Ways to change what a language model does

Seventeen techniques, treated on equal terms: what each one changes, what it costs, and how to work out which one your problem actually calls for.

Reference sheet September 2026 Written for people making build decisions, not for ML engineers

Start here

Choose by what you're trying to fix

Almost every argument about how to improve an AI system is really an argument about what kind of problem is in front of you. These are a mix of goals, symptoms and constraints, grouped accordingly. Find yours in the left column first; the method follows from it.

If this describes your situation	Start with	Why
What you want to be true		
It needs to know something it doesn't know	**Retrieval**	Facts belong in documents you can edit, not in weights you have to retrain.
It must show where the answer came from	**Retrieval**	Only retrieval gives you a citable source. Trained-in knowledge has no provenance.
You need it to behave differently just for this task	**System prompt**	Cheapest possible change, and reversible in seconds.
Output must be valid JSON or match a schema, every single time	**Constrained decoding**	It makes malformed output structurally impossible rather than merely unlikely.
It needs to do something in the world, not just describe it	**Tool use**	Booking, looking up, calculating, writing to a system of record.
It must never say a particular kind of thing	**Guardrail classifier**	A separate checker you can update independently, without touching the main prompt.
It should carry knowledge of a person across sessions	**Memory / per-user state**	Personalisation is a storage problem, not a training problem.
You want your team's working conventions applied consistently	**Loadable instruction sets**	Editable by the people who own the conventions, with no engineer in the loop.
You need something working by this afternoon	**Few-shot examples**	Nothing else on this page has a shorter path from idea to working.
What is going wrong		
It follows the style guide most of the time, but drifts on the awkward cases	**Adapter (LoRA)**	Prompts influence output; they don't change the model's underlying tendencies. That last stubborn fraction is a distribution problem.
Same input, noticeably different answers	**Decomposition, then adapter**	Break the task into smaller checkable steps first; train only if the variance survives.

If this describes your situation	Start with	Why
It falls apart on long, multi-part tasks	**Workflow decomposition**	Reliability comes from smaller steps far more often than from better models.
It invents facts that sound plausible	**Retrieval, plus a grounding check**	Give it the source material, then verify the answer against what was actually retrieved.
Prompts have grown to thousands of tokens and every call pays for them	**Prompt caching**	Often removes the cost argument entirely. Try it before training anything.
It is too expensive at volume	**Caching → routing → distillation**	In that order. Each step is more work than the last.
It is too slow for the interaction you want	**Smaller model plus distillation**	Teach a fast cheap model to imitate the slow expensive one on your narrow task.
A model upgrade broke things that used to work	**An evaluation set**	Without one you cannot see what broke, and every upgrade becomes a gamble.
Nobody can tell whether the last change made things better	**An evaluation set**	This is the real blocker, and no technique on this page fixes it.
What constrains you		
The information changes weekly, or daily	**Retrieval**	A training run is stale the moment it finishes. A document store isn't.
Requirements are still moving around	**System prompt**	Never train against a spec you're still arguing about.
The output shape is easier to show than to describe	**Few-shot examples**	Three good examples beat three paragraphs of instruction.
The people who own the rules are not engineers	**Prompt or instruction sets**	Whoever owns a decision should be able to change it without a deployment.
Two hundred customers each want slightly different behaviour	**Multi-adapter serving**	One base model on one GPU, a small swappable file per customer.
There's no single right answer, only better and worse ones	**Preference tuning (DPO)**	Train on comparisons, because you can judge which draft is better even when you can't write the ideal one.
Correctness can be checked automatically	**RL with verifiable rewards**	If you can write the grader, you can train against it directly.
The domain has vocabulary and structure the model has barely seen	**Continued pretraining**	Rare, expensive, and a genuine foundation-model project.

The full set

Grouped by what you are actually changing

Four layers sit between a base model and a working product. Almost every technique changes exactly one of them, and knowing which layer you're touching is more useful than knowing a technique's name.

Technique	What it is	Strengths	Weaknesses	Best real-world use
Changing the context — what goes into each call				

Technique	What it is	Strengths	Weaknesses	Best real-world use
System prompting	Instructions written in ordinary language at the top of every call.	Instant to change. No training cost. Legible to non-engineers, so the people who own the rules can edit them. Easy to A/B test.	Consumes context on every request. Adherence degrades on long or adversarial inputs. Gets brittle as instructions pile up and start contradicting each other.	Tone of voice, safety rules, task framing. Anything you are still working out.
Few-shot examples	Three to twenty worked examples placed inside the prompt.	The fastest way to pin down an output format. No infrastructure at all. Ideal when a rule is easier to demonstrate than to articulate.	Expensive at scale, since you resend them every time. Example choice biases output hard. Returns flatten out somewhere around twenty examples.	Structured extraction, classification with fuzzy edges, matching a specific output shape.
Retrieval (RAG)	Search your documents at question time and put the relevant ones into the prompt.	Knowledge stays current with no retraining. Gives citations and provenance. A correction is a document edit, made by whoever owns the document.	You are now building and debugging a search system. Answer quality is capped by retrieval quality. Changes knowledge, not behaviour.	Policy and documentation questions, anything where "as of today" matters, anything that must be attributable.
Loadable instruction sets	Folders of procedures the model pulls in only when they're relevant.	Specialisation that non-engineers can write and edit. No training. Only loaded when needed, so it doesn't tax every call.	Still consumes context when active. Doesn't change behaviour at the level of weights, so the same drift applies.	Team playbooks, repeatable document formats, house conventions.
Memory and per-user state	Persistent facts about a person or account, carried between sessions.	Personalisation with no training at all. Improves with use rather than degrading.	The hard part is consent and privacy design, not the storage. Stale memories actively mislead.	Assistants and long-running professional relationships.
Prompt caching	The provider caches your long static prefix so you don't pay full price to resend it.	Large cost and latency savings for no training and almost no work. Changes nothing about behaviour, so it carries no risk.	Only helps when the prefix is genuinely static. Still uses up context window.	Long system prompts and fixed reference material at high volume.
Changing the generation — how output is produced				
Constrained decoding	Output is forced to match a schema or grammar as it is generated.	Total format compliance, with no training and no retries.	Controls shape only, never content quality. A perfectly formatted wrong answer is still wrong.	JSON APIs, form filling, anywhere malformed output breaks something downstream.
Tool use	The model calls external systems instead of knowing things itself.	Real-time data and real actions. Each step is auditable. The source of truth stays where it belongs.	More failure modes and more latency. Depends heavily on well-written tool descriptions.	Bookings, lookups, calculations, anything with an authoritative system elsewhere.
Changing the weights — what the model itself is				
Adapters (LoRA)	Small trained files, often tens of megabytes, sitting on a frozen base model. Many can be swapped per request on one GPU.	Shifts underlying tendencies rather than instructing against them, so behaviour holds on awkward inputs. Short prompts. Many variants for the cost of one model.	Needs hundreds to thousands of examples per adapter, plus an evaluation set. The change loop is days. Goes stale as the base model and the world move on.	Per-customer document processing, high-volume structured output, domain jargon and house style.
Distillation	A large model generates training data; a small model learns from it.	Steep drop in serving cost and latency, often an order of magnitude.	Quality is capped by the teacher. Narrow by construction. Usually needs the same training apparatus as adapters.	Turning an expensive prototype into an affordable production service.

Technique	What it is	Strengths	Weaknesses	Best real-world use
Preference tuning (DPO, RLHF)	Training on pairs of better and worse outputs rather than single correct answers.	Captures taste and judgement in places where no single right answer exists.	Needs comparison data, which is laborious to collect. Subtle to evaluate, and easy to over-optimise into blandness.	Making output feel right rather than merely be right. Editorial quality, helpfulness.
RL with verifiable rewards	Training against an automatic checker — tests pass, the arithmetic is correct.	Very strong wherever correctness is machine-checkable, and improvement can continue without more human labelling.	Only possible when you can actually write the checker. The model will exploit a sloppy one.	Code generation, formal reasoning, structured tasks with a grader.
Full fine-tuning	Retraining all the weights, not just a small add-on.	The deepest behaviour change available short of pretraining.	Expensive, needs real ML skill, produces one model per use case, and easily damages general ability.	Rare in practice. A large organisation with a genuinely distinct domain and a lot of data.
Continued pretraining	Further unsupervised training on a large domain corpus.	Teaches genuinely new vocabulary and structure — legal, clinical, obscure codebases.	Very expensive and needs an enormous corpus. Months, not weeks.	Foundation-model work rather than application work.
Changing the system — how calls are arranged around the model				
Workflow decomposition	One hard task split into several small chained calls, each simple enough to be reliable.	The single biggest reliability gain available. Every step becomes testable and debuggable in isolation.	More engineering. Slower end to end. More places for something to break.	Complex document processing, multi-stage analysis. Frequently beats training outright.
Routing and cascades	A cheap model handles everything first; hard cases escalate to an expensive one.	Large cost savings. Effort is matched to difficulty rather than set once for everything.	Needs a reliable difficulty classifier, and you now maintain two paths.	High-volume support triage and any mixed-difficulty stream of requests.
Guardrail classifiers	Small separate models inspecting inputs and outputs.	Cheap, independently updatable, keeps the main prompt uncluttered. Fails safe.	Another moving part. False positives irritate users quickly.	Compliance, content safety, detecting personal data before it goes anywhere.
Evaluation sets	A fixed collection of test cases with known-good answers, run against every change.	The only thing that tells you whether anything else on this page worked. Makes model upgrades safe.	Tedious to build. Needs maintaining as the product changes. Generic frameworks rarely fit a specific problem.	Every system that anyone depends on. This is infrastructure, not a technique.

On the usual four. Prompting, few-shot, retrieval and fine-tuning get compared with each other constantly, but they aren't a natural set — three change the context and one changes the weights, and the comparison quietly omits decomposition, caching and evaluation, which more often turn out to be the answer. Treat the famous four as a historical accident of how the field discussed itself in 2023, not as a shortlist.

Diagnosis

Seven questions to ask before choosing anything

Work through these before reaching for a technique. In practice the answers narrow the field to two or three options, and the remaining choice is usually about who maintains it rather than what performs best.

Is the gap knowledge or behaviour?
Does it not know something, or does it know and act wrongly anyway? Knowledge problems go to the context layer. Behaviour problems go to instructions first, weights last.

How often does the right answer change?
If it changes weekly, nothing trained can hold it. If it hasn't changed in five years, training it in is defensible.

How many examples of "good" do you actually have?
Under fifty, you are prompting whether you like it or not. Several hundred opens up training. This question ends more debates than any other.

Can you tell good from bad automatically?
If not, you cannot measure improvement, and every technique here becomes a matter of opinion. Fix this before anything else.

Who needs to be able to change it, and how quickly?
A rule owned by the compliance team should not require a training run. Governance decides architecture more often than performance does.

What does a single failure cost?
A wrong tone in a blog post and a wrong figure in a regulated letter justify entirely different amounts of engineering.

Is the volume high enough for unit cost to matter?
At a thousand calls a month, optimise for speed of change. At ten million, unit economics start to dominate every other consideration.

Practicalities

What each one costs you

The comparison that usually decides things in the room, once the technical merits are agreed. The last column is the one people underestimate.

Technique	What you need to have	Time to make a change	Ongoing cost sits in	Who can actually do it
System prompting	An opinion, written down	Seconds	Tokens on every call	Anyone who can write clearly
Few-shot	Three to twenty good examples	Minutes	Tokens on every call	Anyone with domain knowledge
Loadable instruction sets	Written procedures	Minutes	Tokens when loaded	The person who owns the procedure
Prompt caching	A stable prompt prefix	Hours, once	Reduced token cost	A developer, briefly
Constrained decoding	A schema	Hours	Negligible	A developer

Technique	What you need to have	Time to make a change	Ongoing cost sits in	Who can actually do it
Memory	Somewhere to store state, and a consent model	Days	Storage and privacy upkeep	A developer with a privacy review
Retrieval	A document corpus worth searching	Days to build, minutes to update	Search infrastructure plus tokens	A developer to build, anyone to maintain content
Tool use	APIs that already exist	Days	Latency and integration upkeep	A developer
Guardrails	A definition of what must not happen	Days	An extra call per request	A developer with a policy owner
Evaluation sets	Fifty to two hundred cases with known-good answers	Days to build, then continuous	Maintenance as the product changes	A domain expert, with tooling help
Decomposition	A clear understanding of the task	Days to weeks	More calls per job	A developer working with a domain expert
Routing	Traffic and a difficulty signal	Weeks	Two paths to maintain	A developer
Adapters (LoRA)	Hundreds to thousands of examples, plus an eval set	Days per iteration	Training runs and GPU serving	Someone with ML experience
Distillation	A working large-model version to learn from	Weeks	Training, plus serving a second model	Someone with ML experience
Preference tuning	Thousands of ranked comparisons	Weeks	Data collection, mostly	An ML team
Full fine-tuning	A large curated dataset	Weeks	Training, serving and drift	An ML team
Continued pretraining	A domain corpus at serious scale	Months	Substantial compute	A research team

Sequence

The escalation ladder

This is a cost order, not a quality order — later steps are not better, only more expensive. Work down the list and stop at the first step that solves your problem.

1. **Build an evaluation set.**
   Out of order deliberately. Without it you cannot tell whether any later step worked.
2. **Write a better prompt.**
   Genuinely better, not longer. Most quality complaints die here.
3. **Add examples.**
   Three to twenty. Choose them to cover the edges, not the easy middle.
4. **Add retrieval, if the gap is knowledge.**
   Only if the model is missing facts rather than misbehaving.
5. **Decompose the task.**
   The step most often skipped and most often the actual fix.

6 **Turn on prompt caching.**
If cost is the complaint, this may end the conversation.

7 **Route cheap-first.**
Send the easy majority somewhere smaller.

8 **Train something.**
An adapter, or a distilled small model. Once you have examples, an eval set, and a spec that has stopped moving.

9 **Full fine-tuning or preference tuning.**
Only with a team who has done it before.

In combination

What mature systems actually run

These aren't rivals. Almost every serious production system runs several at once, each doing the job it's suited to, and the interesting design decisions are about the seams between them.

> **The assistant stack**
>
> Tools for anything real, memory for continuity, retrieval for documents, and a system prompt for character. Notice there's no training anywhere in it, and this covers a large share of what gets built.

> **The document-processing stack**
>
> Decomposition into per-field extraction, constrained decoding for the schema, retrieval for reference material, and an evaluation set covering the awkward document types. Training enters only if a specific step stays unreliable after all of that.

> **The high-volume stack**
>
> Caching on the static prefix, routing so the easy majority goes to a small model, distillation once the traffic justifies it, and the large model held in reserve for escalations.

> **The regulated stack**
>
> Retrieval so every claim has a source, guardrail classifiers on input and output, tool use for anything that touches a system of record, and an audit trail through the whole chain. Provenance is the design constraint, and it rules several techniques out.

> **The multi-tenant stack**
> One base model with an adapter per customer for their conventions, shared retrieval over each customer's own documents, and a routing layer in front. Worth it once the number of tenants makes per-tenant prompts unmanageable, and not before.

Watch for

The recurring mistakes

Training to teach facts

Training bakes in a snapshot with no provenance and no way to correct a single wrong fact. Facts go in retrieval.

Refusing to train on principle

The mirror error, and less discussed. Teams who treat all training as premature end up with six-thousand-token prompts nobody dares edit, and they pay for that on every call forever.

No evaluation set

Without one you cannot tell whether any change helped. Fifty to two hundred cases with known-good answers is enough to start, and it is worth more than any technique on this page.

Training before the spec stops moving

If the definition of "good" is still under discussion, every training run is a bet on an argument you haven't finished having.

Reaching for a model change when the problem is task design

"It's unreliable on long documents" is usually a decomposition problem wearing a model-quality costume.

Comparing against a weak baseline

Any technique that beats a lazy prompt has proved nothing. Beat your best prompt, with good examples and caching turned on.

Treating prompt tokens as free

They are free in a prototype and one of your largest line items at scale. The instinct formed during prototyping rarely survives contact with the bill.

Forgetting that everything ages

Adapters go stale as base models are replaced. Retrieval corpora rot. Prompts accumulate contradictions. Every choice here is a maintenance commitment.

Terms

Glossary

Base model
The large general-purpose model everything else sits on top of.

Token
Roughly three-quarters of a word. The unit you're billed in.

Context window
How much text the model can consider at once. Everything in the prompt competes for it.

Prompt prefix
The unchanging front portion of a prompt, before the user's actual input.

Zero-shot / few-shot
Asking with no examples, versus asking with a handful included.

Chain of thought
Asking the model to work through steps before answering, which usually improves reasoning.

RAG
Retrieval-augmented generation. Search first, then answer using what was found.

Chunking
Splitting documents into passages small enough to retrieve usefully. A surprisingly large share of retrieval quality lives here.

Embedding
A numeric representation of meaning, used to find passages that are relevant rather than merely word-matching.

Grounding
Tying an answer to retrieved source material, so it can be checked.

Schema
A definition of the structure an output must have.

Adapter / LoRA
A small file of extra weights trained on a frozen base model. It nudges behaviour rather than replacing the model.

Frozen
Weights left unchanged during training.

Distillation
Training a small model on the outputs of a large one.

DPO
Direct preference optimisation. Learning from pairs of better and worse answers.

RLVR
Reinforcement learning from verifiable rewards. Training against an automatic checker.

Eval set
A fixed collection of test cases with known-good answers, used to tell whether a change actually improved anything.

Model as judge
Using a model to score outputs. Only trustworthy once you've measured its agreement with human judgement.

Inference
Running the model to get an answer, as opposed to training it.

Drift
Behaviour degrading over time as the world, the data, or the base model moves on.

Further reading

Where to go deeper

What We've Learned From A Year of Building with LLMs
Six practitioners, organised into tactical, operational and strategic. The closest thing the field has to a canonical text, and free. applied-llms.org

Your AI Product Needs Evals — Hamel Husain
The argument that failed AI products almost always share one root cause: no robust evaluation system. Read this before building anything you'll depend on. hamel.dev/blog/posts/evals

Optimizing LLM Accuracy — OpenAI
The two-axis model: is the problem what the model needs to know, or how it needs to behave? Same distinction as this sheet, more prescriptive. developers.openai.com

Building Effective Agents — Anthropic
On decomposition, and on the discipline of finding the simplest solution and only adding complexity when needed. anthropic.com/engineering/building-effective-agents

Effective Context Engineering for AI Agents — Anthropic
Reframes prompting as a resource-allocation problem: what configuration of context produces the behaviour you want. anthropic.com/engineering

AI Engineering — Chip Huyen (O'Reilly, 2025)
A book-length framework for adapting foundation models, with the questions to ask when evaluating which approach fits. github.com/chiphuyen/aie-book

> Most of the above was written between 2024 and early 2025. Prompt caching, longer context windows and loadable instruction sets have shifted the economics since. The mental models hold up better than the specific cost advice.

One rule underneath all of it: name the layer you are changing before you name the technique. Context, generation, weights, or the system around them. Most bad decisions in this area are made by reaching for a familiar technique before working out which layer the problem lives in.