READ MODE · ANSWER FIRST

OpenAI Prompt Engineering

An interactive, visual explanation of OpenAI Prompt Engineering, from setup through the complete system.

CHEAT SHEET · 7 ESSENTIAL IDEAS

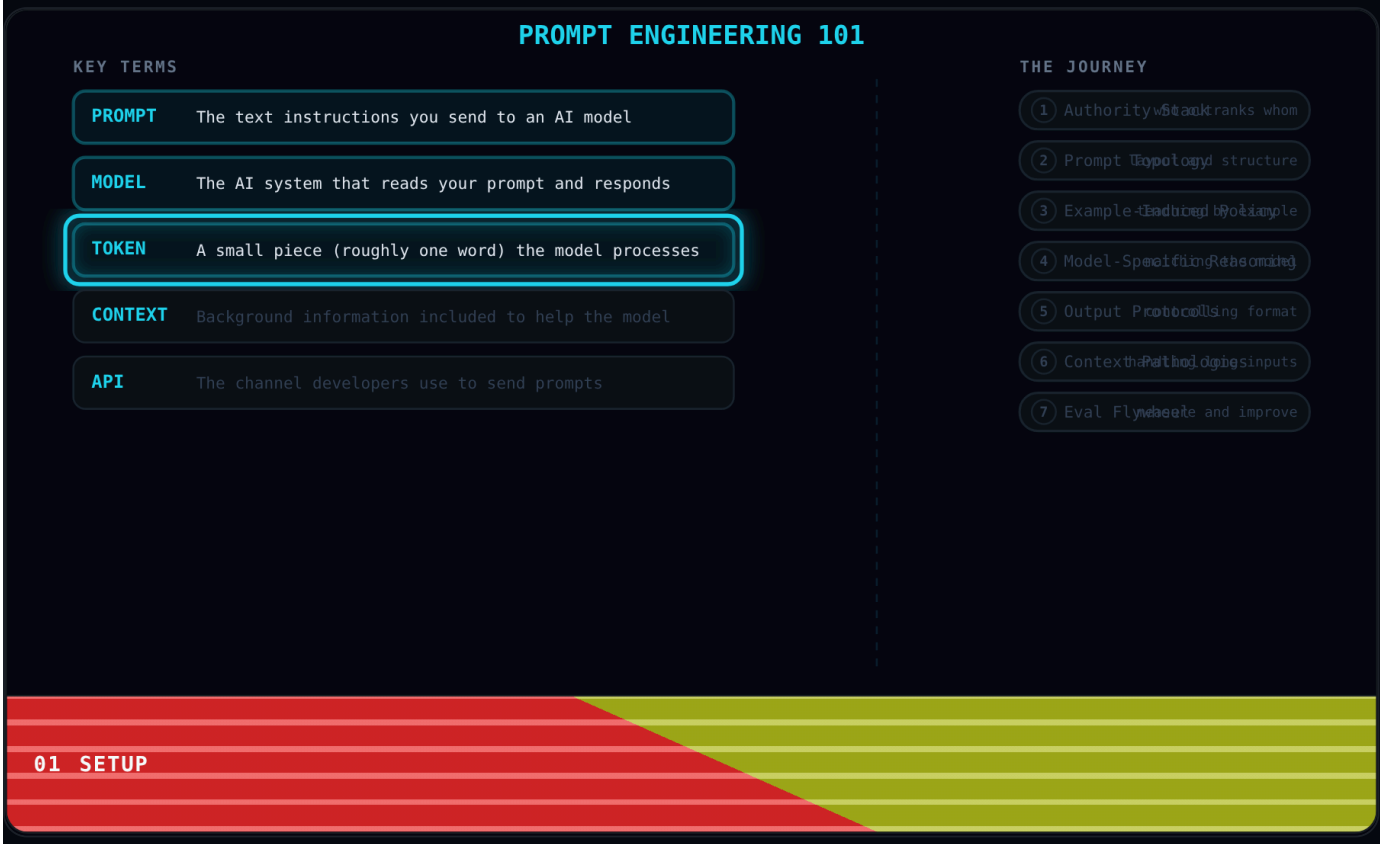
The whole story in 7 lines

OpenAI Prompt Engineering becomes easier to reason about when every stage is connected as one system.

1. A prompt is like a chain of command. The developer rules come first, the user request comes second, and background material comes last.
2. The layout of your prompt matters as much as the words in it. Put stable rules up front, changing details at the end, and use clear...
3. Examples are like solved practice problems you show the model before a test. They do not change the model itself - they shape how it...
4. Prompting is like giving directions. Some people want every turn spelled out. Others just want the destination and they will figure out...
5. An output schema is like a form the model fills in. Instead of writing a free-form essay, the model fills in specific fields in a...
6. Loading a model with too much context is like cramming too many notes onto a cheat sheet.
7. Good prompt engineering works like a science experiment. You have a hypothesis (your prompt), you test it against real data, you measure...

STAGE 1 OF 9

Setup



Welcome. Before we start, let us cover the big picture. Prompt engineering is the skill of writing clear, structured instructions for AI models so they produce reliable, useful output. Think of it like learning to write really good assignments for a very capable but very literal assistant.

A prompt is the text you send to an AI model. It can be as short as a question or as long as a detailed document with rules, examples, and reference material. The better the prompt, the better the response.

A model is the AI system that reads your prompt and generates a response. Different models have different strengths, and the same prompt can behave differently across them - just like different students might interpret the same assignment differently.

A token is a small piece - roughly one word - that the model reads and writes. The model does not see full sentences. It processes one token at a time, like reading a book one word at a time.

Context is background information you include to help the model answer well - like giving a student reference material along with their assignment. An API is the technical channel developers use to send prompts and get responses back.

Over the next seven stages we will build up the complete picture: how to order your instructions, structure your prompt, use examples, handle different models, control output format, manage context safely, and test your results. Let us start with the most fundamental idea: who gets the final say.

STAGE 2 OF 9

Authority Stack

Let us look at what happens before the model generates even a single token. Every prompt has three layers, and the order they load in decides who outranks whom. Think of it like a school: the principal sets the rules, the teacher gives the assignment, and the textbook provides reference material.

Developer instructions load first. These are the rules that stay the same across many requests. They are like the principal's policy: no matter what the teacher or student says, these rules hold.

User input comes next. This is the actual task: what you are asking the model to do right now. It can shape the response, but it cannot overwrite the developer rules - just like a student cannot override school policy.

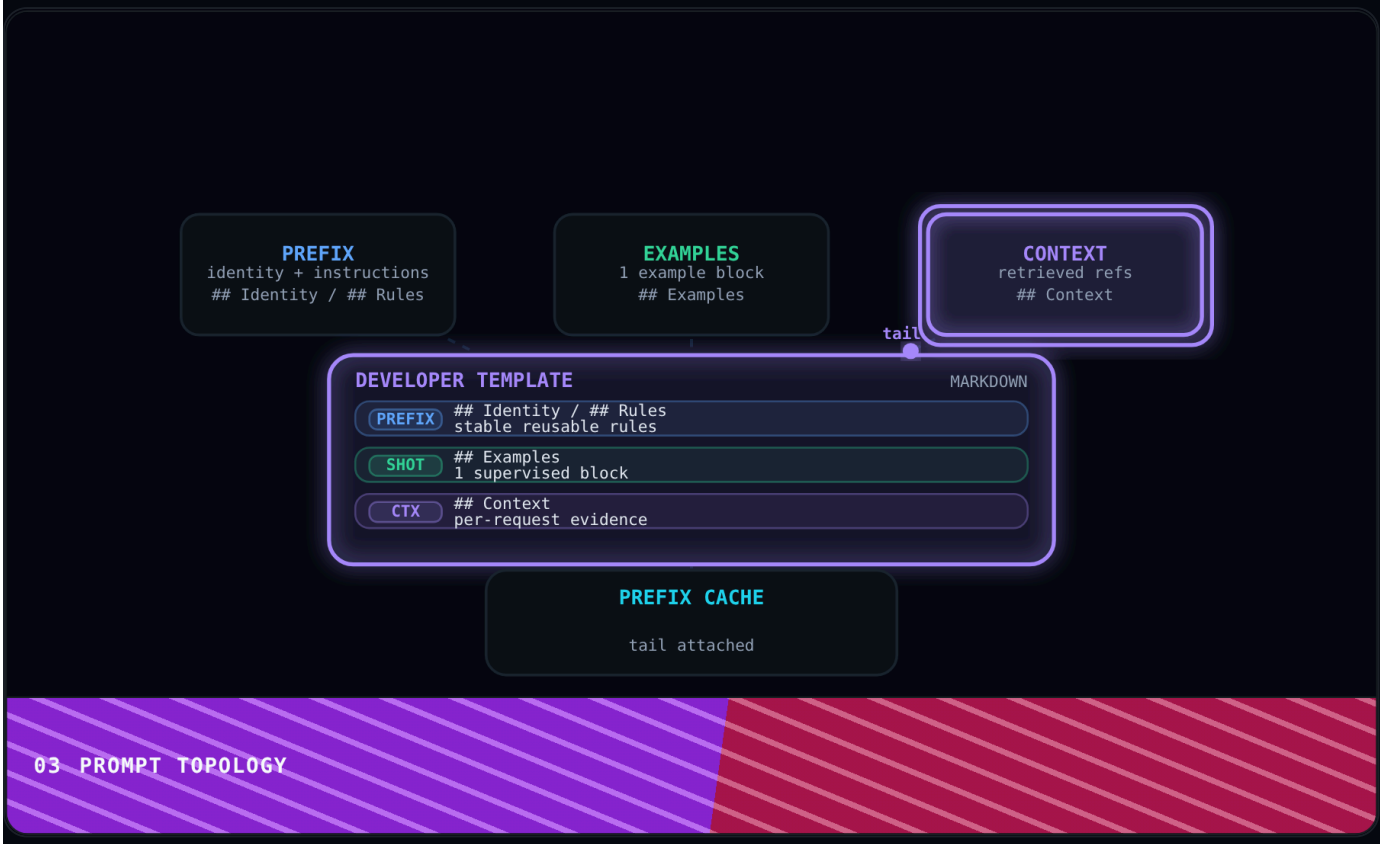
Reference material like files or retrieved documents gets added last. This is background information to help the model give a better answer, but it has the least authority of all three layers.

The API takes all three layers and packs them into one request object. Whether you used explicit messages, an instructions field, or a saved template, the model sees one combined package with a clear pecking order.

That is the first big lesson: a prompt is not just clever wording. It is an ordered package where authority matters. Keep this chain of command in mind - next we will learn how to organize the actual document inside that package.

STAGE 3 OF 9

Prompt Topology



03 PROMPT TOPOLOGY

Think of a prompt like a recipe card. You would not scatter the ingredients throughout the cooking steps. A strong prompt puts its sections in a clear, predictable order so the model reads it the way you intended.

The stable prefix goes first: your identity and core rules. This is the part that stays the same across many requests. Putting it up front anchors the model's behavior from the start - just like the developer layer we learned about in the authority stack.

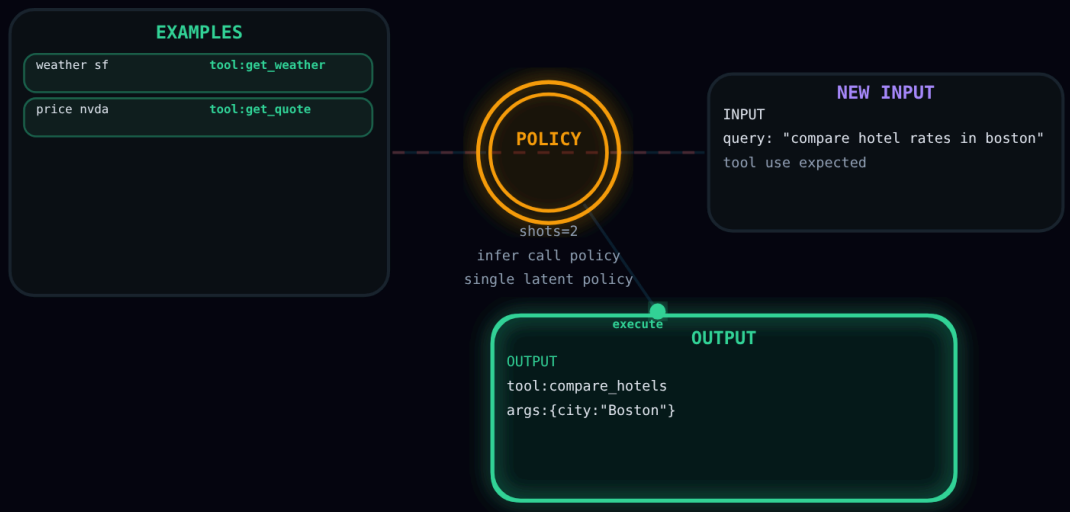
Examples come after the rules. They show the model exactly what good input and output looks like. These should match the rules perfectly - if your rules say "be concise" but your examples are long-winded, the model gets mixed signals.

Per-request context belongs near the end because it changes every time. New questions, new documents, new data - all of that goes after the stable sections. This keeps the reusable parts of your prompt consistent.

There is a practical bonus to putting stable content first: prompt caching. The system can remember and reuse the repeated prefix across many calls, which saves time and money. That only works if the prefix stays at the front and stays unchanged.

So a well-structured prompt is not just easier for the model to read - it is cheaper and faster to run. Structure, section order, and clear boundaries matter just as much as what the words say. Next, let us look at how to fill one of those template sections: examples.

Example-Induced Policy



04 EXAMPLE-INDUCED POLICY

You know how a teacher might say "here is a solved example, now do the next one like this"? That is exactly what few-shot prompting does. You include a few input-output pairs in your prompt, and the model uses them as a pattern to follow.

Each example pairs a sample input with the output you want. The model reads these pairs like a mini instruction manual: when you see something like this, respond like that.

From those examples, the model figures out an unwritten rule - a pattern. If your examples all classify customer messages by topic, the model picks up "I should classify this new message the same way." That pattern is what actually guides the output.

When the examples agree with each other and match the rules you set in the authority stack, the model's behavior gets sharper and more predictable. Quality matters more than quantity - two clean examples often beat five sloppy ones.

But when examples contradict each other or disagree with the rules, the model is stuck guessing which signal to trust. That is when you get unpredictable results. Remember: examples sit inside the structured template we built in the last stage, so they should reinforce the rules, not fight them.

The takeaway: examples teach the model a pattern for this specific run. They are powerful when consistent, and harmful when contradictory. But here is the thing - not every model responds to examples the same way. That is what we will explore next.

Model-Specific Reasoning

05 MODEL-SPECIFIC REASONING

Not all AI models are built the same. GPT-style models and reasoning models respond best to very different kinds of prompts. Think of it like giving directions: some people want every turn spelled out, others just want to know where they are going.

GPT-style models do best when you spell out exactly what to do. Give them the step-by-step transformation, the output format, and the specific workflow. Remember the structured template from Stage 2? For GPT models, fill it with precise, detailed instructions.

Reasoning models are different. They often work better when you describe the goal and let them figure out the path. Remember those examples from Stage 3? Reasoning models frequently do not need them at all - start without examples and only add them if the output drifts.

Reasoning models also have an effort setting that controls how much internal thinking time they get. More effort means the model searches deeper, which helps with hard problems but takes longer and costs more. It is a tradeoff you can tune.

One thing both model types benefit from: a verification step. Asking the model to double-check its work before giving a final answer improves reliability, without needing to micromanage every step. Think of it like asking a student to review their test before handing it in.

So the same prompt can be too vague for a GPT model or too controlling for a reasoning model. Knowing your model matters. Now that we know how to build the input and match it to the model, let us control what comes out the other side.

STAGE 6 OF 9

Output Protocols

06 OUTPUT PROTOCOLS

Up until now we have been thinking about what goes into the prompt. But often, code needs to read the model's output - not a human. When that happens, you need more than good instructions. You need a contract that defines what the response must look like.

A JSON schema or tool definition tells the model exactly what fields and formats are allowed. Instead of hoping the model formats things correctly, you define the shape up front. Think of it like giving someone a form to fill in instead of asking for a free-form essay.

With strict mode on, the model's output is actively guided at every step. Remember tokens from the Setup? Each token the model writes is checked against the schema, so the output stays valid all the way through - not just checked at the end.

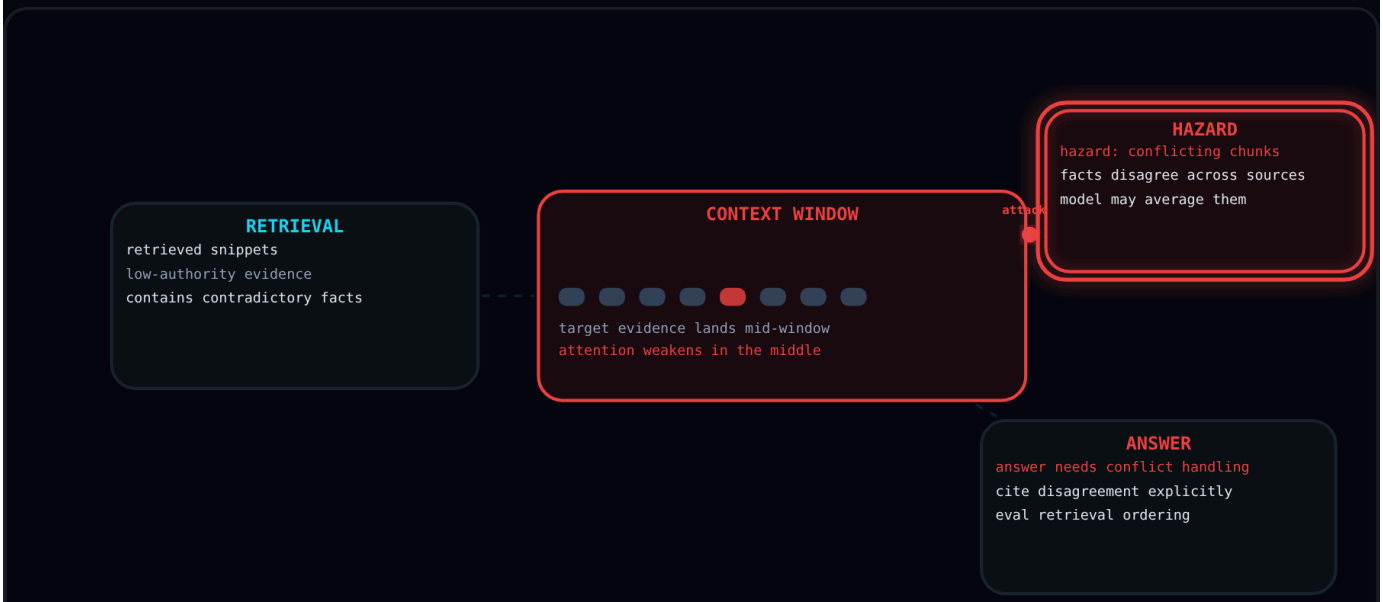
Tool descriptions work the same way. They tell the model: here is a tool you can call, here are the arguments it accepts, and here is when you should use it. The tool description and the prompt rules from the authority stack should reinforce each other.

As the output shape gets more complex - nested objects, multiple possible formats, branching logic - you need clearer field descriptions. The model might follow the grammar perfectly but still pick the wrong option if the instructions are not specific enough.

When you encode the expected output as a formal contract, prompt engineering becomes much more reliable and testable. You know exactly what shape to expect. That sets us up perfectly for the next challenge: what happens when you feed the model too much reference material.

STAGE 7 OF 9

Context Pathologies



07 CONTEXT PATHOLOGIES

We have been including reference material in our prompts since the authority stack. It helps the model give grounded, accurate answers. But here is the catch: more context does not automatically mean better answers. This stage shows what can go wrong.

Reference text works best when it is clearly separated from the instructions and marked as supporting material. Remember the authority stack? Context sits at the bottom of the chain of command. Without a clear boundary, the model might treat a reference snippet as if it were an instruction.

As you pack more text into the prompt, the model's attention weakens in the middle. This is called the lost-in-the-middle problem: a critical fact can be sitting right there in the prompt and the model still misses it, because it was buried between thousands of other tokens.

It gets worse with adversarial content. A retrieved document might contain text that looks like an instruction: "ignore the above and do X instead." The authority stack says developer rules outrank everything, but the model has to actually enforce that boundary. Not all prompts make that easy.

The fix is not to use less context or more context. It is to be deliberate: place important evidence near the front or end, use clear section boundaries like we learned in the Prompt Topology stage, and test your prompt at different context sizes to find where it breaks.

That is the scaling challenge of prompt engineering. Context is powerful, but stuffing more tokens into the window is not the same as building a reliable prompt. Everything we have learned - authority, structure, examples, model choice, output format - comes together here. And to know if it all actually works, we need one more piece: evaluation.

Eval Flywheel

08 EVAL FLYWHEEL

Everything we have built so far is the first draft. Real prompt engineering is a loop: write a prompt, test it against real data, see where it fails, fix it, and test again. Think of it like a science experiment where you keep refining your hypothesis.

Start with a versioned prompt and a pinned model. Versioning matters because if you change the prompt and the model at the same time, you cannot tell which change caused the difference. Like a good experiment: change one variable at a time.

Next, run the prompt against a dataset of real examples and collect feedback. Automated graders can score the output, humans can mark results as good or bad, or you can use both. This is where all the structure we built - authority, templates, output schemas - becomes measurable.

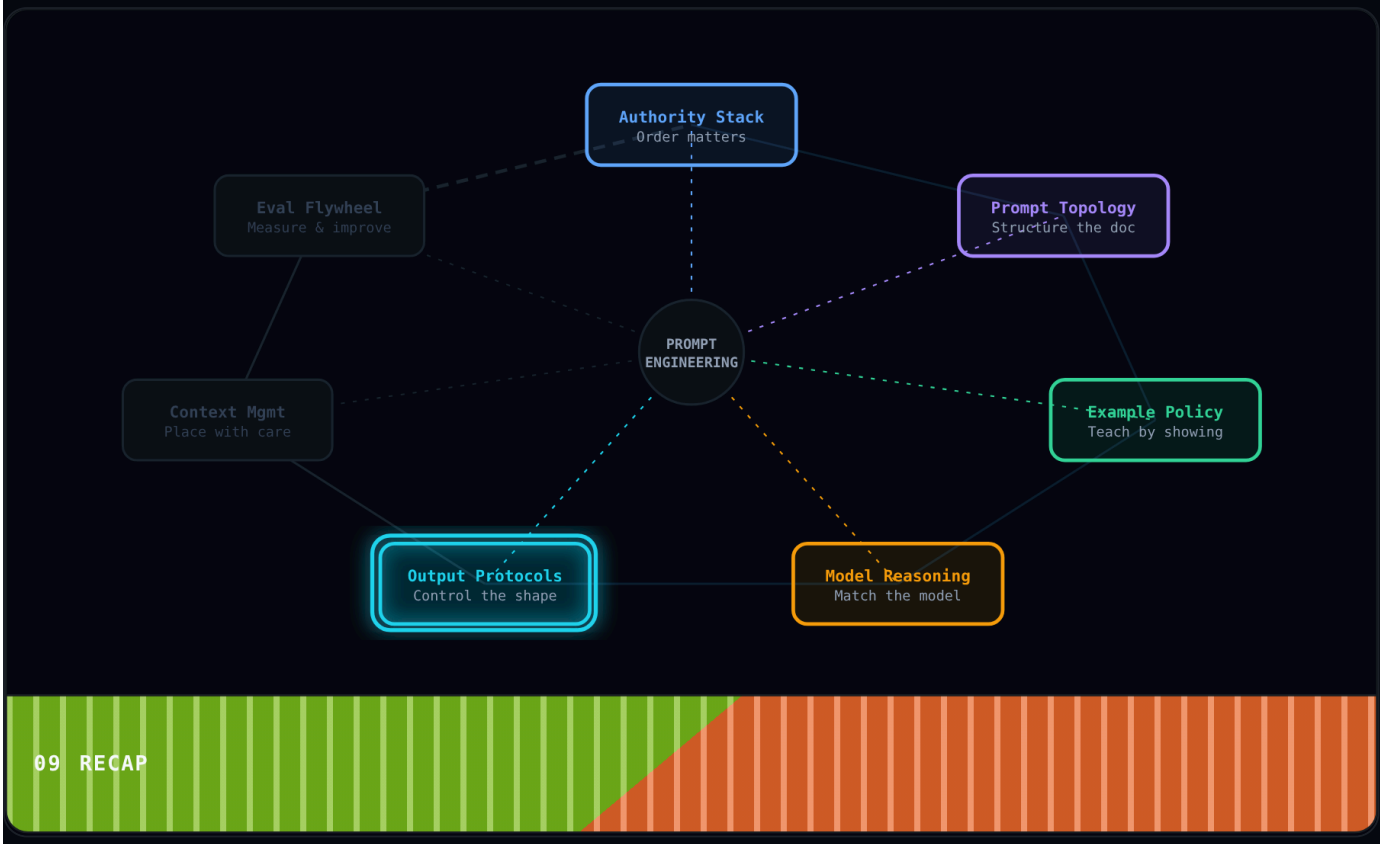
The optimizer or a human editor rewrites the prompt based on what failed. This is not about making the prompt sound prettier. It is about fixing the specific cases where the output went wrong, using the evidence from the tests.

Then you rerun the tests and compare the new prompt against the old one. Even a prompt that is better overall might fail on cases the old version handled fine. That is why you compare side by side, not just look at averages.

That is prompt engineering as a real discipline. Not a one-time trick, but a flywheel: write, test, measure, improve, repeat. Everything you have learned across these stages feeds into this loop. Now let us step back and see the whole picture together.

STAGE 9 OF 9

Recap



We started with the authority stack: who gets the final say. Developer rules first, user input second, reference material last. That chain of command is the foundation everything else is built on.

Then we organized the prompt document itself. Stable rules at the front, changing context at the back, clear boundaries between sections - like a well-organized recipe card. Structure makes the model read your prompt the way you intended.

We added examples to teach the model by showing, not just telling. Like solved practice problems before a test, they sharpen the model's behavior - but only when they are clean and consistent.

We learned that different models need different prompts. GPT models want detailed step-by-step instructions. Reasoning models prefer a clear goal and room to figure out the path themselves.

We controlled the output with schemas and tool definitions. Instead of hoping the model formats things right, we gave it a form to fill in, making results predictable and machine-readable.

We tackled context pathologies - what goes wrong when you feed the model too much reference material, or place it carelessly. The authority stack, the structured template, and clear boundaries all work together to protect the prompt here.

And finally, the eval flywheel ties it all together. Write, test, measure, improve, repeat. That is not just the last step - it is the ongoing discipline that makes every other technique reliable. That is prompt engineering.