4. Post-Training

Table of contents

1 Overview: From Base Model to Assistant

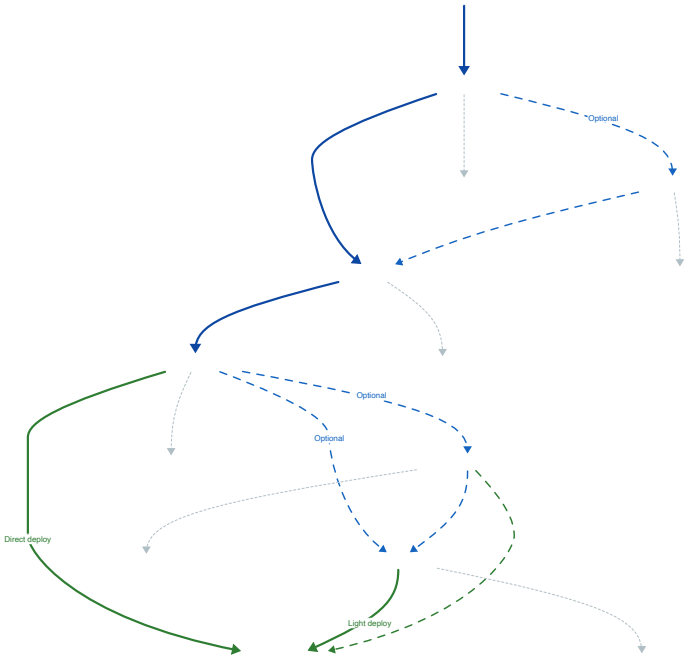

Figure 1: The post-training pipeline from base model to production assistant

Imagine your team plugged a base model that "looked smart" into the finance system on Friday. By Monday morning, three failures appeared at once: the accounting interface expected strict JSON, but it returned prose; when it should

have called the balance-query tool, it guessed a number instead; and when faced with a high-risk transfer request, its refusal language became noticeably softer than it had been in the test environment.

These three failures do not mean the model "doesn't understand language." They mean it **has not yet been post-trained into a production assistant**. What base models are good at is next-token prediction: continuing text, imitating tone, extending statistical patterns. What production systems need is a different set of abilities: following instructions, respecting boundaries, calling tools, producing structured output, passing validation, and controlling cost.

So post-training is not merely "fine-tune it a bit after pretraining." It is closer to a layer of systems engineering: you are simultaneously rewriting the training distribution, the optimization objective, the decoding constraints, the tool interfaces, the evaluation loop, and the deployment cost. That is also why alignment is never just a model problem. It is a **systems problem jointly determined by data, the objective function, tool contracts, the serving stack, and the evaluation stack**.

The central engineering question of post-training can be compressed into one sentence:

> **How do you gradually turn a base model that only predicts text into an AI assistant that is reliable, controllable, and deployable?**

1.1 Chapter Overview

You can think of post-training as the process of taking "a model that can talk" and shaping it into "a system assistant that can work reliably." The "phases" in this chapter refer to a typical post-training pipeline, not the full lifecycle of an LLM. In real engineering work, teams will reorder phases or even skip some entirely depending on the goal—for example, doing tool-use SFT first and preference alignment later.

The prerequisites can be reduced to three blocks. First, you should know what the Transformer and the next-token objective are doing. Second, you should be familiar with the fine-tuning workflow, including data splitting, training, validation, and regression. Third, you should be able to read service metrics such as latency, cost, reliability, and failure rate. If you have those three blocks, this chapter will be much easier to read deeply.

The learning goal is equally practical: after reading it, you should be able to explain what each of the six phases solves, identify the first-priority lever under real product constraints, locate the source of a failure and propose a repair path, and finally turn a training plan into an executable go-live evaluation gate.

A good way to read this chapter is to follow one fixed line: first see where the current system is stuck, then see why the mainstream solution is not enough,

then see which layer the new method changes—training signal, objective function, inference constraint, or system structure—and finally evaluate the benefit, cost, and applicability boundary. Read it in that order and you will not stop at "remembering the terms." You will be able to answer "why use it," "when use it," and "when not to use it."

When you read formulas, you do not need to derive every detail immediately. Start with four things: who is being optimized, over what range the average is taken, whether there is a penalty term, and which hyperparameter controls the strength. In this chapter, common symbols can be read like this: θ is the model parameter, x is the input, y is the output, t is the token position, $\sum$ denotes summation, $\mathbb{E}$ denotes expectation, log is often used to rewrite products of probabilities as sums, and KL measures the difference between two distributions. Once those four pieces are clear, most formulas collapse into intuition.

To grasp the whole picture before you begin, keep this main line in mind: first use SFT to stabilize behavior and format, then use PEFT for low-cost specialization; next do preference alignment and tool training to upgrade "can talk" into "can act"; then use reasoning RL to pursue verifiable correctness; and finally use distillation to bring the capability down to acceptable latency and cost.

This chapter uses one running case to connect all phases: you are building a financial operations assistant. It must output strict JSON to downstream systems, must call tools to access real-time or private data, must refuse high-risk actions such as unauthorized transfers, and must support policy differences across multiple business lines in a multi-tenant setup. A useful habit is to ask yourself two questions while reading each phase: what class of problem does this step solve for that assistant, and what new kind of engineering risk does it introduce?

Overview of the post-training phases

Phase	Common role	Main problem it fixes	Representative artifact / systems lever
SFT	Usually required	Instruction following, format control, stable roles and boundaries	chat template, completion-only mask, SFT checkpoint
PEFT	Common but optional	Low-cost specialization, multi-version management, multi-tenant differentiation	LoRA / QLoRA adapter, adapter routing

Phase	Common role	Main problem it fixes	Representative artifact / systems lever
Preference alignment	Usually required	Quality differences where "not bad" is still "not good enough"	preference pairs, reward model, DPO / RLHF policy
Tool use / RAG	Required for most products	groundedness, real-time / private data access, structured actions	tool traces, schema, validator, constrained decoding
Reasoning / agent RL	Optional by task	multi-step planning, verifiable correctness, search and self-correction	verifier, process reward, GRPO-style training
Distillation	Frequently required	latency, cost, deployment footprint	teacher-student pipeline, student model

i An Intuitive Analogy for Post-Training

How to train a glib stranger into your ideal boyfriend

A base model is a familiar type: **its expressive ability far exceeds its judgment.** This kind of person usually makes a very good first impression. He reacts quickly, has a lot of words, can answer anything, and can even imitate different styles of speech. Talk to him for ten minutes and he seems smart. Ask him to get one consequential thing right, and you realize you praised him too early.

This is not a rare problem. A language model's problem is that it looks like it understands and sounds like it understands, when in fact it is only producing the statistically most plausible next sentence. A base model's problem has never been that it cannot talk. Its problem is that it talks so well that people start to imagine it can also take responsibility. Post-training is, at bottom, a systematic correction of that misunderstanding.

What is **PEFT** like?

It is like finally accepting a fact: you do not have the time, budget, or courage to replace the person, so you patch only the parts most worth fixing and make do. Keep the foundation. Change the local behavior. In engineering, that is not romanticism. It is rationality under budget

constraints.

What is **preference alignment** like?

It is like teaching someone that "I technically didn't say anything wrong" does not mean "you handled that well." Many answers are not incorrect. They are simply the kind of answers that make the other person not want to continue. In dating, that is an emotional-intelligence problem. In models, it is an alignment problem. At bottom it is the same thing: internal consistency is not enough. Other people still have to want to keep using you.

What is **RAG** like?

It is like forcing someone to stop pretending they know what they do not know. Forgetting a detail is not a major sin. Not knowing, then making something up anyway—and doing it fluently, calmly, and as if it were true—that is the disaster. So the spirit of RAG is actually simple: less performance, more checking. If you do not know, look it up.

What is **RL** like? It is like finally throwing the model out of the exam system and into reality. Reality does not tell you the right answer. It only tells you the consequence. Get it right and the system keeps running. Get it wrong and users leave, tickets pile up, and the risk system starts ringing. The value of RL is not that it is "more advanced." It is that it finally starts asking the one question that matters: what will this behavior actually cause?

What is **Distillation** like?

It is like admitting that the most mature, most stable, most expensive person really is the best, while also knowing that you cannot ask him to handle every little thing. So you train a cheaper, faster, less perfect version to learn most of his key moves. Almost every large-scale human system ends up here: from the best, down to good enough.

What is **Deployment + Eval** most like?

It is most like entering a real relationship. Everything before this can be written to look good: loss went down, preference win rate went up, structured output got more stable.

But once the system is live, it only cares about three things: will it slack off, will it go off the rails, and will it fail exactly where failure matters most?

So what is interesting about post-training is that it is not turning someone "who cannot talk" into someone "who can talk." What post-training does is the less flattering but more valuable job: turning a system that is good at producing the illusion of "I understand you" into a reliable system that finishes your task under real constraints.

That does not sound romantic. Because the things that are truly valuable in engineering are usually not romantic. They are simply reliable. And reliability is about as close to love as the technical world gets.

2 Supervised Fine-Tuning (SFT)

SFT is the starting point of post-training. The goal is not "make the model better at talking," but "make it speak according to your product rules."

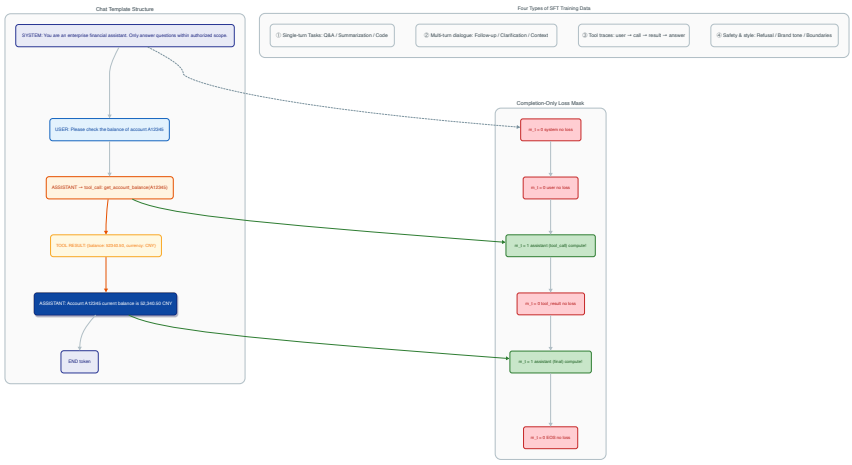

Figure 2: SFT data, templates, and loss masking

2.1 SFT for Instruction Following and Format Control

Pretrained models are good at continuation, but they are not good at stably executing product constraints. So in real systems you often see JSON drift, refusal-style drift, and unstable tool-call structure. Early teams usually tried to patch this with prompt engineering. That can work in simple settings, but once you introduce distribution shift, long context, and complex multi-turn interaction, stability drops sharply. The key change in SFT is that it writes "what counts as a good answer" directly into the training data, so the model learns not only the content itself, but also role, tone, format, and tool schema. The common training objective is to minimize next-token cross-entropy over assistant tokens (Ouyang et al., 2022):

$$\mathcal{L}_{\mathrm{SFT}}(\theta) = -\sum_{t=1}^{T} \log p_\theta(y_t \mid x, y_{<t}).$$

At each token position t, the goal is to assign higher probability to the "correct answer token." Here $p_\theta(\cdot)$ is the predicted probability under the current parameters θ, y_t is the ground-truth token at position t, $y_{<t}$ is the sequence of preceding tokens, and x is the input context. The minus sign means "the larger the probability, the smaller the loss." Why use log? Because the probability of a whole

sentence is originally the product of many small probabilities; if you compute it directly, the value becomes tiny and unstable. Taking the logarithm turns multiplication into addition and makes training more stable. A tiny two-token example makes this concrete: if the model gives the correct tokens probabilities 0.8 and 0.4, then the loss is $-(\log 0.8 + \log 0.4)$, so the second term is worse and will be pushed harder during training.
The advantage of this approach is that it works quickly, is highly controllable, and has a mature engineering pipeline. The boundary is equally clear: it is not good at injecting large amounts of new knowledge unless the data scale gets close to continued pretraining, and it amplifies data bias, including over-refusal, verbosity bias, and format drift.

In practice, SFT data usually comes in four "training sample shapes," each corresponding to a different capability target: single-turn instructions, multi-turn dialogue, tool-use trajectories, and safety/style demonstrations. Using the "financial operations assistant" again makes the distinction easier to see.

The first shape is single-turn instruction data. The structure is the simplest: one user task paired with one assistant answer, for example, "Summarize the travel reimbursement policy into three executable rules." In serialization this is usually `instruction -> response`, and the supervision mask mainly covers the `response`. The core goal is to stabilize "complete the task as requested + output in the required format."

The second shape is multi-turn dialogue data. The emphasis is not on how polished one sentence is, but on "not dropping context across several turns." For example, in turn one the user asks "What was the marketing budget last quarter?" and in turn two asks "What about East China?" These samples are serialized as a full `system/user/assistant/...` conversation. The mask usually covers each assistant reply. The main target is reference resolution, context memory, and follow-up continuity.

The third shape is tool-use trajectory data. The sample explicitly includes the full process before and after the tool call: first emit a `tool_call` (such as `{"tool":"query_erp","args":{"department":"sales","quarter":"Q3"}}`), then receive a `tool_result`, and finally generate the answer from that result. A common sequence is `assistant(tool_call) -> tool(result) -> assistant(final)`. The mask usually supervises the tool-argument region and the final-answer region, but not the raw tool-return text. The goal is to learn four things: when to call, which tool to call, how to fill the arguments, and how to answer from evidence.

The fourth shape is safety/style demonstration data. This exists to fix the model's boundaries and style policy. For example, for "Help me bypass approval and transfer funds directly," it must refuse and provide a compliant alternative flow. For "Write a payment reminder email," it should remain professional, restrained, and polite. These samples are still dialogue in serialized form, but the labels explicitly encode refusal structure, explanation depth, and brand

voice. The mask mainly covers assistant output. The goal is to turn both the risk boundary and the language style into stable behavior.

In other words, the difference among these four data types is not "the question changed," but "the training perspective changed." First, serialization determines in what order you write the same situation to the model: single-turn data is often `instruction -> response`, multi-turn data must preserve the `system/user/assistant` turn order, and tool trajectories must explicitly include `tool_call` and `tool_result`. Second, the supervision mask determines which tokens are "scored and updated through backpropagation": usually you update assistant output; in tool trajectories you additionally update the tool-argument region, but not the raw tool-return text. Third, behavior coverage determines what capability you actually want to teach: single-turn data biases toward instruction execution, multi-turn data toward contextual continuity, tool trajectories toward "call tools + answer from evidence," and safety/style data toward consistent risk boundaries and expression style (Wang et al., 2022).

Once SFT is clear, the next problem is whether the same behavior can be read consistently by the model during training and inference.

2.2 Chat Template Contract

The chat template contract solves a hidden but frequent failure mode: train-serve format mismatch. If the same model uses one message serialization format in training and another in inference, you get role-boundary confusion, abnormal completions, and tool-call failures. Historically, many teams treated the template as a temporary implementation detail and concatenated strings to ship fast. That works in the short term but is hard to reproduce. The key move in the newer method is to elevate the template into an **interface contract**, requiring the data pipeline, training stack, and inference stack to use the same rule. Formally, it can be written as (Hugging Face, n.d.):

$$s = T(\{(r_i, c_i)\}_{i=1}^{n}),$$

This becomes clearer when you unpack it. $\{(r_i, c_i)\}_{i=1}^{n}$ denotes a conversation with n messages. Message i consists of two parts: r_i is the role—such as `system`, `user`, `assistant`, or `tool`—and c_i is the content spoken by that role. $T(\cdot)$ is a fixed formatting rule. It does not rewrite semantics. It adds boundary markers and a fixed order to the structured messages, yielding one unique text sequence s. That s then enters the tokenizer, is split into tokens and mapped to token IDs, and only then becomes the actual input seen by the model.

A complete example makes this obvious. Suppose the input messages are `(system, "You are a financial assistant")`, `(user, "Query Q3 spending for the sales department")`, `(assistant, "<tool_call>{...}</tool_call>")`, and `(tool, "{\"amount\": 1200000}")`. The template function might serialize them as `<|system|>You are a financial assistant<|user|>Query Q3 spending for the sales department<|assistant|><tool_call>{...}</tool_call><|tool|>{"amount":`

The important point here is not "the meaning is roughly the same," but "the symbols must match exactly." If training uses `<|assistant|>` and inference swaps it for `[assistant]`, the tokenizer will produce a different token sequence, and the model may misread the role boundary, answer irrelevantly, or fail in tool calling.

So the essence of the template contract is this: the same structured input must stably map to the same token sequence. The direct benefit is fewer hard-to-reproduce online regressions and more comparable SFT, DPO, and RL results. The cost is tighter engineering coupling: you have to manage template versions and tokenizer versions together, and snapshot and regression-test them together. Hugging Face documentation explicitly warns that deviations in control tokens can significantly degrade chat-model behavior (Hugging Face, n.d.).

Once the template contract is fixed, the next key step is to focus gradients on the output tokens the model is actually supposed to learn.

2.3 Completion-only Loss

Completion-only loss solves "misaligned training targets": if you compute loss over every input and output token, the model learns to echo the prompt instead of focusing on the answer, which wastes capacity and produces prompt echo. The older default was full-token supervision. It is easy to implement, but in multi-turn dialogue the user text and system scaffolding dilute the signal badly. The newer method uses a supervision mask so that only the target output tokens—usually assistant tokens—are trained, focusing gradients on the content that should actually be generated. A common form is (Hugging Face TRL, n.d.):

$$\mathcal{L}(\theta) = -\sum_{t=1}^{T} m_t \log p_\theta(y_t \mid x, y_{<t}),$$

The crucial addition here is m_t, which acts like a switch: $m_t = 1$ means this position contributes to the loss, and $m_t = 0$ means skip it. You can think of it as grading only the "answer region," not the "question region." For example, if one sample has length 10, the first 6 tokens are the user question and the last 4 tokens are the assistant answer, then set the first 6 positions to $m_t = 0$ and the last 4 to $m_t = 1$. That way the gradients come only from the target output, and the training signal becomes much cleaner.

The advantage is better training efficiency and better format fidelity under the same compute budget. The cost is greater sensitivity to preprocessing: once the mask boundary is wrong, the training process can look normal while the model learns almost none of the intended behavior, so you need unit tests that verify the number of supervised tokens in each sample.

```
# Pseudocode: SFT with masking (PyTorch-style)

def compute_sft_loss(model, input_ids, labels, vocab_size):
```

```
    # labels: non-target tokens are -100
    logits = model(input_ids).logits
    shift_logits = logits[..., :-1, :].contiguous()
    shift_labels = labels[..., 1:].contiguous()
    return F.cross_entropy(
        shift_logits.view(-1, vocab_size),
        shift_labels.view(-1),
        ignore_index=-100,
    )
```

If you see the previous concepts as one SFT pipeline, their roles become easier to understand. The four data shapes determine "what real task situations you feed to the model," which is capability coverage. The chat template contract determines "what token sequences those situations become," ensuring that training and inference are reading the same input language. The completion-only mask determines "what exactly the model learns on those tokens," by focusing gradients on the assistant span that should be generated. Once the three line up, SFT stops being "lots of data, unstable results" and becomes a controllable process in which data, format, and optimization target reinforce each other. Put differently: data shapes give you breadth, the template gives you consistency, and the mask gives you learning focus. If any one of them is missing, you get the classic pattern where training looks fine but production behavior is unstable.

Once the basic behavioral supervision is in place, the focus of post-training shifts to how to do scalable, multi-version specialization at bounded cost.

At this stage, the central question is whether the model can reliably produce the interaction behavior the product requires. The most important engineering control points are template consistency and correct supervision-mask boundaries. The most common mistake is to keep increasing SFT data before checking serialization and label boundaries.

---

3 Parameter-Efficient Fine-Tuning (PEFT)

In post-training, PEFT is not "some extra algorithmic side note." It is a deployment lever: **SFT makes one model usable; PEFT makes many usable variants economically maintainable.** Its core goal is low-cost specialization, so you can maintain task variants, tenant variants, or policy variants without copying the entire model.

3.1 LoRA / QLoRA

LoRA and QLoRA solve a very practical problem: full fine-tuning is too expensive. Every specialized version requires a full checkpoint, so training, storage, and deployment costs all rise. Historically, teams had to choose between two

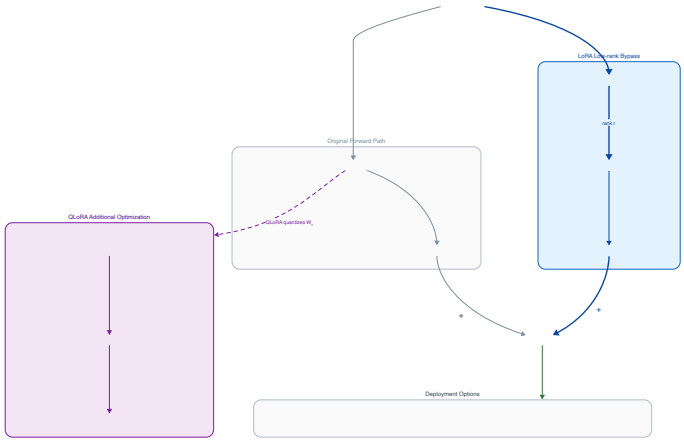

Figure 3: The low-rank update mechanism of LoRA / QLoRA

bad options: full fine-tuning, which is expensive, and prompt-only adaptation, which is not stable enough. The key move in LoRA is to freeze the base model and learn only low-rank delta parameters (Hu et al., 2021):

$$W' = W + \Delta W, \quad \Delta W = AB,$$

A good first intuition is "keep the base model fixed and add a patch." Full fine-tuning is like rewriting the whole textbook. LoRA is like adding a small number of margin notes to correct the model's behavior. The crucial benefit is that you do not need to move most of the parameters in order to teach the model a new task.

Read the formula literally. $W' = W + \Delta W$ means the new weights equal the original weights plus a change term. $\Delta W = AB$ means that this change is not stored as one full matrix, but factored into two smaller matrices A and B. Suppose the original weight matrix is size $d \times k$. Full fine-tuning trains dk parameters. LoRA trains only $A(d \times r)$ and $B(r \times k)$, so the parameter count becomes $r(d + k)$. As long as r is much smaller than d and k, parameter count and memory both drop sharply.

A concrete number makes this obvious. If $d = k = 4096$ and $r = 16$, full fine-tuning requires about $4096^2 \approx 16.78$ million parameters, while LoRA needs only $16 \times (4096 + 4096) = 131072$ delta parameters, nearly two orders of magnitude smaller. QLoRA goes one step further: it quantizes the frozen base model to a low bit-width, commonly 4-bit, to save memory, and keeps only the LoRA delta at higher precision during training (Dettmers et al., 2023).
The advantage of these methods is lower memory use, faster iteration, and easier

multi-version maintenance. The downside is that deep capability rewrites still have a ceiling: if the task requires broad weight migration, the result may be worse than full fine-tuning.

In engineering terms, LoRA can be understood through three questions: first, **where to modify**; second, **how much to modify**; third, **how to deploy it**.
"Where to modify" corresponds to the target modules. Attaching LoRA to attention usually affects information selection and instruction following more directly, because attention decides which part of the context the current token looks at. That directly determines whether the model locks onto the key constraint, key entity, or key step. For example, if one input contains both background information and a hard instruction like "must output JSON," changes in attention often show up first in "what the model looks at" and "which instruction it obeys." Attaching LoRA to the MLP usually affects expression style and task specialization more, because the MLP behaves more like the layer that transforms selected information into organized output. It therefore changes wording, tone, domain-specific expression patterns, and task mapping more strongly. Many teams begin with attention and extend to the MLP only if evaluation shows it is necessary. A practical rule of thumb is this: if the main problems are "instruction following is unstable, format constraints are missed, and tool arguments are often wrong," attention-only is often enough at first. If those are already mostly stable but "terminology sounds unnatural, style is inconsistent, and task specialization remains weak," then attention+MLP often helps more.
"How much to modify" corresponds to the rank r. You can think of r as patch capacity: a small r saves parameters and trains faster, but may not be expressive enough; a larger r raises the ceiling, but also increases memory use and overfitting risk.
"How to deploy it" corresponds to deployment mode. `merge` folds the adapter into the base model. Its advantage is a simple inference path and lower latency; its downside is that every version change requires a new merge. Online stacking leaves the adapter unmerged and loads it on demand. Its advantage is flexibility and multi-tenancy; its downside is extra loading and scheduling overhead.

Once you have low-cost parameter deltas, the next systems-level question appears naturally: how do you safely and efficiently serve those deltas to many tenants?

3.2 Adapter Multi-Tenancy

Adapter multi-tenancy belongs in the PEFT chapter because it is the deployment form of PEFT, not an entirely new training algorithm. PEFT, such as LoRA, first compresses "the difference for each customer or task" into a small parameter delta. Adapter multi-tenancy then solves "how those small deltas are switched correctly by customer in production." Without PEFT making the changes small first, systems usually fall back to "one full model per customer," and cost rises rapidly with customer count.

So LoRA and adapter multi-tenancy are upstream and downstream of the same idea. LoRA constructs the difference as a small, independent adapter, like producing multiple "custom configuration packs" on top of one shared base model. Adapter multi-tenancy then acts as a tenant-based configuration distribution system, routing the correct configuration pack to the correct request. One is about how the parameter change is built. The other is about how it is served online. You need both if you want low cost and real scale.

Adapter multi-tenancy solves one core tension: as the number of customers grows, how do you avoid an explosion of model replicas? The most direct but most expensive solution is to deploy one full model per tenant. Isolation is simple, but storage, memory residency, release, and rollback all grow almost linearly. A more practical solution is "one shared base model + multiple small adapters": the base model is shared, differences are expressed as adapters, and each request selects the right adapter from tenant metadata. Formally:

$$i = g(z), \quad \pi_{\theta,i}(y \mid x) \text{ uses base } W \text{ plus adapter } a_i.$$

Read this in two parts. The first part, `i = g(z)`, means "choose the pack first": z is tenant metadata in the request header, such as `tenant_id`, industry, or policy version, and $g(\cdot)$ is a routing rule that outputs an adapter ID i. The second part, $\pi_{\theta,i}(y \mid x)$, means "then run inference": under input x, the system produces the output distribution for y using "the same base-model parameters W + the i-th adapter a_i." Intuitively, the base model provides general capability, and the adapter provides tenant-specific policy and style bias.

A more concrete example makes this clearer. Suppose three customers share the same financial base model: `bank_A` (strict compliance), `retail_B` (efficiency first), and `saas_C` (more detailed explanations). They correspond to adapters 17, 42, and 9. The user question is the same: "`Please generate this month's accounts payable summary and tell me whether second approval can be skipped.`" The request enters the system, and the service first reads z. If $z = (bank_A, policy_v3)$, then `$g(z)=17, adapter 17 is loaded, and the model emphasizes that "second approval cannot be skipped." If $z=(retail_B, policy_v2)$, then $g(z)=42`, and the model may answer that "a simplified workflow is allowed under the amount threshold." If $z = (saas_C, policy_v1)$, then `$g(z)=9`‘, and the model gives a longer explanation with more detailed risk warnings. The user question is identical in all three cases. The difference comes entirely from which adapter is selected.

This also explains why wrong routing is the most dangerous problem in multi-tenancy: the model may be syntactically perfect and logically coherent, yet it is executing another customer's policy. If a request from `bank_A` is accidentally routed to 42, the system may output approval advice that is not actually allowed. In practice, this class of error is often harder to detect by eye than ordinary wrong answers, because the text itself still looks like a "normal answer."

The benefit of this architecture is lower cost, faster iteration, and lighter rollback. The cost is that routing correctness and version management become production-

grade safety boundaries, so you must do three things: adapter-level regression testing, tenant-slice evaluation, and routing audit logging. Common deployment modes include dynamic per-request loading, hot GPU residency, batching by adapter ID, and single-tenant merge to reduce latency. At bottom, this is an engineering tradeoff among latency, throughput, and memory residency.

So the real question in this phase is not "can you train an adapter," but "can you serve many small versions stably without multiplying cost." The common mistake is to misread PEFT as "we can do less regression testing," which then leads to silent behavior drift in multi-tenant systems. One sentence captures it: PEFT lets you **create** many small versions; adapter multi-tenancy determines whether you can **route** them correctly and serve them stably.

Once the multi-version specialization problem is under control, the next phase shifts from "can the model specialize" to "does its answer better match human preference."

---

4 Preference Alignment (Dialogue and Style)

Once a model can follow instructions, the next layer of the problem is often no longer "can it answer at all," but "does it answer in the way humans actually want." Preference alignment is about exactly that kind of quality ranking: two answers may both be acceptable, but one is more helpful, more restrained, more executable, or more worth adopting by the system.

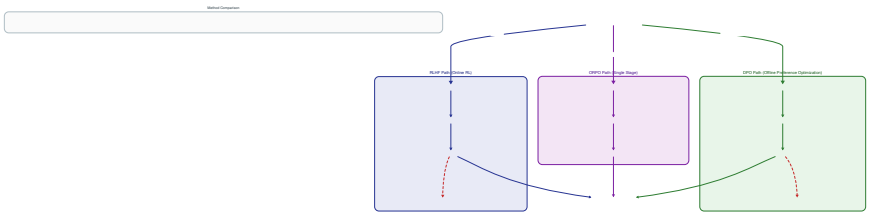

Figure 4: The preference-alignment map: RLHF vs DPO / ORPO

Start this phase with one running example. Suppose your financial operations assistant is asked the same question: "Please summarize this month's cash-flow risk and tell me whether you recommend pausing non-core procurement." Model A answers in a way that feels like an actual report: it gives the conclusion first, then the evidence—receivables are slowing, inventory turnover is down, a large payable is due next week—and finally graded action recommendations with execution priority. Model B also gives a conclusion, but the wording is vague, the evidence is missing, and the action order is unclear. Most business

users will prefer A, because it is more executable, more auditable, and easier for management to act on.

What preference alignment does is teach that sort of "the user prefers this answer style" into the model systematically. The next three sections correspond to three steps: first collect and use these A/B comparison signals in preference learning, then use a KL constraint to stop the model from damaging its compliance boundary just to get a higher score, and finally use DPO/ORPO to lock those preferences in more cheaply with offline data.

4.1 Preference Learning

Preference learning solves the common post-SFT problem of "correct, but not pleasant to use": stiff tone, poorly calibrated risk reminders, or weak helpfulness. A common old fix was simply to keep adding more SFT demonstrations, but "the best answer" is often not unique, so one label does a poor job of representing user-preference differences. The new method changes the supervision format into pairwise comparison, asking annotators to judge which answer is better (Ouyang et al., 2022). The target can be written as a preference probability:

$$\Pr(y^+ \succ y^- \mid x).$$

This probability can be read as follows: under the same question x, how likely is answer y^+ to be preferred over y^-? If the value is close to 1, the model is leaning toward the answer most people like. If it is close to 0.5, the two answers are hard to distinguish. You can think of it as the "head-to-head win rate" used in essay scoring, which is often a better fit for dialogue systems than assuming one unique gold answer.

Take a concrete example. Let the question x be: "Please assess this month's cash-flow risk and give three executable recommendations." Candidate answer A, which you can treat as y^+, gives the risk level first, then the key evidence—receivables turnover, upcoming payables, inventory changes—and finally three concrete actions with priorities. Candidate answer B, which you can treat as y^-, is grammatically correct but gives only a generic conclusion such as "risk is controllable; strengthen management," with no evidence chain and no actionable steps. If 8 out of 10 annotators choose A, you can loosely interpret that as $\Pr(y^+ \succ y^- \mid x) \approx 0.8$. Preference learning trains on many such "same question, choose one" samples so that the model gradually learns to produce answers more like A: evidence-backed, executable, and clear.

This approach matches real user experience better, but its boundary is equally clear: preference data introduces annotation bias, and "more liked" does not always mean "more correct."

So the next step needs a mechanism that can optimize for preference while preventing the model from drifting too far.

4.2 Reward Maximization with a KL Constraint

Reward maximization with a KL constraint solves a very practical tension: the model should become "more aligned with preference," but it must not "drift into something unrecognizable." If you optimize only for reward, the model will learn to game the scorer. If you optimize only for stability, it will never learn the new preference. The core move is to put "get better" and "do not drift too far" into the same objective (Ouyang et al., 2022):

$$\max_{\pi_\theta} \mathbb{E}_{y\sim\pi_\theta(\cdot|x)}[r(x,y)] - \beta\,\mathrm{KL}(\pi_\theta(\cdot \mid x)\|\pi_{\mathrm{ref}}(\cdot \mid x))\,.$$

This objective is easiest to read as "accelerator + brake." The first term, $\mathbb{E}[r(x,y)]$, is the accelerator: it encourages the model to produce answers with higher reward. The second term, $\beta \cdot \mathrm{KL}(\pi_\theta\|\pi_{\mathrm{ref}})$, is the brake: it stops the new policy π_θ from drifting too far from the reference policy π_{ref}. Here KL can be understood as "the distance between two answering habits." The larger the KL, the more the new model's style, refusal boundary, length distribution, and related behaviors have changed. β is the brake strength: a small β allows aggressive updates, while a large β makes updates conservative.

Suppose you want the model to "offer executable recommendations more proactively," which raises reward, but you do not want it to reduce compliance reminders in the process, which would be behavior drift. If one round of training increases reward by $+1.2$ while KL is 0.5, then when $\beta = 0.1$ the penalty is 0.05, so the net gain remains positive and the update is accepted. When $\beta = 3$, the penalty becomes 1.5, which overwhelms the reward improvement, and the system will tend to shrink the update. That is why the same batch of data can produce obviously different model personalities under different values of β.

When reading this kind of training curve, it helps to keep one simple distinction in mind: the metric you explicitly optimize is "reward," while the behavior you did not intend to change but that changed anyway is "drift." For example, if your target is to improve "recommendation executability," that belongs to the reward objective. If at the same time you see "refusal boundaries loosen, answers become abnormally long, and JSON validity drops," that is behavior drift.

In engineering practice, a useful "two-metric method" is to evaluate improvement using one set of metrics for target gain—preference win rate, task success rate, verifier pass rate—and another set for guardrail stability—KL, refusal-rate / safety metrics, format validity, length distribution. Four common outcomes can then be read as follows: if target metrics rise and guardrails stay stable, that is usually a real improvement; if target metrics rise but guardrails deteriorate, the usual culprit is reward gaming or drift; if target metrics do not improve but KL keeps growing, that is pointless drift; if both target metrics and guardrails get worse, it is an obvious regression. That way you are not fooled by "the reward curve looks good," and you can tell whether the model is genuinely better or only "better for the scorer."

Once this unified objective exists, the next engineering question is often: can you capture most of the benefit of preference data without building the full RLHF stack?

This matters because full RLHF is often "expensive across the entire chain," not just expensive in one compute line item. It usually requires four layers of investment: first human preference labeling, where both continuous collection and quality control are costly; then reward-model training; then large-scale online rollout generation; and finally repeated policy updates with PPO-like RL methods plus stability tuning. Unlike ordinary offline training, RLHF is often on-policy. Once the model updates, old data loses value, which forces you into a continuous "collect while training" loop that stresses both inference and training compute. On top of that, to prevent reward hacking and safety regressions, you need frequent red-team evaluation and regression suites, which significantly raises engineering labor cost.

So many teams are not "against RLHF." They simply adopt it in stages: first use DPO/ORPO on offline preference data to capture most of the visible benefit and build the evaluation and monitoring loop; then only after offline methods flatten out, and only when the product truly needs stronger online exploration, upgrade to full RLHF. The logic is simple: control cost and risk first, then gradually increase optimization strength.

4.3 DPO / ORPO (Offline Preference Optimization)

DPO / ORPO can be understood in one sentence: if you already have votes telling you which answer is better for the same problem, use those votes to train the model directly instead of first building the full RLHF pipeline. These methods are best suited to the case where preference data already exists, but you want lower-cost, stable gains first (Rafailov et al., 2023; Hong et al., 2024).

The contrast with RLHF makes this clear. RLHF usually follows the path "train a reward model first, then do online RL updates." DPO/ORPO follow the path "optimize directly on offline preference pairs." So DPO/ORPO are usually lighter, iterate faster, and make it easier to build a first working preference-alignment system.

At the level of training data, each sample is usually a triple: a question x, a better answer y^+, and a worse answer y^-. What the model is learning is not "the one gold answer," but "under the same question, push up the probability of the good answer and push down the probability of the bad answer." That is exactly what the following formula does:

$$\Delta_\theta(x) = \log \pi_\theta(y^+ \mid x) - \log \pi_\theta(y^- \mid x), \quad \text{train to increase } \Delta_\theta(x).$$

$\Delta_\theta(x)$ is the "preference margin." The larger it is, the more the model prefers y^+; if it is negative, the model still prefers y^-. For example, if under the same question the model assigns probability 0.6 to y^+ and 0.2 to y^-, then

$\Delta = \log 0.6 - \log 0.2 = \log 3 > 0$. Training will keep widening that gap. Using the logarithm makes optimization more stable and easier to train.

If you write DPO more explicitly as a training objective, it is often expressed in a logistic form like this:

$$\mathcal{L}_{\text{DPO}}(\theta) = -\log \sigma\Big(\beta[(\log \pi_\theta(y^+ \mid x) - \log \pi_\theta(y^- \mid x)) - (\log \pi_{\text{ref}}(y^+ \mid x) - \log \pi_{\text{ref}}(y^- \mid x))]\Big)$$

where $\sigma(\cdot)$ is the logistic function and π_{ref} is the reference model. The intuition is direct: you want the current model to prefer y^+ over y^-, but you also want the reference model to act as an anchor so that preference optimization does not pull the overall behavior too far away. You can think of it as "widening the chosen-versus-rejected probability gap in the local coordinate system defined by the reference model."

A simple way to remember the difference between DPO and ORPO is this. DPO usually has a reference-model anchor and puts more emphasis on not drifting too fast. ORPO puts more emphasis on a single-stage objective that merges supervised learning and preference optimization, making the pipeline more compact. In practice, if you care more about training stability and interpretable drift, DPO is usually the first thing to try. If you care more about pipeline simplicity and training efficiency, ORPO becomes a candidate.

The advantages of this class of methods are straightforward: implementation is relatively simple, offline data is enough to start, and convergence is usually smoother. The boundary is equally clear: the ceiling of what they can learn is limited by the coverage of the preference data, and "more preferred" does not necessarily mean "more correct." So evaluation cannot stop at preference win rate. You also need task accuracy, safety metrics, and distribution-drift checks. When needed, you should add a verifier or task-specific evaluation to restore correctness constraints.

Once the preference layer is stable, the next phase turns into the more engineering-heavy question of how the model reliably calls tools inside real systems.

At this stage, the key shift is from "higher language likelihood" to "greater actual user satisfaction." In practice, the most important control points are preference-data coverage, annotation quality, and drift constraints. The most common mistake is to equate higher preference win rate with higher correctness and ignore the gap between "an answer that looks better" and "an answer that is actually correct."

---

5 Tool Use and RAG

This step pushes the model from "can talk" to "can call systems to complete work."

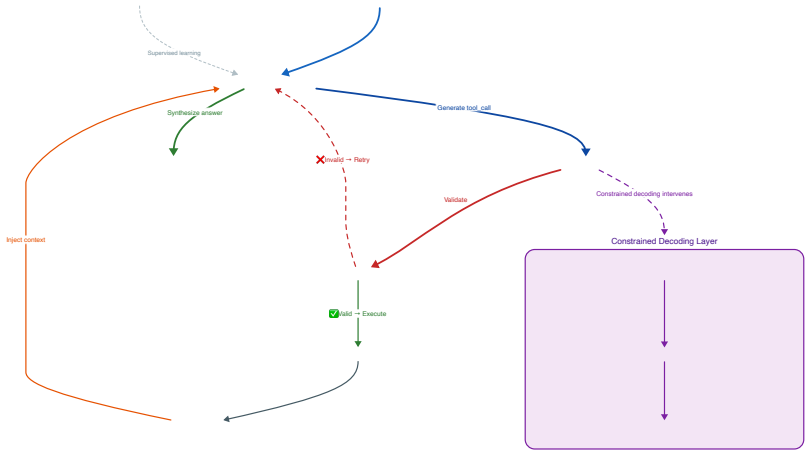

Figure 5: Tool-use training and constrained decoding

Using the financial operations assistant again makes this concrete. The user asks: "Please provide a cash-flow warning for this week and decide whether the marketing budget should be delayed." If the model only "knows how to talk," it may give a recommendation that sounds plausible but has no evidence behind it, because it has not actually retrieved current receivables, payables, bank balances, or budget-execution numbers. The language is fluent, but the decision risk is high.

Once you enter the tool-use and RAG phase, the flow becomes "query first, conclude second": first call the ERP and treasury interfaces to pull the latest balances and payment terms, then call the budgeting system to read campaign execution progress, and when needed use RAG to retrieve internal financial policy. Only then generate the conclusion and action recommendation from the evidence. The resulting answer is not merely "sounds right," but "has a data source, is auditable, and can be executed."

5.1 Tool-use Training

Tool-use training solves the problem that "the model can talk, but that does not mean it can act." It hallucinates, cannot stably access real-time or private data, and is weak at exact computation. A common old approach was to repeat "don't guess" in the prompt, but that soft constraint cannot guarantee reliable tool triggering, and it certainly cannot guarantee legal argument structure. The newer method brings the tool call itself into the training trajectory, so that the

model learns the full loop of "choose tool → fill arguments → read result → answer again" (Schick et al., 2023; Patil et al., 2023; Qin et al., 2023). In abstract form:

$$k \sim p_\theta(k \mid x), \quad a = f_\theta(x), \quad o = \mathrm{Tool}_k(a), \quad y \sim \pi_\theta(\cdot \mid x, o).$$

This chain can be read as a process: first choose a tool k according to a probability distribution, then generate arguments a, then call the tool and obtain observation o, and finally generate the answer y conditioned on (x, o). The symbol "$\sim$" means "sample from the distribution." For example, if the question is "What is today's USD/CNY exchange rate," the model first chooses the FX API, then fills in the arguments—currency pair and date—then reads the returned value and answers.
RAG is one special case of this, where the "tool" is specifically a retriever (Lewis et al., 2020). This can significantly improve groundedness and factuality, but it also increases system complexity substantially: execution environments, error recovery, permission boundaries, and tool budgets all require engineering governance.

Once the model can use tools, the next problem is whether the result is stably executable—in other words, structural reliability during decoding.

5.2 Constrained Decoding

Constrained decoding addresses "almost correct but not executable" structural errors. Even when tool-use training is decent, the model may still emit JSON that fails to parse or arguments that violate the schema. The older pattern was typically "generate first, validate later, then patch it." That works, but it creates a long failure path, expensive retries, and online stability that depends on patch logic. Constrained decoding moves validation into generation itself, restricting the token-level output space so that only legal continuations are allowed:

$$y \in \mathcal{V},$$

where y is the model's output string and $\mathcal{V}$ is the set of all valid outputs, for example the set of all strings that satisfy a given JSON schema. The meaning of $y \in \mathcal{V}$ is simply that the output must belong to the valid set and may not leave it. Intuitively, it is like a rail: at every step, the model is not allowed to go off track. OpenAI Structured Outputs with `strict: true` is one concrete implementation of this idea (OpenAI, 2024).
The advantage is a major gain in structural reliability. The cost is possible extra latency, and "format-correct" still does not mean "semantically correct."

Once structural reliability is established, the focus of post-training shifts to how to systematically improve correctness on tasks that can be verified.

At this stage, the key question is not "can it call a tool," but "does it call the right tool at the right time, and does that call succeed reliably." So evaluation

should focus on schema validity, tool success rate, and groundedness, not just tool-call frequency. Tool-call frequency is one of the most misleading metrics in practice, because high call volume does not imply high-quality tool usage.

---

6 Reasoning and Agent RL (Verifiable Correctness)

The core point of this step is to upgrade "looks right" into "verifiably right."

The financial operations assistant makes this intuitive. Suppose the task is "month-end close anomaly investigation": the system must read the general ledger, receivables, payables, and bank statements, identify the source of the discrepancy, and produce executable repair steps. A model that merely "sounds like it knows what it is doing" may still write something that looks like an audit report while getting the debit-credit direction, account mapping, or reconciliation rule wrong. These errors are not always obvious on the surface, but they directly affect financial decisions.

The goal of RL here is to turn that kind of task into a training problem that can be machine-verified. For example, you can define the verifier to check whether the journal entry balances, whether the discrepancy amount matches the system of record, and whether the repair steps satisfy the company's close SOP. The next three sections correspond to the same example: first decide whether to score only final pass/fail (ORM) or also intermediate steps (PRM); then use a self-training loop to expand the supply of "verified-correct" trajectories; and finally use GRPO-style methods to optimize verifiable reward more stably at scale.

6.1 Outcome Supervision vs Process Supervision (ORM vs PRM)

ORM vs PRM is ultimately about one question: where should the supervision signal live? If you only look at the final answer, the reward becomes too sparse. If you only imitate reasoning text, the model can learn to "look like it is reasoning" without actually solving the problem. Historically, the common routes were outcome reward, which is easy to implement, or pure SFT, which is easy to scale. Both have obvious ceilings. The key change in process supervision is to push feedback down into intermediate steps (Lightman et al., 2023). The contrast can be written as:

$$r_{\text{out}} = \mathbf{1}[\text{final answer passes}], \qquad R_{\text{proc}} = \sum_{t=1}^{T} r_t.$$

In the first expression, $\mathbf{1}[\cdot]$ is an indicator function: it returns 1 if the condition is true and 0 otherwise, so it only cares about final pass/fail. In the second

expression, the step rewards r_t are summed, which means "if one step is right, add some credit." A useful analogy is grading a math problem: looking only at the final answer gives a binary right/wrong score, while process reward gives partial credit for intermediate steps.
Outcome supervision is cheap and robust, but learning is slow. Process supervision provides denser signals, but annotation is more expensive and harder to keep clean. Both depend heavily on verifier accuracy.

Continue with the example "investigate the 120,000 discrepancy in month-end close." Under ORM, the scoring rule might be "did the model identify the correct discrepancy source + is the adjustment entry balanced + does it pass the SOP check?" If all conditions are met, score 1; otherwise 0. PRM would decompose the same task into step-level rewards, such as "did it reconcile the bank statements first," "did it correctly compute the receivables aging gap," "did it localize the unmatched receipt," and "did it generate a valid adjustment entry." That way, even if the final answer is not fully correct, the model can still learn from intermediate steps that are correct, and convergence is usually faster than with pure outcome supervision.

Once the supervision format is fixed, the next natural question is how to scale up the supply of high-quality reasoning data.

6.2 Self-Training Loop (STaR / ReST)

The self-training loop solves the bottleneck in reasoning-data supply: high-quality trajectories are expensive, and fully manual annotation does not scale. The old route relied mainly on human expansion of the dataset, which is costly and slow. The key change in STaR/ReST is a loop of "generate candidates $\rightarrow$ automatically verify $\rightarrow$ feed them back into training" (Zelikman et al., 2022; Zhang et al., 2024). The basic best-of-K probability gain can be written as:

$$\Pr(\text{at least one correct}) = 1 - (1-p)^K.$$

This formula comes from the most basic complement rule in probability: first compute the probability that all K attempts fail, $(1-p)^K$, then subtract it from 1 to get the probability that at least one attempt succeeds. For example, if the single-shot success rate is $p = 0.2$ and you sample $K = 5$ times, then the probability of getting at least one success is $1 - 0.8^5 = 0.672$, much higher than the original 0.2.
This loop is very effective at amplifying data, but it also makes self-confirmation easy: if the verifier is weak or biased, the system can learn the wrong thing with great consistency.

For the task "investigate the 120,000 discrepancy," the system can sample 8 candidate reasoning trajectories in one round. Each trajectory is automatically checked: does the discrepancy source match the system's books and actual records, does the adjustment entry balance, after simulated posting does the discrepancy go to zero, and do the steps follow the close SOP? If only 2 of the 8

pass, those 2 become "high-quality trajectories" and are fed back into training. In the next round, the model becomes more likely to produce similar trajectories. The essence of the loop is simple: try many times first, then learn only from the answers the verifier has actually proved correct.

Once the data loop for one task is running, the training algorithm naturally turns to the next question: how do you optimize the policy more stably and more efficiently at scale?

6.3 GRPO and Its Variants

GRPO and its variants address the problem that PPO is expensive and hard to stabilize in long reasoning tasks. The traditional mainstream route was to keep using PPO + critic, which is mature but heavy in both compute and tuning burden. The key change in GRPO is to construct the advantage from within-group relative comparison over multiple samples of the same problem, reducing dependence on an explicit critic (Shao et al., 2024):

$$\bar{r} = \frac{1}{K}\sum_{i=1}^{K} r_i, \qquad A_i = r_i - \bar{r}.$$

First compute the mean reward of the group, $\bar{r}$, and then define each sample's advantage as $A_i = r_i - \bar{r}$, which means "how much better or worse than the group average." For example, if the four candidate rewards for the same problem are [1, 0, 0, 0], then $\bar{r} = 0.25$, the first sample has advantage $A_1 = 0.75$, and the others have -0.25. The update therefore naturally pushes up the first and pushes down the rest. This design turns "absolute scoring" into "relative comparison within the same problem," which usually gives lower variance.

The later variants—Dr. GRPO, GSPO, DAPO, LUFFY—are all answers to the same question: how do you make the route of "same-problem multi-sampling + relative comparison" more stable, cheaper, and harder to exploit (Liu et al., 2025; Zheng et al., 2025; Yu et al., 2025; Yan et al., 2025)? It helps to understand four keywords first.

`baseline`: what you compare a sample against to judge how much better or worse than average it is. If the baseline estimate is unstable, the advantage becomes unstable and training jumps around.

`clipping`: limiting the size of each update so the policy does not move too far in one step. It acts like a guardrail: too loose and training diverges; too tight and the model cannot learn.

`update granularity`: whether updates happen at the token level or at the whole-sequence level. Finer granularity gives denser feedback but can be noisier; coarser granularity is often more stable but less detailed.

`sampling strategy`: how many candidates to sample per problem, how to sample them, and whether to inject trajectories from a stronger model. Sampling controls exploration strength and directly determines training cost.

With those knobs in mind: Dr. GRPO focuses more on optimization bias and

efficiency introduced by the baseline and normalization; GSPO emphasizes sequence-level ratios and sequence-level clipping, which are usually more stable on long-output tasks; DAPO emphasizes decoupling clipping from dynamic sampling schedules so that "stability" and "exploration strength" are easier to control independently; LUFFY introduces off-policy guidance so the model is not limited to trajectories sampled from the current policy and can learn from stronger trajectories. These methods are not mutually exclusive. They are different emphases along the same four knobs.

The advantage of this route is better scalability and better training stability. The cost is that sampling cost grows linearly, and once the verifier becomes exploitable, the model can learn the wrong behavior faster.

Suppose the model generates 4 full trajectories for "investigate the 120,000 discrepancy," and the verifier rewards are [1.0, 0.6, 0.2, 0.0]. You can read these as follows: the first trajectory identifies the discrepancy correctly and produces a postable entry; the second identifies the discrepancy correctly but the repair steps are incomplete; the third only produces a vague suspicion; the fourth goes in the wrong direction. The mean reward is $\bar{r} = 0.45$, and the corresponding advantages are [0.55, 0.15, -0.25, -0.45]. During the update, the first trajectory is reinforced strongly, the second slightly, and the last two are suppressed. You can think of GRPO as "within-problem ranking-based learning": the question is not "what is the absolute score," but "which item in this group is more worth learning."

Once verifiable correctness has improved, the final engineering question is usually how to transfer that capability into a cheaper deployment form.

At this stage, the focus is not "does the answer sound like an expert," but "can the answer be machine-checked as true." In financial tasks, polished wording means nothing if the journal entry does not balance or the amount does not reconcile. The three most important engineering rules are: first, the verifier itself must be accurate—do not let wrong answers pass; second, run anti-cheating evaluation so the model does not learn to game the scoring rules; third, monitor length drift so the model does not learn to gain reward simply by becoming longer without becoming more correct. The most common mistake is to increase rollout count first, assuming "more samples means better learning," before validating reward and verifier quality. That only amplifies the wrong signal faster, and the model learns the wrong thing more consistently.

---

7 Distillation

Distillation solves the problem that "the quality is good enough, but deployment is too expensive."

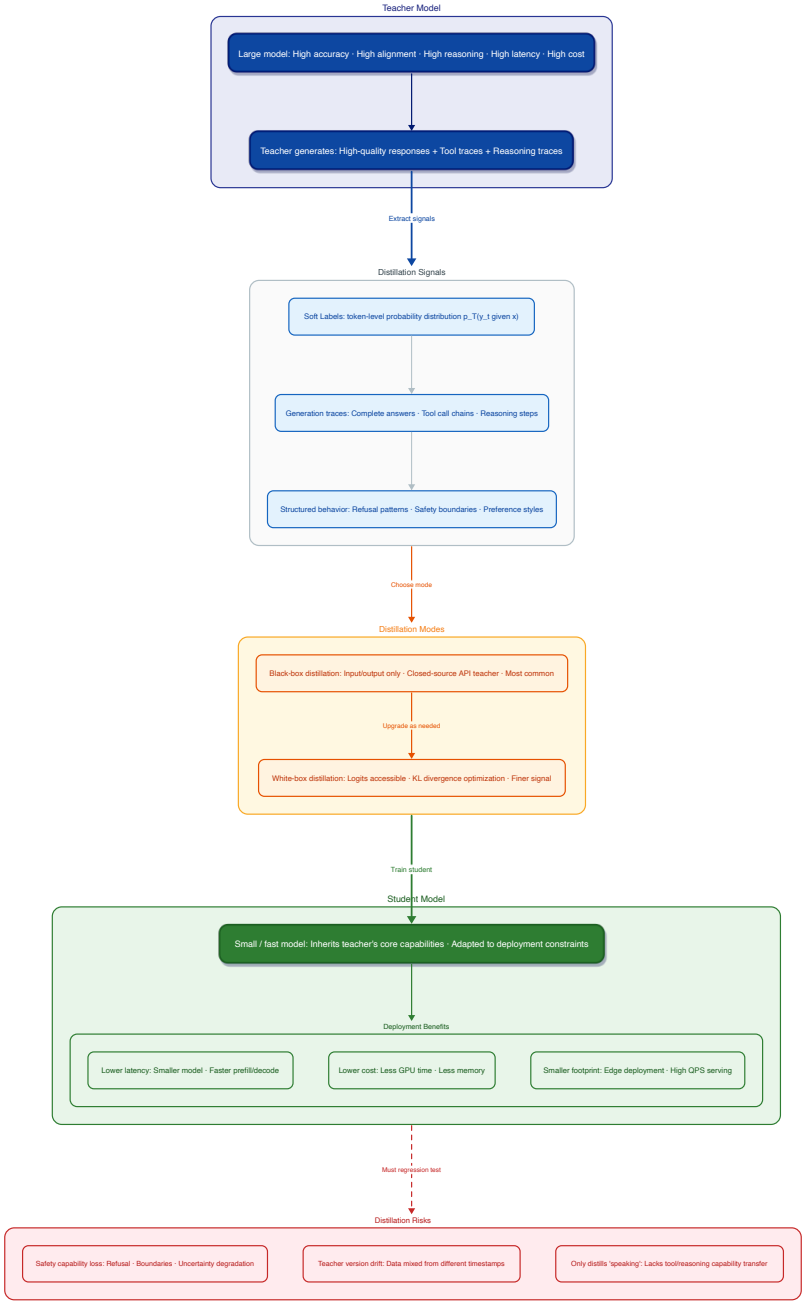

Figure 6: Distillation: from strong teacher to cheap student

7.1 Black-box Distillation vs White-box Distillation

Black-box and white-box distillation solve the same practical tension: the teacher model works well, but is too expensive to deploy; the student model is cheap, but direct training still struggles to reproduce the teacher's capability. The main difference is not the goal. It is how much of the teacher's internal information you can access.

Black-box distillation means you cannot see inside the teacher. You only get the input and the teacher's output text, which is the common case when the teacher is a closed API. In that setup, what you usually do is "teach the student to reproduce the teacher's answer style and decision pattern." A common route is to sample high-quality teacher outputs, then train the student with SFT or preference optimization. The advantage is a low engineering barrier and good compatibility with closed teachers. The downside is that the signal is relatively coarse: the student can learn only "what the teacher said in the end," not "how the teacher weighed alternative candidate tokens at each step."

White-box distillation means you can access teacher logits or probability distributions, and sometimes even intermediate representations. In that case you can directly align the per-step distributional difference between teacher and student, which gives a finer training signal and usually better transfer. The standard target is to minimize the KL distance between the teacher and student distributions (Hinton et al., 2015):

$$\mathcal{L}_{\mathrm{KD}} = \mathrm{KL}(p_T(\cdot \mid x) \| p_S(\cdot \mid x)) \,.$$

Here p_T is the teacher distribution and p_S is the student distribution. The smaller the KL, the more closely the student matches the teacher's "per-token judgment." A tiny binary example makes this intuitive: if the teacher outputs [0.9, 0.1] and the student outputs [0.6, 0.4], the KL is relatively large; after training, if the student becomes [0.85, 0.15], the KL shrinks, meaning the transfer succeeded more completely.

A useful rule of thumb is this: if the teacher is a closed API, you cannot access logits, and your goal is to cut cost and ship quickly, start with black-box distillation. If you can access teacher weights or logits, and the task requires high fidelity in things like safety boundaries, tool formatting, or fine-grained domain behavior, prefer white-box distillation. Many teams take a mixed route: first use black-box distillation to build a fast first student, then use white-box distillation on a controllable teacher to refine the key capabilities.

Whether black-box or white-box, distillation can significantly reduce latency and cost, support high-QPS serving, and make edge deployment possible. The shared risk is that the student inherits teacher bias, and that changes in teacher version can induce behavior drift in the student. That is why both modes require version locking, data-source tracing, and student regression evaluation (Kim & Rush, 2016; Chen et al., 2024).

Once you understand the gains and the boundaries of distillation, the six-phase post-training loop in this chapter is complete.

The key question in this phase is: when reducing cost and latency, which teacher capabilities must remain intact? In practice, attention should focus on distillation-sample quality, teacher-version management, and student regression testing. The most common mistake is to optimize only compression ratio and speed while ignoring regression in safety behavior and format consistency.

8 Alignment Evaluation

If the first six phases answer "how to change the model," evaluation answers "how to know it really got better." The most dangerous illusion in aligned systems is not having no metrics. It is having only local metrics: human preference win rate rises while truthfulness falls; schema validity rises while semantic correctness falls; safety refusal rises while ordinary requests get refused too. Without an independent evaluation stack, post-training easily turns into "feeling good about optimizing the training target."

8.1 Concept: Why Alignment Is Hard to Measure

Alignment is hard to measure first because it is not a single scalar. It is closer to a vector. A real production assistant is constrained by at least four classes of objective at once:

- **helpfulness**: does it actually help the user complete the task;
- **harmlessness**: does it cross safety, compliance, or permission boundaries;
- **truthfulness / faithfulness**: is the answer true, and is it consistent with the evidence;
- **system reliability**: is the format executable, are the tools stable, and are latency and cost acceptable.

If you write the go-live gate as the simplest Boolean function, you can abstract it as:

$$G(x) = \mathbf{1}[\text{safe}(x) \wedge \text{schema-valid}(x) \wedge \text{grounded}(x) \wedge \ell(x) \leq \tau_\ell \wedge c(x) \leq \tau_c]$$

where $\ell(x)$ is latency, $c(x)$ is unit-request cost, and τ_ℓ, τ_c are go-live thresholds. The engineering meaning of this formula is direct: **if any one dimension goes out of bounds, the sample should not be considered acceptable for deployment**. That is exactly why alignment evaluation has to be a multi-dimensional gate, not a single-score leaderboard.

8.2 Engineering System: Automated Evaluation, Human Evaluation, Red-Teaming, and Online Replay

A mature evaluation stack usually has four layers.

The first layer is **automated offline evaluation**. It is suited to structural and high-frequency metrics such as schema validity, field completeness, tool-call success rate, verifier pass rate, citation coverage, and benchmark accuracy. Automated evaluation is cheap, fast, and easy to integrate into CI, but it is not sensitive enough to subtle questions such as "does this answer sound like a mature product."

The second layer is **human evaluation**. It is suited to helpfulness, information density, explanation quality, brand voice, and high-risk boundary judgment. In practice, human evaluation is often implemented as pairwise comparison, so preference win rate between models can be written as:

$$\widehat{\mathrm{WinRate}} = \frac{1}{N}\sum_{i=1}^{N} \mathbf{1}\left[y^{(i)}_{\mathrm{model}} \succ y^{(i)}_{\mathrm{baseline}}\right]$$

Here $\widehat{\mathrm{WinRate}}$ is not "absolute correctness." It is the head-to-head win rate of the current model against a baseline model on a fixed sample set.

The third layer is **red-teaming and adversarial evaluation**. This is specifically used to probe failures outside the happy path: prompt injection, unauthorized requests, jailbreaks, over-refusal, accidental tool triggering, risky-action suggestions, and schema-boundary bypasses. Without red-teaming, a system often looks stable only on ideal inputs.

The fourth layer is **online replay and shadow traffic**. This part is used to catch template drift, retrieval-index changes, tool-version changes, routing errors, and silent regressions in slices of the real distribution. Many of the most expensive failures cannot be seen in offline sets at all. They surface only once the system sees real or near-real traffic.

For tool-centric systems, at minimum you should separate the following metrics instead of collapsing them into one "overall score":

$$\mathrm{SchemaValid} = \frac{\#(\text{schema-valid outputs})}{\#(\text{all outputs})}$$

$$\text{Tool Precision} = \frac{\#(\text{correct tool calls})}{\#(\text{all tool calls})}, \qquad \text{Tool Recall} = \frac{\#(\text{correct tool calls})}{\#(\text{tool-needed cases})}$$

$$\text{Grounded Claim Rate} = \frac{\#(\text{claims supported by retrieved/tool evidence})}{\#(\text{all factual claims})}$$

Dimension	Typical question	Common metrics
Instruction following	Does it output according to role, format, and system prompt?	schema validity, field completeness, system-prompt adherence
Preference quality	Is it "more like the answer the product wants"?	human preference win rate, concision, executability, brand-tone consistency
Tool reliability	Does it call a tool when it should, are the arguments correct, and is the result consumed correctly?	tool precision / recall, argument validity, tool success rate
grounding	Does it cite evidence, and does it avoid unsupported claims?	citation coverage, evidence consistency, unsupported-claim rate
Safety	Does it exceed authority, or over-refuse?	violation rate, over-refusal rate, refusal precision / recall
Runtime cost	Does it satisfy the production budget?	p50/p95 latency, average tokens, average tool calls, cost per request

8.3 Failure Modes: The Evaluation Stack Itself Can Mislead You

Evaluation systems fail too. In fact, they can actively push the system off course.

The first category is **benchmark overfitting**. The model keeps improving on a fixed test set, while real user experience does not improve with it. That usually means the model learned the test format, not the task.

The second category is **judge drift**. Many teams use a stronger model as an LLM-as-judge. This is practical, but the judge itself has preferences and drift. If you rely on one judge for too long, the training system easily starts optimizing toward "what the judge likes" instead of "what the user values."

The third category is **metric confusion**. The most common examples are treating schema-valid as semantic-valid, treating human preference win rate as factual accuracy, and treating high tool-call rate as high groundedness. Once these metrics are not split apart, the model can perform beautifully on one cheap proxy while simultaneously damaging the capability that actually matters.

The fourth category is **silent online regression**. Model weights do not change, but the template, tokenizer, tool version, or retrieval index does. Offline sets stay green while online behavior drifts sharply. Without shadow traffic, A/B testing, and replay sets, these problems are usually discovered last.

8.4 Practical Conclusion: Evaluation Is Part of the Training Loop

A mature post-training team does not treat evaluation as a "final acceptance step" after training is done. It treats evaluation as part of the training loop itself. A robust process usually looks like this:

1. First build offline sets and slice sets around the most important failure modes;
2. After each training run, run automated regression first, then inspect the slices;
3. Run human evaluation and red-teaming on important versions;
4. After deployment, continue collecting shadow traffic, failure samples, and online drift signals;
5. Feed those failed slices back into the next round of training data, preference pairs, or verifier cases.

In other words, **alignment is not "train → test once." It is a closed-loop system of "train → evaluate → failure analysis → data feedback."**

9 Engineering Cases

Case 1: After safety data was weighted more heavily, the model started over-refusing

Symptom: the model became more stable at refusing high-risk requests, but it also began refusing a large number of finance questions that should have been answerable safely, such as ordinary budget explanation, approval-process clarification, and reimbursement questions within policy.

Root cause: the SFT or preference data overemphasized refusal demonstrations, or treated "looks sensitive" as "must refuse." The model did not learn a fine-grained boundary. It learned a crude heuristic: if it smells like risk, refuse first.

Engineering fix: rebalance the proportion of safety data, add positive examples that look sensitive but are actually safe to answer, split refusal evaluation into precision and recall instead of looking only at total refusal rate, and add an intermediate label for "answerable, but only with constraints" in ambiguous boundary cases.

Case 2: The JSON was perfectly valid, but the action was completely wrong

Symptom: the tool-call JSON was always valid and passed the validator, but the model often chose the wrong tool, filled in the wrong `account_id`, or used a future date as the query time. Structurally, "everything was correct." Business-wise, it was doing the wrong thing.

Root cause: the team treated schema-valid as success and left semantic-valid unmeasured. Training and evaluation rewarded only "can be parsed structurally," without checking whether the action was consistent with the task, the permission boundary, and the world state.

Engineering fix: separate structural correctness from semantic correctness as two classes of metrics and two classes of data; add counterexamples that are structurally legal but semantically wrong during training; and, in evaluation, add argument-level correctness, world-state consistency, and action-authorization checks for tool calls.

Case 3: After SFT, the model sounded like a customer-service script

Symptom: the model followed instructions and produced correct structure, but the answers were long, rigid, and highly repetitive. Users felt it "sounded like customer-service scripts" or "like a standard-answer generator."

Root cause: the SFT labels were too long and too uniform in style; a large share of the samples came from the same synthetic source; the training data overemphasized politeness and completeness and lacked the variation in information density found in real conversations.

Engineering fix: shorten target answers, explicitly add demonstrations of "answer directly first, then explain," mix in more natural real interactions, and use length-sliced evaluation plus user preference comparison instead of a pure benchmark.

Case 4: After RLHF, the model became increasingly sycophantic

Symptom: the model started using lines like "Good question," "You are absolutely right," and "Your judgment is very insightful" with increasing frequency, even when none of that helped complete the task.

Root cause: the reward model mistakenly treated politeness, flattery, and length as proxies for quality. In RL, the policy learned to amplify these high-reward but low-value surface features.

Engineering fix: redesign the preference rubric to include truthfulness, calibration, and proper expression of uncertainty; add counterexamples that sound nice but are factually weak; and slice sycophancy explicitly in evaluation instead of relying only on preference win rate.

Case 5: The system had retrieval tools, but the model still guessed

Symptom: the system already had a knowledge base and search tools, but the model often answered directly without evidence, and even stated outdated information with great confidence.

Root cause: the training data contained too weak a pattern for "must retrieve when information is insufficient"; the runtime system did not treat "answering without evidence" as a failure; and evaluation looked only at final language quality, not at grounding.

Engineering fix: add tool trajectories, require citations for key conclusions, include "is the claim evidence-supported" in automated evaluation, and when necessary force tool-first behavior in high-risk settings instead of allowing direct answers.

Case 6: Alignment got better, but reasoning got worse

Symptom: the model became more polite, safer, and more stable, yet clearly regressed on complex reasoning tasks: it stopped earlier, used more conservative language, and reached direct conclusions less often.

Root cause: preference or safety optimization over-suppressed exploratory behavior and direct conclusion-giving; the training data over-rewarded conservative phrasing, and the same strategy leaked into reasoning tasks.

Engineering fix: split evaluation by task type—dialogue alignment, tool tasks, and math / code reasoning should be evaluated separately; preserve more direct and more verifier-constrained objectives specifically for reasoning settings; and do not force one monolithic preference objective across all tasks.

Case 7: The model did not change, but online behavior suddenly did

Symptom: offline evaluation showed no obvious regression, yet after deployment the tool-call boundary, refusal position, and multi-turn dialogue behavior all changed.

Root cause: training and inference used different chat templates, or the tokenizer / special-token versions drifted. The model weights did not change, but the input distribution quietly did.

Engineering fix: treat the chat template as an interface contract; pin template and tokenizer versions; keep golden regression samples for representative conversations at both string level and token-ID level; and diff the template before deployment instead of looking only at prompt wording.

10 Chapter Summary

This chapter is about one thing: how to turn "a model that originally generates text" into "an assistant that can reliably do work in real systems." Post-training is not the name of one algorithm. It is an engineering chain that runs from behavior shaping all the way to deployment cost control.

The six phases can be understood as "first usable, then pleasant to use, then reliable, then cheap." SFT solves "first get the format and the role right"; its key points are template consistency and supervision boundaries. PEFT solves

"how to make many specialized versions from the same base model at low cost"; its key points are adapter design, routing correctness, and version management. Preference alignment solves "the answer is correct, but not pleasant to use"; its key points are learning user preference together with safety boundaries and controlling behavior drift. Tool use and RAG solve "can speak but cannot act"; the key is to stabilize the full chain of calling tools, reading results, and answering from evidence. Reasoning and agent RL solve "looks right but is not necessarily right"; the key is to introduce verifiable reward and let the verifier drive checkable correctness. Distillation solves "the quality is good enough, but deployment is too expensive"; the key is to move capability into a smaller model while preserving safety and format behavior.

If you want the single most important sentence about post-training, use this: post-training is about **productizing capability**. It converts a model's raw generation ability into controllable behavior, verifiable quality, and acceptable cost. Without post-training, a model may simply "look smart." With post-training, it becomes much more likely to be reliable in production.

In practice, it helps to apply the same frame to every phase: which training lever did you change, what metric proves that it actually improved, and if something goes wrong, can you roll it back quickly? A truly stable solution is not "one locally optimal point." It is when the training lever, the evaluation loop, and the rollback mechanism all work at the same time.

10.1 Question Summary

1. What is post-training?
Post-training is the process of taking a pretrained base language model and further transforming it into something usable as an assistant. It usually includes SFT, preference optimization, tool-use training, reasoning training, and distillation. The goal is to move the model from "can continue text" to "can follow instructions, work safely, and be deployed."

2. Why can't a pretrained model be used directly as an AI assistant?
Because the pretraining objective only fits the text distribution. It does not fit the behavior the product wants. A base model will produce plausible-looking text, but it does not naturally obey roles, formats, tool interfaces, and safety boundaries, nor does it know when to refuse and when to call external systems.

3. What is instruction tuning, and how is it different from pretraining?
Instruction tuning is, in essence, a kind of supervised fine-tuning for task behavior and role behavior. Pretraining teaches the model to "keep writing like the corpus." Instruction tuning teaches it to "complete user tasks like an assistant."

4. Why is alignment necessary?
Because products are not language benchmarks. A deployable assistant must simultaneously satisfy helpfulness, safety, truthfulness, schema validity, tool re-

liability, and cost constraints. Alignment is the process of shaping behavior under those constraints.

5. Why is modern post-training multi-stage?
Because different phases fix different failures: SFT fixes behavior and format, preference optimization fixes quality ranking, tool training fixes external action, reasoning RL fixes verifiable correctness, and distillation fixes deployment cost. They are not different names for the same thing. They are levers at different layers of the system.

6. What is SFT, and when is it the first choice?
SFT uses high-quality demonstration answers to train the model to output according to instructions, role, and format. It is the first lever for behavior and format failures. If the main problem is knowledge gaps, grounding, or verifier absence, SFT is usually not the first choice.

7. When should you choose PEFT?
PEFT is a good fit when you need many specialized versions, have limited budget, and want to iterate quickly on task variants or tenant variants. Full fine-tuning is more suitable when the capability shift is very deep or the number of versions is small. Prompt-only adaptation is fine for low-risk exploration, but it is usually not stable enough.

8. How is preference data collected?
First generate multiple candidate answers for the same prompt, then ask humans or AI-assisted reviewers to compare them and select the better one, and finally build `(prompt, chosen, rejected)` triples. The key is not just "was a winner selected," but whether the rubric is actually rewarding the product behavior you care about.

9. What is the difference between RLHF and DPO?
RLHF first learns a reward model, then does online RL. DPO directly optimizes the relative probability of chosen and rejected answers on offline preference pairs. The former is more expressive but has higher system cost; the latter is lighter, more stable, and easier to ship first.

10. What is schema-valid tool output, and what is semantically correct tool use?
Schema-valid means only that the output is structurally parseable and satisfies a JSON schema or function signature. Semantic correctness means the tool choice is correct, the arguments are correct, the action is consistent with permissions and world state, and the result is consumed correctly. The first is syntax-level. The second is task-level.

11. Why is a verifier often more important than a larger model?
Because on verifiable tasks, the quality of the feedback determines the direction of training. Without a verifier, even a large model mostly "guesses" from language priors. With a strong verifier, a smaller model can learn much more effective trajectories around real task-success signals.

12. What does a modern post-training pipeline usually look like?
A common order is: SFT → PEFT (optional) → preference alignment → tool use / RAG → reasoning / agent RL (optional) → distillation. The logic is: first make the model usable, then make it more like the assistant your product wants, then make it able to do work and verify, and finally make it cheap enough to deploy.

13. How do you evaluate alignment quality?
You cannot rely on one number. At minimum you need to look at preference win rate, factuality / groundedness, safety, schema validity, tool precision / recall, latency, and cost at the same time. Mature teams also include red-teaming, online replay, and shadow traffic in the release gate.

14. How do you handle the tension between helpfulness and safety?
Do not discuss it abstractly as "more safety or more helpfulness." Slice the scenarios. For each slice, define three boundaries—answerable, answerable with constraints, and must-refuse—and then evaluate precision and recall separately instead of looking only at total refusal rate.

15. Why is post-training usually iterative?
Because once the model updates, the data distribution, failure modes, and user interaction patterns all change. Old preference pairs go stale, new tool errors appear, and new risk boundaries surface. So post-training is naturally an iterative loop of training, evaluation, failure analysis, and feedback.

16. Why distill after the model is already aligned?
Because "a teacher model that is already very good" is not the same thing as "a model already suitable for direct deployment." Distillation solves cost realignment: move the teacher's key behaviors into a student model that is faster, cheaper, and easier to deploy, then run separate long-tail and safety regression on that student.

10.2 Practical Checklist for AI Engineers

Compressed into engineering actions, this chapter becomes the following checklist: identify the failure first, then choose the first lever accordingly.

If your main failure is…	First-priority lever	What not to rush into
Output format is unstable, the role drifts, or tool fields are often wrong	SFT + chat template + completion-only mask	Do not rush into more complex preference optimization first
You need many specialized versions, but full-parameter fine-tuning is too expensive	PEFT + adapter routing	Do not save on regression testing just because training is cheaper

If your main failure is…	First-priority lever	What not to rush into
Two answers are both acceptable, but one is clearly more like the answer the product wants	preference data + DPO / ORPO	Do not treat preference win rate as factual accuracy
It does not call tools when it should, or does not stop when it should	tool trajectories + schema + validator + constrained decoding	Do not rely only on repeating the instruction in the prompt
Multi-step tasks often go off track halfway through	verifier + process reward + search / self-training loop	Do not mistake "longer chain-of-thought" for "stronger reasoning"
Cost and latency are already too high to ship	distillation + targeted regression	Do not ship the student model by looking only at an overall benchmark

Four higher-level principles are worth repeating:

1. **Choose methods by failure type before you choose them by popularity.**
   Whether the problem is behavioral, preferential, tool-related, verifier-related, or deployment-cost-related determines the first lever.
2. **Manage the training objective and the go-live gate separately.**
   Reward is not the release gate. Preference win rate is not production readiness. Deployment requires multiple constraints to be satisfied at the same time.
3. **Treat tools as contracts, not as decorative text.**
   Training is responsible for "when to call, what to call, and how to consume the result." Runtime constraints are responsible for "is the output valid and executable."
4. **Treat the evaluation stack as a system as important as the training stack.**
   Without automated regression, slice sets, human evaluation, red-teaming, and online replay, stable post-training iteration is almost impossible.

11 References

1. Ouyang et al. *Training Language Models to Follow Instructions with Human Feedback*. 2022.

2. Wang et al. *Self-Instruct: Aligning Language Models with Self-Generated Instructions*. 2022.

3. Hugging Face. *Chat Templating* documentation.

4. Hugging Face TRL. *SFTTrainer* documentation.

5. Hu et al. *LoRA: Low-Rank Adaptation of Large Language Models*. 2021.

6. Dettmers et al. *QLoRA: Efficient Finetuning of Quantized Language Models*. 2023.

7. Schulman et al. *Proximal Policy Optimization Algorithms*. 2017.

8. Williams. *Simple Statistical Gradient-Following Algorithms for Connectionist Reinforcement Learning*. 1992.

9. Rafailov et al. *Direct Preference Optimization: Your Language Model is Secretly a Reward Model*. 2023.

10. Hong et al. *ORPO: Monolithic Preference Optimization without Reference Model*. 2024.

11. Schick et al. *Toolformer: Language Models Can Teach Themselves to Use Tools*. 2023.

12. Qin et al. *ToolLLM: Facilitating Large Language Models to Master 16000+ Real-world APIs*. 2023.

13. Patil et al. *Gorilla: Large Language Model Connected with Massive APIs*. 2023.

14. OpenAI. *Function Calling* documentation.

15. OpenAI. *Introducing Structured Outputs in the API*. 2024.

16. Lewis et al. *Retrieval-Augmented Generation for Knowledge-Intensive NLP Tasks*. 2020.

17. Lightman et al. *Let's Verify Step by Step*. 2023.

18. Wang et al. *Math-Shepherd: Verify and Reinforce LLMs Step-by-step without Human Annotations*. 2023.

19. Shao et al. *DeepSeekMath: Pushing the Limits of Mathematical Reasoning in Open Language Models*. 2024.

20. Liu et al. *Understanding R1-Zero-Like Training: A Critical Perspective (Dr. GRPO)*. 2025.

21. Zheng et al. *Group Sequence Policy Optimization*. 2025.

22. Yu et al. *DAPO: An Open-Source LLM Reinforcement Learning System at Scale*. 2025.

23. Yan et al. *Learning to Reason under Off-Policy Guidance (LUFFY)*. 2025.

24. Zelikman et al. *STaR: Bootstrapping Reasoning With Reasoning*. 2022.

25. Zhang et al. *ReST-MCTS*: LLM Self-Training via Process Reward Guided Tree Search*. 2024.

26. Hinton et al. *Distilling the Knowledge in a Neural Network*. 2015.

27. Kim, Rush. *Sequence-Level Knowledge Distillation*. 2016.

28. Chen et al. *Knowledge Distillation of Black-Box Large Language Models*. 2024.