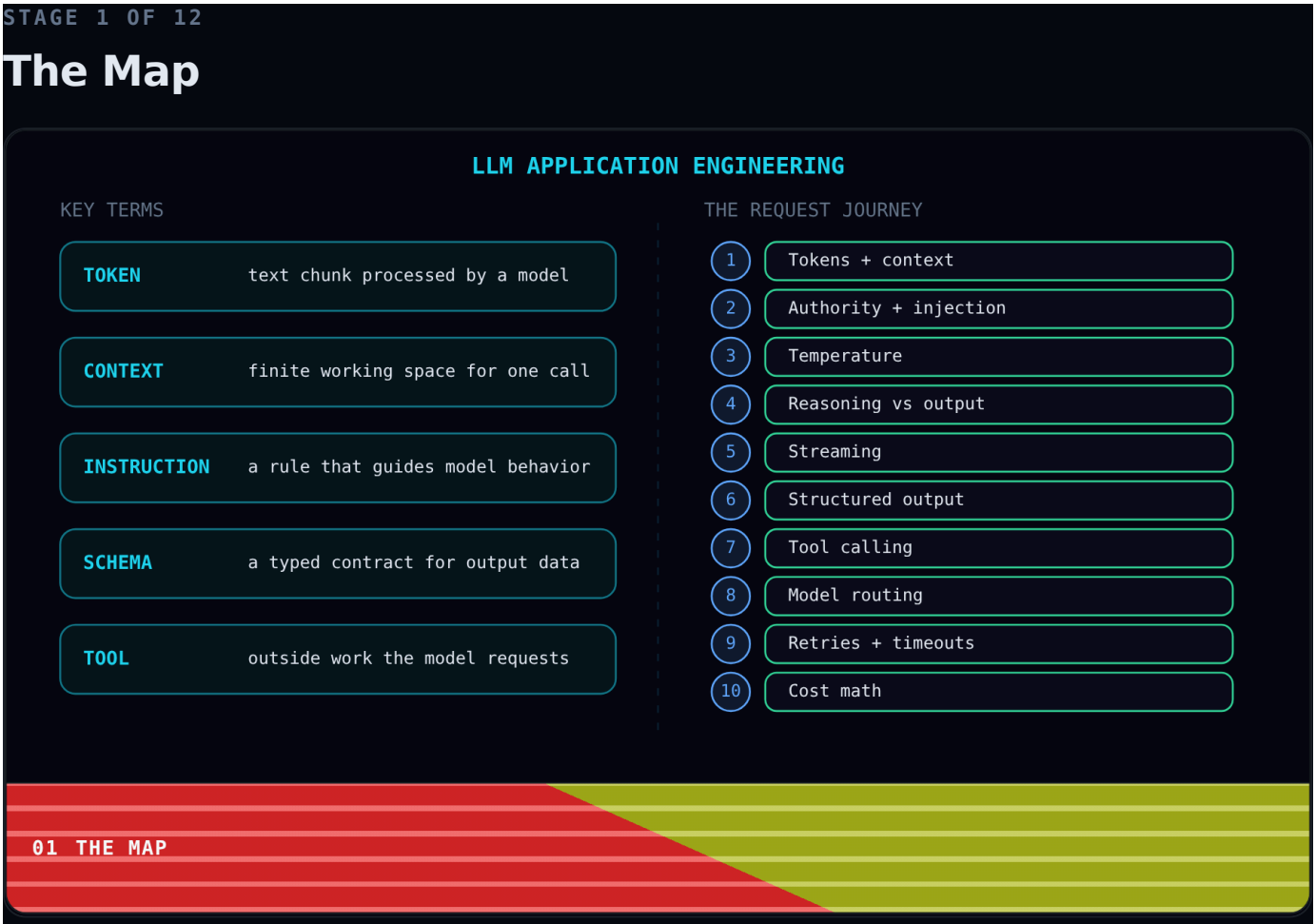
The whole story in 10 lines

A real LLM application is ordinary software around a probabilistic model, with explicit budgets, boundaries, contracts, tools...

1. A capacity tray makes prompt usage, output reserve and overflow visible from length alone.
2. The hero contrasts a trusted authority stack with an isolated untrusted document that attempts an injection.
3. Shared probability bars show the distribution sharpening or flattening without changing the candidate set.
4. A bounded hidden workbench feeds a small public result without pretending to reveal chain of thought.
5. Ordered chunks fill the same client buffer either progressively or all at once.
6. Two lanes use the same payload encoding so valid JSON and strict schema adherence can be compared directly.
7. A return path makes the application execution boundary and second model request unavoidable.
8. A routing matrix replaces the false idea that one model is best for every request.
9. A linear timeline shows exponential gaps growing toward an immovable deadline.
10. Two contribution meters make the input and output parts of the bill visible before traffic multiplies them.

STAGE 1 OF 12

The Map



You might think an LLM app is just a clever prompt. It is really software that prepares a request, calls a model, and turns an uncertain answer into a dependable product.

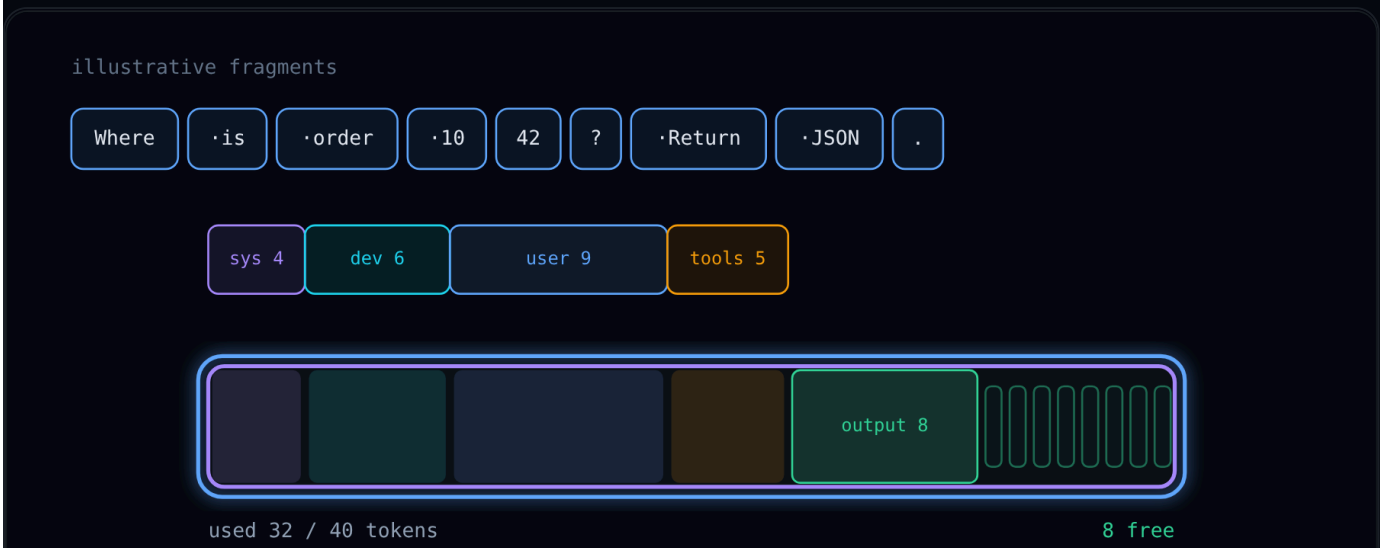
Let us start with two basic terms. A token is a chunk of text, and context is the limited workspace that holds everything the model can use for one request.

Instructions tell the model what to do. A schema defines the answer shape your code expects, and a tool lets the model ask your app to do outside work.

A real product also needs model routing, deadlines, retries, security checks, evals, and cost tracking. In other words, this is a software engineering problem.

We will follow one order request through all ten jobs. First, let us look at the basic limit every request faces: all its tokens must fit in one context window.

Tokens + Context



02 TOKENS + CONTEXT

Now let us turn the order question into tokens. The model may split words into different pieces, so the token fragments shown here are only an example.

Here is the important part: system rules, developer rules, the user message, and tool definitions all share one context budget, which means their token counts add up.

We also need to save space for the model's answer. So what happens when the input and the reserved output space are larger than the whole window?

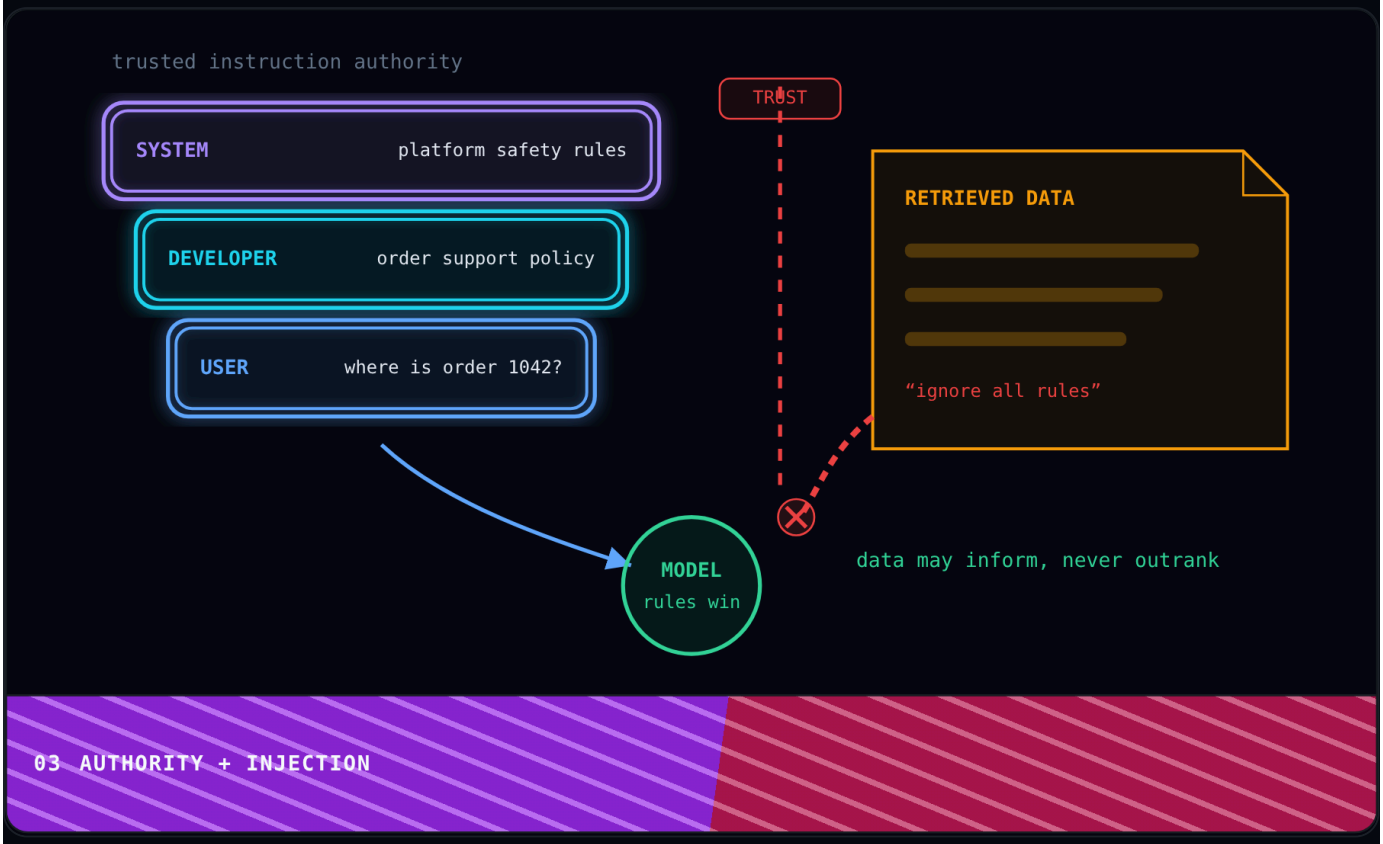
Choose 28 in Window and notice how the input plus output reserve crosses the limit, which means the app must reject, shorten or rebuild this request.

Now choose 40 in Window and the same request fits and still has eight token slots left after the output reserve, giving us a safe budget for the next stages.

So context is really capacity planning, not magic memory. Count every input and save room for output. Next, we will decide which text has authority.

STAGE 3 OF 12

Authority + Injection



03 AUTHORITY + INJECTION

Our request fits, but now we need a chain of command. System rules come first, developer instructions come next, and the user request comes after them.

Think of the developer message as the app's policy. The user provides the goal and order number, much like passing arguments into a function.

Retrieved pages are still data, even when they contain command-like sentences. Set Retrieved text to Normal and notice that this data stays in its own lane.

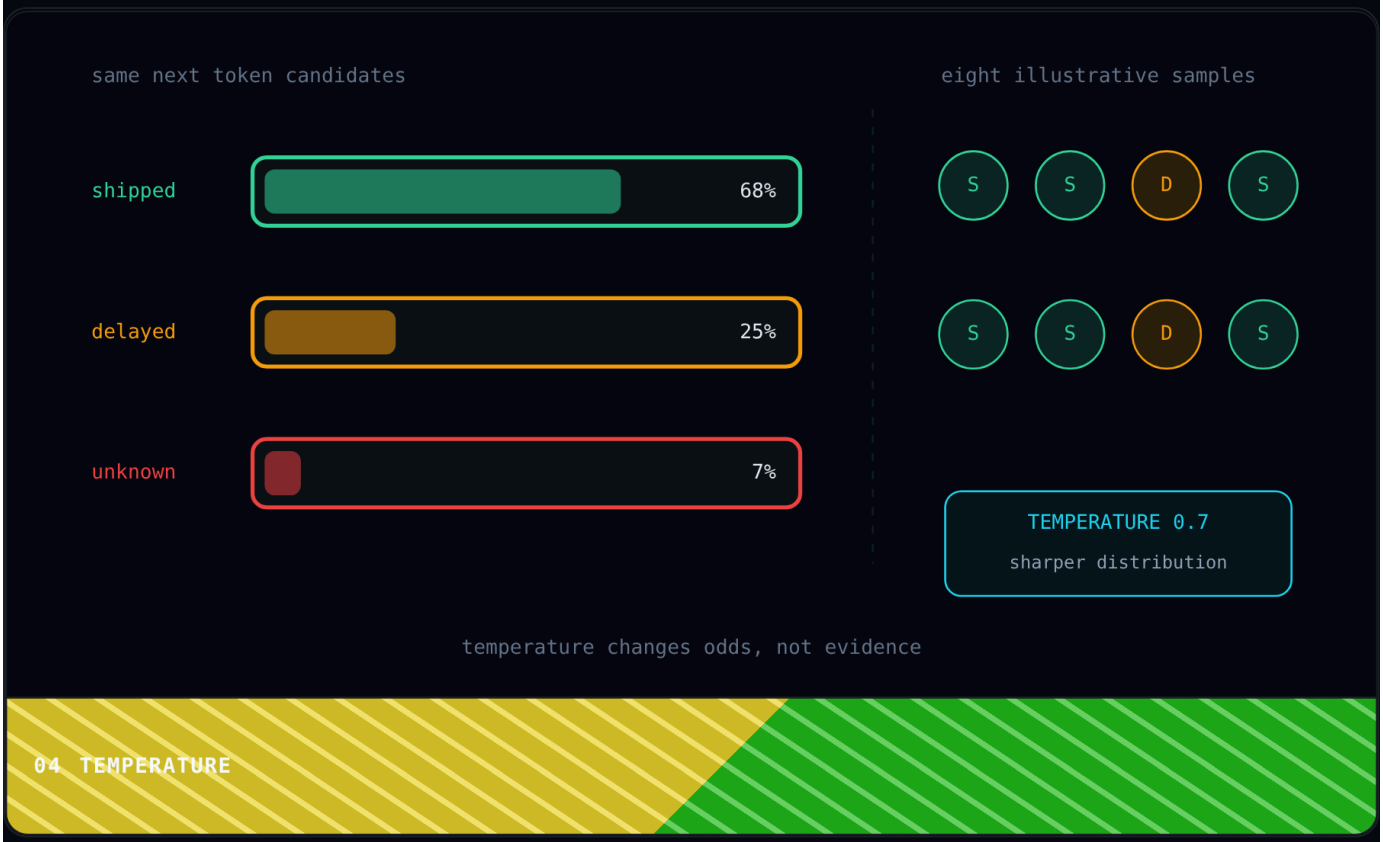
Now set Retrieved text to Injected and the page says “ignore all rules.”, so the model can read that sentence, but does that give the page more authority?

No, because the page stays below the trust boundary while system and developer rules remain in charge, so readable text does not automatically become policy.

Prompt injection is a security boundary problem, so treat outside text as untrusted, limit tools and check effects before we explore token sampling.

STAGE 4 OF 12

Temperature



04 TEMPERATURE

The trusted request is ready and the model does not write the whole answer at once. It scores several choices for the next token, picks one, and repeats.

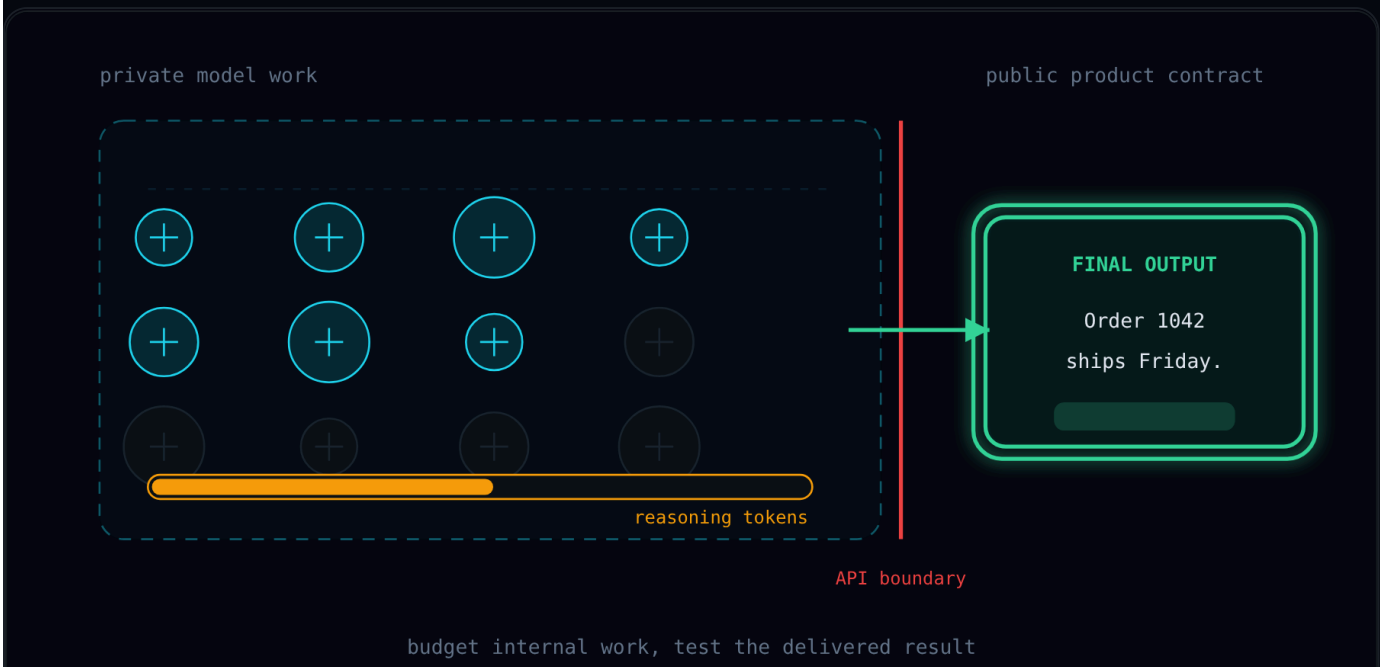
In this simple sampler, temperature zero picks the highest scored token. That reduces random variation, but it does not make the whole model service perfectly fixed.

What do you think happens when we raise Temperature but keep the same candidate scores?

Choose 1.2 in Temperature and watch the probability bars flatten, which gives lower ranked tokens a better chance of being chosen.

Temperature changes sampling, not truth and for repeatable behavior, pin a model snapshot and run evals too. Next, we will separate reasoning from the final answer.

Reasoning vs Output



05 REASONING VS OUTPUT

Sampling chooses tokens, but some models spend extra reasoning tokens on internal work first and that hidden work and the final answer are not the same product.

Those reasoning tokens can use context space and count toward billed output, even though the raw internal reasoning is not returned as normal text.

If part of the work stays hidden, what should your app treat as the stable contract?

The final answer is the contract, so choose High in Reasoning effort and notice that hidden work grows while the delivered answer keeps its typed shape.

Budget the reasoning, then judge the final result with evals. Do not build product logic around hidden thought text. Next, let us stream the visible answer.

STAGE 6 OF 12

Streaming

CLIENT BUFFER output_text.delta

Order 1042 ships

same final answer, different time to first useful text

06 STREAMING

We now have an answer to deliver and without streaming, the user waits while the service generates the complete response.

With streaming, typed delta events carry small text chunks in order and the client keeps adding each new chunk to the same output buffer.

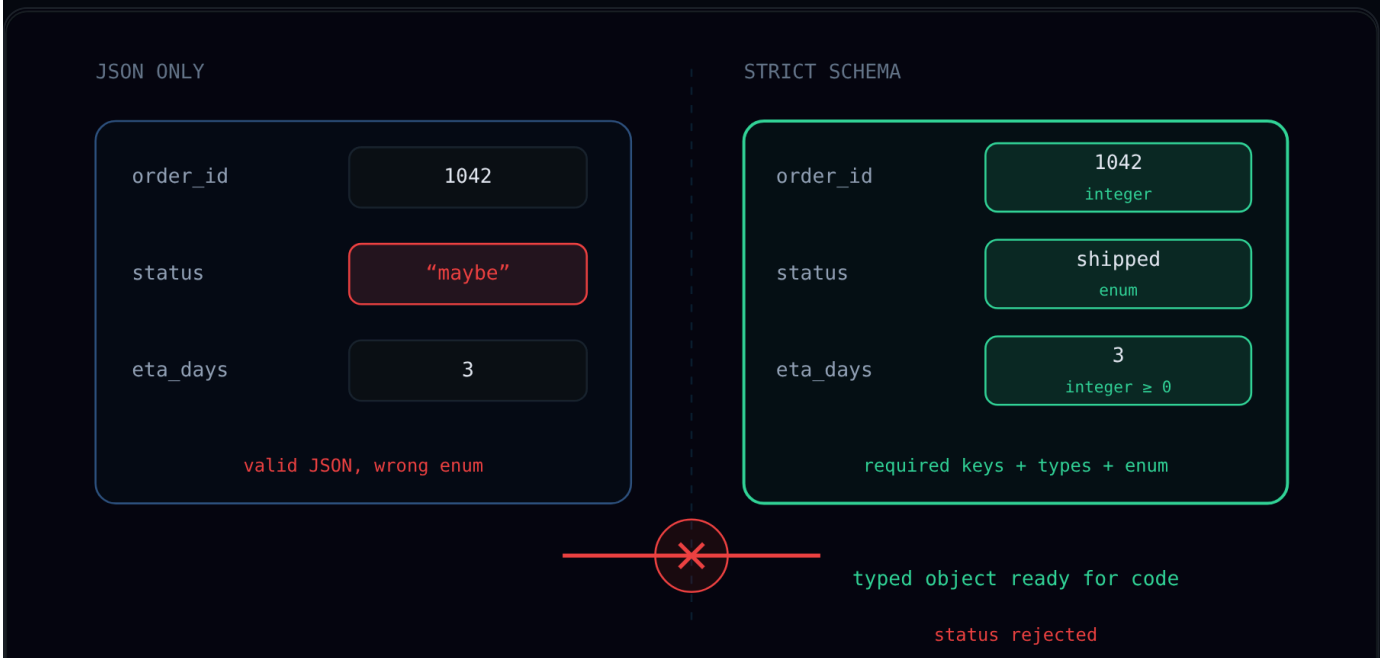
Here is the key question: does streaming make the model finish sooner?

No, because choosing Stream in Delivery reveals useful text earlier, but the final chunk still arrives at the same completion time.

Streaming improves time to first useful text, not total generation time. It also makes errors and moderation harder. Next, we will require a strict answer shape.

STAGE 7 OF 12

Structured Output



07 STRUCTURED OUTPUT

Streaming gave us text, but application code usually needs structured data and plain JSON can be valid while still having the wrong keys or values.

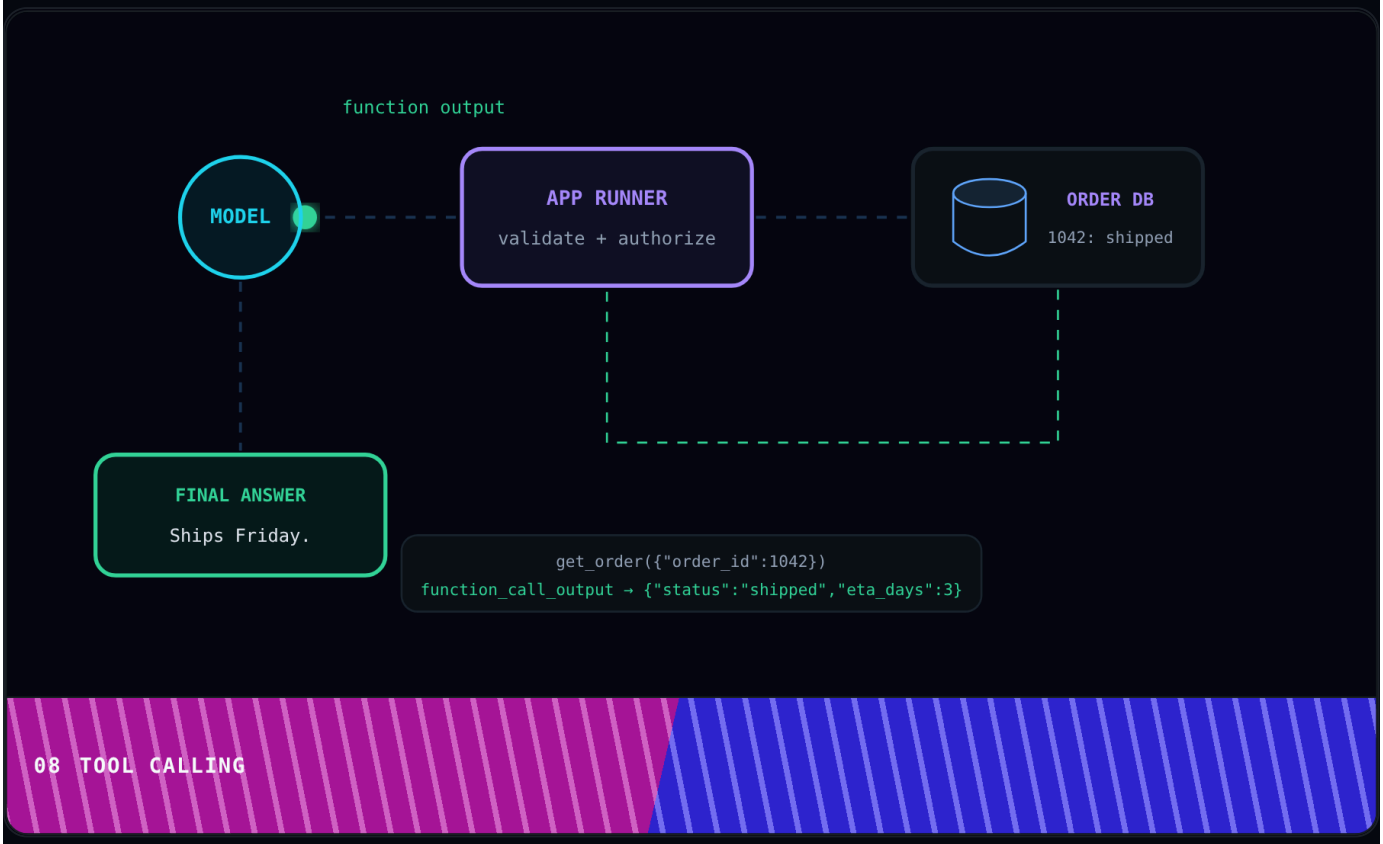
Our order schema requires an integer id, an approved status, and a day count that is zero or more. These rules are part of the product contract.

A response can be valid JSON and still put “maybe” in the status field, so is successful JSON parsing enough to protect the app?

No, because choosing Strict schema in Format blocks the bad status from its enum slot while the correct order object fits every typed field.

Use schemas when your code needs typed data, and handle refusals separately, so next, we will let the model ask a tool for current information.

Tool Calling



The schema shapes the reply, but the model still needs current order data, so to make that possible, the app includes tool definitions in the request.

The model returns a function call with a name and arguments. Notice what has not happened yet: the model requested work, but it did not run database code.

Your app validates the arguments, checks permission, and runs the real tool and choose Weather in Tool to see the same loop with another allowed function.

The tool result now exists outside the model, so how does that new fact get back into the answer?

The app sends a function call output item back to the model and only after this return step can the model write the final user-facing answer.

So your code owns the tool loop, not the model. Validate inputs, permissions, and effects every time. Next, we will choose a model for each workload.

STAGE 9 OF 12

Model Routing

09 MODEL ROUTING

The tool loop can use many models. Instead of starting with a leaderboard, start with the job and measure the quality, latency, and cost that job needs.

A fast tier can handle repeated classification. A balanced tier works for many schema tasks, while a reasoning tier may be worth the cost for hard planning.

The strongest model can probably do every job, so so why not send every request to it?

Change Workload and watch each job move across the same three gauges until it reaches the cheapest model tier that still passes its evals.

Route using eval results, and pin model snapshots when stability matters, so next, we will handle rate limits, temporary failures, and slow networks.

STAGE 10 OF 12

Retries + Timeouts

10 RETRIES + TIMEOUTS

Even the right model can hit a rate limit, get a temporary network error, or run past your deadline, which means your application must handle those failures.

Retry only errors that might recover because exponential backoff creates longer waits, while random jitter stops many clients from retrying together.

The timeout is one deadline for the whole operation. Every wait and attempt uses part of it. An idempotency key protects side effects when a request repeats.

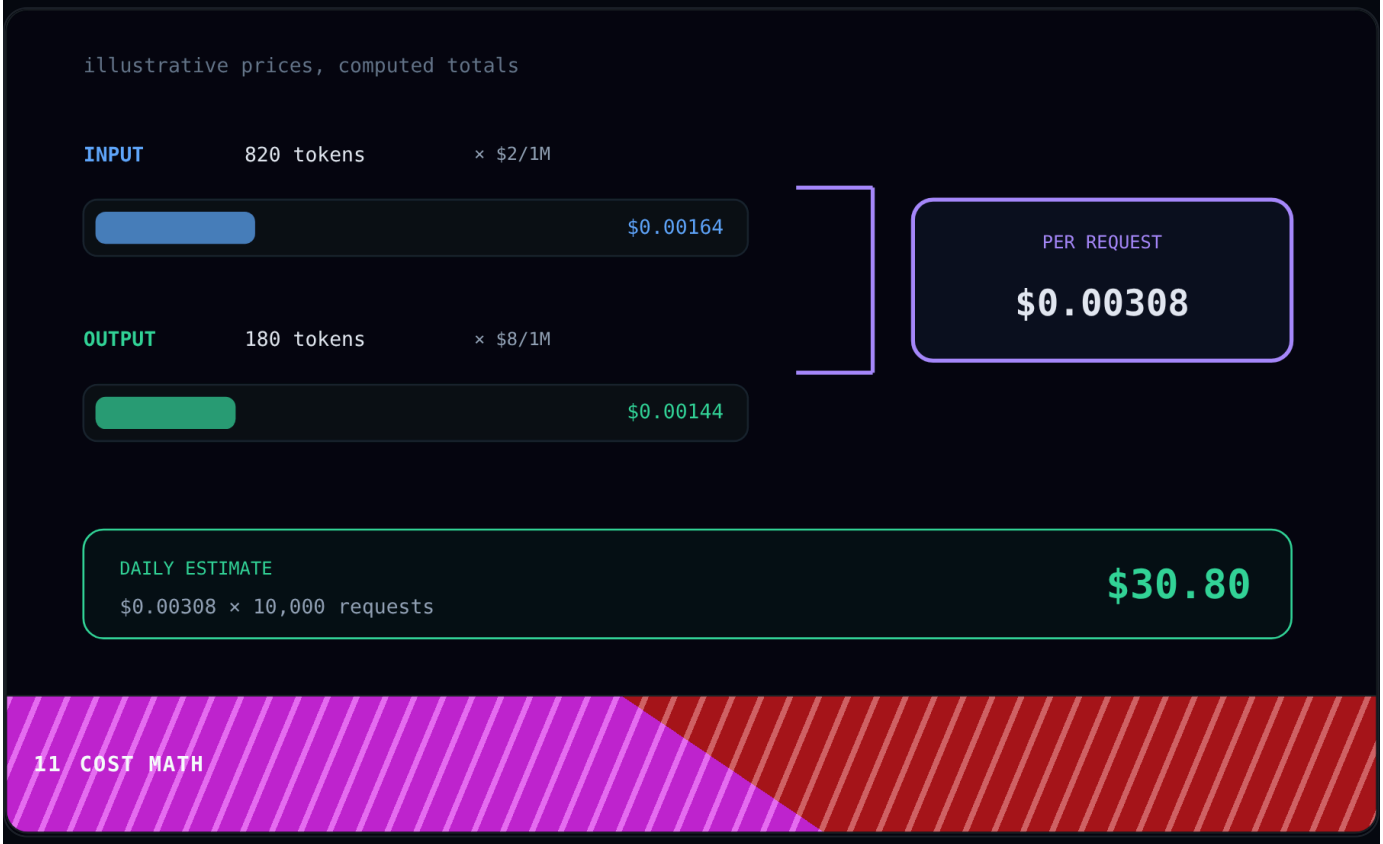
Now ask yourself: how many attempts with growing waits can fit before the deadline ends the operation?

Adjust Retries and Failure to see rate limits back off, successful calls stop immediately and timeouts refuse to schedule work beyond the red deadline.

Retries are limited chances to recover rather than permission to loop forever, so log request ids and honor deadlines before we calculate token cost.

STAGE 11 OF 12

Cost Math



The usage object tells us how many input and output tokens the request used and providers may charge different rates for each, so we need two meters.

This example uses $2 per million input tokens and $8 per million output tokens and these prices are illustrative, so load current rates for a real estimate.

One request uses 820 input tokens and 180 output tokens, so which side do you expect to add more to the bill?

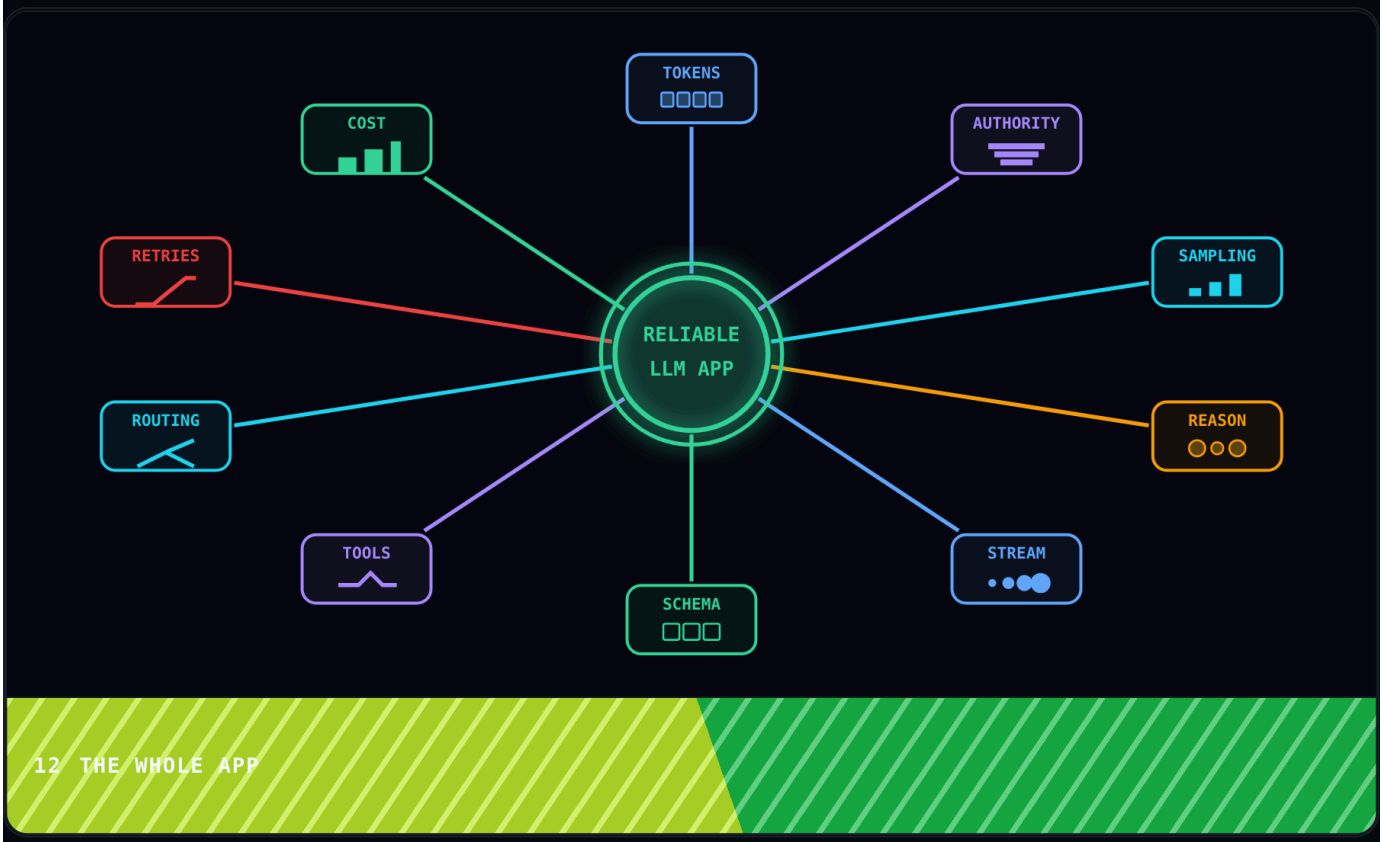
The bars are almost equal because output tokens cost more in this example. We compute the exact request cost by pricing the two counts separately and adding them.

Change Output size and Daily requests to see longer answers grow the costly output bar while traffic multiplies each request into a daily estimate.

So cost is token math times traffic, not a guess. Measure real usage and evaluate cheaper routes. Now let us connect the whole application.

STAGE 12 OF 12

The Whole App



12 THE WHOLE APP

We began with the context budget and rules, data, tool definitions, and output tokens all compete for space in one limited window.

Then we separated authority from trust and retrieved text can be useful data without becoming a trusted instruction.

Temperature changed the sampling odds and for stable production behavior, we also needed pinned model versions and evals.

Reasoning was the model's internal work, while the final answer stayed the public contract that application code could use.

Streaming sent ordered chunks early and the user saw useful text sooner, even though the model did not finish generation sooner.

Structured output gave model data a typed shape and we learned that valid JSON by itself is not enough.

Tool calling made a return loop and the model asked for work, but the application checked it, ran it, and sent the result back.

Model routing matched each workload to the cheapest tier that still met its quality, latency, and cost goals.

Retries used backoff inside one firm deadline and errors that could not recover stopped instead of looping forever.

Finally, we turned usage into cost with separate token rates and real traffic and the prices were illustrative, but every displayed total was calculated.

Here is the big picture: prompts matter, but dependable LLM apps come from budgets, trust boundaries, schemas, tools, evals, deadlines, and cost controls working together.