O'REILLY®

Post-Training AI

A Practical Guide to Fine-Tuning and Reinforcement Learning

Early Release
RAW & UNEDITED

Ben Burtenshaw

Post-Training AI

A Practical Guide to Fine-Tuning and Reinforcement Learning

With Early Release ebooks, you get books in their earliest form—the author's raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.

Ben Burtenshaw

Post-Training AI

by Ben Burtenshaw

Published by O'Reilly Media, Inc., 141 Stony Circle, Suite 195, Santa Rosa, CA 95401.

O'Reilly books may be purchased for educational, business, or sales promotional use. Online editions are also available for most titles (https://oreilly.com). For more information, contact our corporate/institutional sales department: 800-998-9938 or *corporate@oreilly.com*.

Acquisitions Editor: Nicole Butterfield

Development Editor: Michele Cronin

Production Editor: Elizabeth Faerm

Copyeditor: TO COME

Proofreader: TO COME

Indexer: TO COME

Cover Designer: TO COME

Cover Illustrator: TO COME

Interior Designer: David Futato

Interior Illustrator: Kate Dullea

August 2027: First Edition

Revision History for the Early Release

- 2026-08-18: First Release

See https://oreilly.com/catalog/errata.csp?isbn=9798295900266 for release details.

979-8-295-90023-5

[LSI]

Brief Table of Contents (*Not Yet Final*)

Chapter 1. Introduction to Post-Training

A NOTE FOR EARLY RELEASE READERS

With Early Release ebooks, you get books in their earliest form—the author's raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.

This will be the 1st chapter of the final book. Please note that the GitHub repo will be made active later on.

If you'd like to be actively involved in reviewing and commenting on this draft, please reach out to the editor at *mcronin@oreilly.com*.

Every useful language model in production today went through post-training. The models behind chatbots, coding assistants, and customer-service agents are not next token prediction models. They have been trained further to follow instructions, refuse unsafe requests, reason through tasks, and match the product they serve.

Post-training is this second training phase. It is shorter and cheaper than pre-training, but it determines much of what users experience.

This chapter defines post-training, separates it from pre-training, and introduces supervised fine-tuning, preference learning, and reinforcement learning with verifiable rewards. It also explains why open base models and modern tooling have made these techniques usable outside frontier labs.

1.1 What post-training is and isn't

Post-training is the second training phase a modern language model goes through. The three subsections below separate it from pre-training, name the techniques that make up a typical post-training pipeline, and mark the limits of what post-training can change.

1.1.1 Pre-training vs post-training: the two phases of a modern LLM

Building a modern language model happens in two phases. Pre-training comes first. It consumes trillions of tokens of text (web pages, books, code repositories, scientific papers) and trains the model to predict the next token in a sequence. This is an unsupervised objective. The model is not told what good output looks like. Instead, it is told what the next word is across the entire corpus. The result is a model that has absorbed the statistical structure of language, along with a broad and noisy representation of world knowledge.

Pre-training is expensive. Training a frontier model from scratch requires thousands of GPUs running for weeks or months, and datasets measured in the tens of terabytes. The investment buys something valuable but incomplete: a base model that can continue any text sequence in a plausible way, but cannot respond to a user's instructions. Prompt a base model with a question and it may answer the question, or it may generate five more questions, or it may continue with something that looks like a textbook paragraph. It is completing text, not responding to a user's instructions.

Post-training is the second phase. It takes the base model and applies smaller, targeted training runs that shape the model's behavior. If pre-training teaches the model what language looks like, post-training teaches the model how to use that knowledge. The training data is smaller (thousands to millions of examples rather than trillions of tokens), but it is curated, structured, and designed to produce specific behavioral outcomes.

The two phases optimize different signals. Pre-training on a broad corpus builds representations that transfer across tasks. Post-training on curated data uses those representations to produce targeted behavior. Training a

model from scratch on curated instruction data produces worse results because the model never develops the broad representations that make it flexible. Skipping post-training produces a model that has knowledge but no discipline in how it uses that knowledge.

1.1.2 The post-training recipe: SFT, preference learning, RL

Post-training usually combines three stages. Each stage changes a different part of the model's behavior.

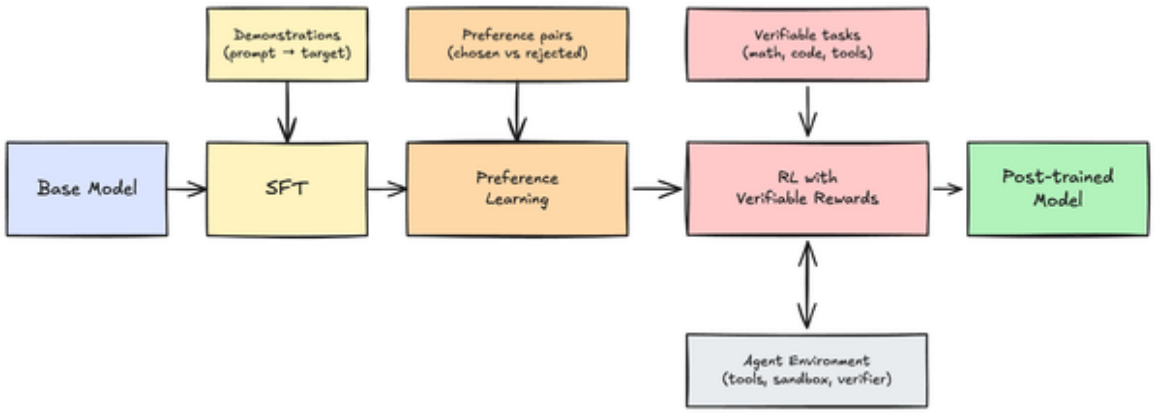


Figure 1-1. TO COME

Supervised fine-tuning (SFT) trains the model on examples of desired input-output behavior. Each example pairs a prompt with a target completion. The objective is still next-token prediction, but the data is a curated set of demonstrations for which the model is trained to produce the target completion. SFT teaches format and style: how to structure an answer, when to use bullet points, how to handle multi-turn conversations, and how to produce outputs in JSON or code.

SFT teaches imitation. It does not give the model a signal that separates a great answer from a mediocre one, and it cannot teach the model to be better than its demonstrations.

Preference learning adds that signal by moving the model from imitation toward optimization. It teaches the model to prefer outputs that humans judge as more helpful, accurate, concise, cautious, or appropriate for the

task. It trains the model on pairs of responses to the same prompt, where one response is marked as preferred and the other as rejected. The model learns to assign higher probability to the preferred responses and lower probability to the rejected ones. Classic reinforcement learning from human feedback (RLHF) trains a reward model on those comparisons, then uses reinforcement learning to optimize the language model against that reward. Direct preference optimization (DPO) removes the separate reward-model step and trains the language model directly on preference pairs.

Reinforcement learning with verifiable rewards trains the model on tasks where correctness can be checked automatically: mathematical problems with known answers, coding tasks with unit tests, and structured reasoning tasks with verifiable conclusions. The model generates candidate solutions, a verifier checks them, and the model is updated to produce more correct solutions. The reward signal comes from the task itself.

DeepSeek-R1 and subsequent reasoning models showed that verifiable rewards can improve performance on mathematics, code, and multi-step reasoning in ways that SFT and preference learning alone do not.

Not every model needs all three stages. A structured extraction model may need only SFT. A general assistant usually benefits from preference learning. A math or coding model may need verifiable rewards. The choice depends on the task, the base model, the data, and the available compute.

1.1.3 What post-training cannot fix

Post-training can shape format, install preferences, and teach reasoning on verifiable tasks. However, it still operates within constraints set by the base model.

Post-training cannot add knowledge that is not already present in some form in the base model’s weights. If the base model was not trained on medical literature, post-training will not turn it into a reliable medical assistant. SFT can teach the model to respond in a medical format, and preference learning can teach it to prefer cautious, well-hedged medical responses, but neither can inject factual medical knowledge that the model never learned during

pre-training. The model will produce fluent, well-formatted medical text that may be wrong.

Post-training cannot fix fundamental capability gaps. If the base model struggles with arithmetic, post-training can improve its arithmetic performance on the specific kinds of problems it sees during training, but it cannot give the model a general ability to do arithmetic that it lacked before. Reinforcement learning with verifiable rewards can push the model further than SFT by training it to generate and check chains of reasoning, but even RL works within the representational capacity that pre-training established.

Post-training cannot reliably eliminate behaviors that are deeply embedded in the pre-training distribution. Safety fine-tuning can teach a model to refuse harmful requests in the formats it has seen during training, but adversarial users routinely find novel prompt formats that bypass these refusals. This is not a failure of post-training technique. It is a consequence of the fact that post-training adjusts the model's behavior on a surface that pre-training defined. The deeper the pattern, the harder it is to override with a relatively small amount of post-training data.

Post-training also cannot substitute for good data infrastructure. The quality of post-training depends on the data and reward signals it uses. A poorly constructed SFT dataset will teach the model bad habits. A noisy preference dataset will teach a reward model that does not reflect real human preferences. An unreliable verifier will reward incorrect solutions. Data quality, reward design, and failure analysis decide how much the training run can improve the model.

Finally, post-training cannot solve the alignment problem in any complete or permanent sense. It can make models more helpful, more honest, and safer within the distribution of situations the post-training data covers. It cannot guarantee that a model will behave well in situations that are very different from anything it has seen. The field continues to work on this problem, and the techniques in this book represent the current best practice, not a solved problem.

These limitations help you choose the right base model, set realistic expectations, and spend your post-training budget where it can change behavior.

1.2 Why Post-Training Matters Now

Three things have made post-training a practical option for engineers outside frontier labs: the cost and latency of frontier APIs at scale, the release of capable open base models, and the rise of on-device AI applications. The four subsections below cover each driver in turn, then extend the story to agentic tasks where verifiable rewards do the training.

1.2.1 The economics: small post-trained models vs frontier APIs

Post-training is not an essential step for all AI projects. In most cases, if you want to use a post-trained model, you can call an API and pay per token. General models work, and for many applications it is the right choice. But token costs, rate limits, governance constraints, and model-version changes become harder to accept as applications grow.

The per-token cost of a frontier API is small in absolute terms, but it scales linearly with usage. An application that processes a million customer-service conversations per month, or that runs an AI coding assistant for a hundred-person engineering team, or that embeds a language model in a consumer product with millions of users, faces API costs that grow proportionally with adoption. There is no economy of scale on the demand side. The marginal cost of the millionth conversation is the same as the first.

A post-trained model that you own and serve yourself has a different cost structure. There is an upfront investment in training (compute, data, engineering time) and an ongoing cost for inference. But the inference cost per token on an optimized model served on cloud hardware is a fraction of the per-token cost of a frontier API. For applications with high volume, the

crossover point where self-hosted inference becomes cheaper than API calls arrives earlier than most teams expect.

Cost is only one dimension. Control is the other. An API-served model is a dependency you do not control. The provider can change the model's behavior, pricing, rate limits, terms of service, or availability. You cannot inspect the weights, reproduce the model's behavior, or modify its responses at the model level. You can prompt-engineer, but prompting is a blunt instrument compared to training.

A model you have post-trained is a model you own. You control what it does, how it responds, and how it changes over time. You can specialize it for your domain or use case. You can run it on your own infrastructure, behind a security boundary, or with your own data governance. You can iterate on it weekly or daily, pushing improvements without waiting for a provider's release cycle.

1.2.2 The open-source post-training wave: Mistral, Gemma, Qwen, DeepSeek, Olmo

The economic argument only works if you have something to post-train. For years, the bottleneck was access to a good base model: Pre-training a competitive model from scratch was a multi-million-dollar project that required datasets, infrastructure, and expertise that very few organizations had.

That bottleneck broke open between 2023 and 2025. Meta released the Llama series under permissive licenses, providing base models at 7B, 13B, 70B, and eventually 405B parameters that matched or approached frontier performance. Google released the Gemma series. Alibaba released Qwen with full training recipes and Apache 2.0 licensing. DeepSeek released a series of increasingly capable models, culminating in DeepSeek-R1, which demonstrated that reinforcement learning with verifiable rewards could produce reasoning capabilities competitive with the best closed models.

These releases changed the landscape. An engineer who wants to post-train a model no longer needs to pre-train one first. They can start from a base

model that has already absorbed broad knowledge and language capability, and focus entirely on the post-training pipeline that shapes that capability into the behavior they need. The base model is free, the tooling is open-source, and the techniques are documented.

Three foundational releases show what this pipeline looks like in practice: Llama 3, Qwen 2.5, and DeepSeek-R1.

The Llama 3 post-training recipe illustrates the state of practice. Meta's team used an iterative pipeline: rejection sampling to generate candidate responses, supervised fine-tuning on the best candidates, and direct preference optimization to align the model with human preferences. They ran this loop for five rounds, using each round's output as the starting point for the next round's data generation. The training data was synthetic, generated by the previous model version and filtered for quality. This approach produced one of the most capable open-weight instruction-following models available at the time.

Qwen 2.5 followed a similar pattern, combining SFT, DPO, and GRPO across approximately one million post-training examples. The full model family, from 0.5B to 72B parameters, was released under Apache 2.0 with training recipes, evaluation scripts, and integration support for quantization and fine-tuning frameworks.

DeepSeek-R1 pushed further. The DeepSeek team trained a model using Group Relative Policy Optimization (GRPO), a reinforcement learning algorithm that does not require a separate critic network, directly on tasks with verifiable answers. Their initial experiment, DeepSeek-R1-Zero, skipped supervised fine-tuning entirely and trained the base model with RL alone. The result was a model that spontaneously developed chain-of-thought reasoning behavior without being explicitly taught to do so. The full DeepSeek-R1 model, which added SFT and iterative refinement, achieved performance competitive with OpenAI's o1 on mathematics, code, and reasoning benchmarks. DeepSeek released the model weights, the distilled variants, and the code under the MIT license.

Post-training is no longer confined to frontier labs. The base models are available. The techniques are published and reproduced. The tooling exists. The remaining work is engineering: designing the data, reward functions, and training configurations for a specific application.

1.2.3 From prompting to owning the weights

Most engineers' first experience with language models is through prompting. You write a system prompt that describes the persona, the constraints, and the task format you want. You send the prompt along with each user query to an API. The model responds according to the instructions in the prompt, more or less.

Prompting works well for prototypes and applications where behavior can vary. Production systems need lower prompt overhead and more consistent behavior.

A prompt is consumed with every request. A system prompt that is 2,000 tokens long means that every API call pays for those 2,000 tokens of context before the user's actual input is processed. For high-volume applications, this overhead is significant. More importantly, the model is interpreting the prompt each time, which means its adherence to the prompt's instructions can vary across inputs. A long, detailed system prompt can improve consistency, but it can also create conflicts. The model may attend to one part of the prompt at the expense of another, especially when the user's input creates tension with the system instructions.

Retrieval-augmented generation (RAG) extends prompting by injecting relevant documents into the context window at inference time. This addresses the knowledge limitation: the model can now reference information that was not in its training data. But it does not address the behavior limitation. The model still relies on prompt instructions for its persona, format, and decision-making, and those instructions are still consumed and interpreted anew with every request.

Post-training changes the cost structure and the reliability model. When you fine-tune a model, the behaviors you train become part of the model's

weights. They do not consume context tokens. They do not need to be re-interpreted with every request. The instructions have become learned behavior.

This affects behaviors that need to stay consistent across inputs: tone, formatting conventions, safety boundaries, domain-specific terminology, and reasoning patterns. A prompted model re-reads its instructions on every call, but post-trained model has internalized them.

Many production systems combine the two. A post-trained base model handles stable behaviors. Prompting handles task-specific variation. RAG supplies dynamic knowledge. This layered approach gives you the reliability of training with the flexibility of prompting.

1.2.4 Training for agentic tasks with verification

The most recent shift in post-training is driven by the rise of agentic applications. These are systems where a language model takes actions in an environment over multiple steps: a coding agent that writes, runs, and debugs code; a research agent that searches, reads, and synthesizes information; or a customer-service agent that reads account data, makes decisions, and executes transactions.

These applications demand capabilities that conversational post-training does not produce. An agent needs to plan across multiple steps, recover from errors, use tools correctly, and verify its own outputs. These are learned behaviors, not prompt-engineered ones, and they respond well to reinforcement learning because the outcome of an agent's actions is often verifiable. The code compiles or it does not. The test passes or it does not. The retrieved document answers the question or it does not.

Reinforcement learning with verifiable rewards is a natural fit for training agents. The model generates a trajectory (a sequence of actions and observations) and receives a reward based on whether the trajectory achieved the desired outcome. The model is updated to produce more trajectories that succeed and fewer that fail. This is the same loop that

produced reasoning models like DeepSeek-R1, extended to multi-step tasks with tool use and environmental interaction.

Training agents introduce new challenges that do not arise in single-turn text generation. The episodes are longer, which makes credit assignment harder. If the agent fails after ten steps, which step was the mistake? The action space is larger, because the model can call tools, write code, and produce structured outputs in addition to natural language. The environment matters, because the model needs something to interact with during training: a sandbox to run code in, an API to call, a simulated user to converse with.

These challenges are active areas of development, and the tooling is evolving. Later chapters cover environment design, reward construction for multi-step tasks, and training configurations for agentic RL. The target is a model that can plan, use tools, recover from errors, and verify its work.

1.3 The Post-Training Stack

The post-training stack has four main parts: SFT, preference learning, reinforcement learning with verifiable rewards, and agent environments. Each part deals with a different training signal, and we'll discuss each in turn below.

1.3.1 Supervised fine-tuning: teaching format and behavior

Supervised fine-tuning is the entry point to post-training. It takes a base model that generates text by continuing sequences and turns it into a model that responds to instructions. The mechanism is straightforward: you give the model examples of the input-output behavior you want, and you train it to reproduce that behavior using next-token prediction objective based on the responses.

The data for SFT is structured as conversations or instruction-completion pairs. A typical example might consist of a user message ("Explain the

difference between TCP and UDP in two sentences"), a system message that specifies the model's persona and constraints, and an assistant response that demonstrates the desired behavior. The model is trained to predict the tokens of the assistant response, conditioned on everything that came before. Tokens in the user message and system message are typically masked from the loss function so that the model learns to generate the response, not to reproduce the prompt.

The format of these examples matters as much as their content. Modern language models use chat templates, which are structured formats that mark the boundaries between roles (system, user, assistant) using special tokens. These templates vary across model families. Llama uses one format, Qwen uses another, and training libraries like TRL abstract over these differences so that your training code does not need to handle them manually. Getting the template wrong produces a model that generates malformed output or ignores its system prompt.

SFT is the fastest post-training stage. You show the model what you want, and it learns to produce something similar. The quality of the result depends almost entirely on the quality of the data. A thousand high-quality examples with consistent formatting and accurate content will produce a better model than a hundred thousand noisy, inconsistent examples. Data quality dominates data quantity in SFT.

The behavioral effects of SFT are visible immediately. A base model's output will clearly change in format to use a chat template. However, SFT still teaches imitation. It does not teach the model to evaluate the quality of its own outputs or to choose better responses when multiple right options exist.

1.3.2 Preference learning and RLHF: teaching what "good" looks like

Preference learning picks up where SFT leaves off. SFT teaches the model what format to use. Preference learning teaches it what quality means.

The original formulation, reinforcement learning from human feedback (RLHF), was developed at OpenAI and published in the InstructGPT paper in 2022. The pipeline has three stages. First, a model is trained with SFT on demonstration data. Second, a reward model is trained on human preference data: annotators compare pairs of model outputs and indicate which one they prefer, and the reward model learns to predict these preferences. Third, the SFT model is fine-tuned using reinforcement learning, specifically proximal policy optimization (PPO), to produce outputs that score highly according to the reward model while staying close to its SFT starting point to avoid catastrophic forgetting or reward hacking.

The InstructGPT results showed the value of this quality signal. A 1.3B-parameter model trained with RLHF was preferred by human evaluators over the raw 175B-parameter GPT-3. The smaller model did not have more knowledge. It had better judgment about what to do with its knowledge, because the reward model supplied a signal that SFT alone could not provide.

RLHF with PPO is effective but operationally complex. It requires maintaining four models simultaneously during training: the policy model being optimized, a reference model to compute a KL divergence penalty, a reward model, and a value model for advantage estimation. This is memory-intensive and introduces multiple sources of instability. The reward model can be poorly calibrated. The PPO optimization can overfit to reward-model artifacts. The KL penalty can be too strong (preventing learning) or too weak (allowing reward hacking).

Direct preference optimization (DPO), published in 2023, simplified the pipeline. DPO derives a loss function that directly optimizes the language model on preference pairs, without training a separate reward model or running reinforcement learning. The mathematical insight is that the optimal policy under the RLHF objective can be expressed in closed form as a function of the preference data, which means you can optimize for it directly using a supervised-style training loop.

In practice, DPO trains the model on pairs of responses, one chosen and one rejected, and it adjusts the model to increase the probability of the chosen response relative to the rejected one. It requires only two models in memory (the policy and a frozen reference model) instead of four. It is more stable, easier to implement, and less sensitive to hyperparameters than PPO. For these reasons, DPO and its variants (IPO, KTO, ORPO) have become the default preference learning method in most open-source post-training pipelines.

There are multiple configurations of DPO across the ecosystem. For example, Meta's Llama 3 pipeline applies DPO iteratively across multiple rounds of synthetic data generation. Alibaba's Qwen 2.5 combines DPO with GRPO across roughly a million preference and verifiable-reward examples. Both teams produced models competitive with the best closed systems without maintaining the four-model PPO setup that defined the original RLHF pipeline. A team with enough memory to hold two models can now run the same alignment stage, and that accessibility is most of why open-source post-training has matured so quickly.

Preference learning, whether via RLHF or DPO, teaches the model to distinguish good outputs from bad ones along dimensions that are hard to specify in a dataset of demonstrations. It can teach the model to be more helpful, more concise, more accurate, or more cautious, depending on what the preference data rewards. But it relies on the quality of the preference annotations. If the annotators disagree, or if the preference data rewards superficial features such as length or formatting, the model will learn the wrong lessons.

1.3.3 Reinforcement learning with verifiable rewards: teaching reasoning and capability

The third stage of the modern post-training stack goes beyond human preferences. Instead of training the model to produce outputs that humans prefer, it trains the model to produce outputs that are correct, where correctness is verified by an automated system rather than judged by an annotator or model.

This stage requires tasks where the answer can be checked. Mathematics is the canonical domain: the model generates a solution, and a grader checks whether the final answer matches the known correct answer. Code is another: the model writes a function, and a test suite runs against it. Logical puzzles, constraint satisfaction problems, and planning tasks also qualify.

The algorithm that has driven most of the recent progress in this stage is Group Relative Policy Optimization (GRPO), which was introduced by DeepSeek. GRPO works by sampling a group of candidate responses for each prompt, scoring each response with the verifier, computing advantages relative to the group mean, and updating the model to increase the probability of above-average responses and decrease the probability of below-average ones. Unlike PPO, GRPO does not require a learned value function or critic network. The group of samples serves as its own baseline. This makes GRPO simpler to implement, more memory-efficient, and more stable in practice.

DeepSeek-R1-Zero, trained with GRPO on verifiable tasks without any SFT, spontaneously developed chain-of-thought reasoning. The model learned to break problems into steps, check its own work, and revise its approach when an intermediate step failed. It was never shown examples of this behavior. The reasoning emerged because it was rewarded: solutions that included careful reasoning were more likely to arrive at correct answers, and the RL training amplified this pattern.

The full DeepSeek-R1 model, which combined SFT with GRPO training, achieved performance competitive with the strongest closed reasoning models on mathematics, coding, and science benchmarks. It was released with full weights under an open license, and the training approach has been reproduced by multiple teams using open-source tooling.

Reinforcement learning with verifiable rewards separates the current generation of post-trained models from the previous one. SFT and preference learning can produce a model that follows instructions well and produces outputs that humans rate highly. RL with verifiable rewards can produce a model that solves problems it was never shown solutions to,

because the training signal rewards successful reasoning rather than imitation.

This stage is also the most demanding in terms of infrastructure. Each training step requires generating multiple complete responses for each prompt, running each response through a verifier, and computing policy gradient updates based on the results. The generation step is the bottleneck. Producing long chains of thought for thousands of prompts per batch requires fast inference, typically served by a dedicated inference engine like vLLM running alongside the training process.

1.3.4 Post-training agents and the role of environments

The three stages described above (SFT, preference learning, and RL with verifiable rewards) were developed primarily for single-turn text generation. The model receives a prompt, produces a response, and is evaluated on that response. But many of the most valuable applications of language models involve multi-turn interaction with external systems: writing and executing code, searching and reading documents, calling APIs, navigating user interfaces.

Training models for these agentic tasks extends the post-training stack in a specific direction: the model needs an environment to interact with during training. An environment provides the model with observations (the output of a code execution, the content of a retrieved document, the state of a user interface), accepts the model's actions (tool calls, code to execute, messages to send), and returns outcomes that can be used to compute rewards.

The environment protocol is conceptually simple: reset the environment to an initial state, let the model take an action, return an observation and optionally a reward, and repeat until the episode ends. This is the standard reinforcement learning loop, applied to language-model agents. Useful environments must produce realistic training signals, vary enough to prevent memorization, and run fast enough to generate the thousands of episodes that RL training requires.

Environment design is an area of active development. The OpenEnv project, for example, provides a standardized protocol for connecting language models to training environments, with support for procedural generation (creating new task instances on the fly to prevent the model from memorizing solutions) and sandboxed execution (running the model's code in isolated containers). You can find environments for web navigation, database interaction, and API tool use.

Post-training agents also introduce questions about reward structure. In a single-turn task, the reward is a function of the model's output. In a multi-turn task, the reward might depend on the final outcome (did the agent complete the task?), the efficiency of the trajectory (how many steps did it take?), the quality of intermediate steps (did it use the right tools?), or safety constraints (did it avoid actions that could cause harm?). Decomposing the reward into components and weighting those components correctly is a design problem that does not arise in single-turn post-training.

1.3.5 How the pieces compose in a real pipeline

In practice, post-training is a sequence of checkpoints. Each stage starts from the best checkpoint produced by the previous stage.

A typical pipeline starts with a base model, such as `google/gemma-4-E2B`, and applies SFT first. SFT teaches the model to follow instructions, use the right format, and handle the examples your application depends on. If the SFT model passes your evaluations, the pipeline may stop there.

Preference learning is optional. Add DPO or another preference method when the model already responds in the right format but still needs better judgment about tone, helpfulness, accuracy, or refusal behavior.

RL with verifiable rewards is optional too. Add GRPO or another RL method only when the task has checkable outcomes, such as unit tests, exact math answers, or structured verifiers, and evaluation shows that SFT or preference learning is not enough.

In practice, you may choose to build environments for your tasks but only use them for evaluation, stopping before full reinforcement learning.

Use the shortest pipeline that changes the target behavior. Add later stages only when evaluation shows a gap that the previous stage cannot close. Prompting solves many use cases, SFT solves fewer, and reinforcement learning even fewer.

1.4 What You'll Build in This Book

The rest of the book follows a single model through each stage of post-training. This section outlines the structure we will use throughout this book.

1.4.1 A running-example model: from base to post-trained

This book follows a single model from base checkpoint to post-trained system. You will start with an open-weight base model of 4B parameters, and apply each post-training technique to it in sequence. A 4B-parameter model is large enough to demonstrate real post-training behavior on tasks like coding, but small enough to be easily trained.

Each chapter builds on the previous one. Chapter 2 implements the core training loops from scratch in PyTorch, so you understand the mechanics before using higher-level abstractions. Chapters 3 through 6 cover the algorithms, data, and infrastructure for each post-training stage using TRL and the Hugging Face ecosystem. Chapter 7 connects the stages into a production pipeline. Chapter 8 pushes into reasoning and reinforcement learning with verifiable rewards. Chapter 9 covers safety, evaluation, and operating a post-training experiment.

Each stage starts from the checkpoint produced by the previous stage. Exercises are a sequence of operations that you can run, inspect, and modify.

1.4.2 Tooling overview: TRL, transformers, PEFT, the Hub

The Hugging Face ecosystem provides the primary tooling for this book. These tools are widely used for open-source post-training, support the techniques covered here, and integrate with the broader Python ML stack without much glue code.

Transformers Reinforcement Learning (TRL) is the library at the center of the post-training workflow. It provides trainers for each stage of the pipeline: `SFTTrainer` for supervised fine-tuning, `DPOTrainer` for direct preference optimization, `GRPOTrainer` for group relative policy optimization.

TRL is built on top of `transformers`, which provides the model architectures, tokenizers, and training infrastructure that the rest of the stack depends on. If you have used `transformers` for inference or for standard fine-tuning, the TRL trainers will feel familiar. They accept the same model and tokenizer objects, use the same configuration patterns, and integrate with the same logging and checkpointing tools.

Parameter-Efficient Fine-Tuning (PEFT) is the library that makes post-training practical on limited hardware. Full fine-tuning requires enough GPU memory to hold the model's weights, the optimizer states, and the gradients. That is roughly 28 GB for a 4B parameter model in half precision. PEFT methods, particularly Low-Rank Adaptation (LoRA) and its quantized variant QLoRA, reduce the number of trainable parameters by orders of magnitude while preserving most of the training quality. LoRA adds small trainable matrices to the model's attention layers and freezes everything else, reducing the memory footprint to the point where a 4B model can be fine-tuned on a single 12 GB GPU.

The Hugging Face Hub is the infrastructure layer. Models, datasets, and training artifacts are stored on and loaded from the Hub. When you train a model, you can push checkpoints to the Hub at each stage, track experiments with model cards, and share your results with the community.

The Hub also hosts the base models you will start from, the datasets you will train on, and the evaluation tools you will use to measure your results.

Later chapters also use vLLM for fast inference during RL training, `datasets` for data loading and preprocessing, and evaluation frameworks for measuring model quality. Each tool appears when the workflow needs it.

1.4.3 Hardware assumptions and how to scale them

The examples in this book are designed to work on a single GPU with at least 12 GB of memory, such as a T4 or equivalent. This is the hardware that most individual practitioners have access to, and it is sufficient for post-training models up to about 4B parameters using parameter-efficient methods like QLoRA. If you have access to a larger GPU, you can train larger models.

If you do not have local GPU access, the book's examples are compatible with cloud GPU providers and with Hugging Face Jobs or Google Colab. The cost of running the examples in this book on cloud GPUs is modest. The book does not assume any specific cloud provider or hardware configuration.

1.4 Summary

This chapter defined post-training as the second training phase that turns a base model into a useful product. It introduced the three techniques that compose a modern pipeline (supervised fine-tuning, preference learning, and reinforcement learning with verifiable rewards), explained why open base models and falling infrastructure costs have made this work accessible outside frontier labs, and outlined the running example you will build over the rest of the book.

Chapter 2 starts that work. You will set up the development environment, load a 4B-parameter base model, and write a minimal training loop in PyTorch so the mechanics are visible before higher-level libraries take over.

Chapter 2. Speed Run Post-Training from First Principles

Chapter 1 explained what post-training is and why it now sits within reach of engineers outside frontier labs. You have the vocabulary: supervised fine-tuning, preference learning, and reinforcement learning with verifiable rewards. You have heard about the challenges and considerations of post-training.

In this chapter, you will build an end-to-end example of post-training from first principles, using simple training loops and minimal dependencies. You will implement the two training loops that anchor post-training, supervised fine-tuning (SFT) and group relative policy optimization (GRPO), from scratch and on a single GPU.

The code is short, with each loop under 100 lines, which lets you focus on the core concepts without layers of abstraction. Think of the chapter as a nano version of the whole book: the smallest thing that still trains a model end to end. The code runs, but it is deliberately stripped down, trading convenience for brevity. It leaves out the infrastructure that a real training

stack provides, and focuses instead on the primary operations that SFT and GRPO perform at the token and parameter level. You are building intuition about post-training methods, through simplified code examples, that the rest of the book builds upon.

The complete, runnable versions of every example live in the book's companion repository at [REPO URL], so you can clone them, run them, and extend them as you read.

This chapter covers post-training in one process: load a model, run an SFT step, run an RL step, and use the model for inference. Later chapters return to each component, slow down, and mature them with production considerations and tooling. You will set up a minimal environment with a single GPU and notebook, define an SFT loop, and probe the model before and after to see what changed. Then you build an RL loop on a small verifiable task and watch the reward climb. The chapter ends with a mental model that separates what SFT does to a model from what RL does, and clarifies when to reach for each.

> **NOTE**
>
> The two loops share a base model and an optimizer. They differ in one place: where the training signal comes from. SFT copies a signal you provide, while RL discovers a signal from a reward you define. That contrast is the through-line of the chapter and the rest of the book.

Setting Up

Before any training, you need a working environment and a model small enough to iterate on. This section names the libraries you will use and pins down a setup that runs end-to-end on modest hardware. The two subsections cover the software stack and the reproducible single-GPU configuration the rest of the chapter assumes.

Python, PyTorch, and Hugging Face

The examples in this chapter use four libraries. PyTorch provides tensors, autograd, and the optimizer[1] . The Hugging Face `transformers` library provides the model and the tokenizer, whilst the `datasets` library loads training data (see Tunstall et al., *Natural Language Processing with Transformers*, Revised Edition, O'Reilly, 2022).

This chapter goes underneath Transformers Reinforcement Learning (`trl`), which is Hugging Face's standard toolkit for post-training methods like SFT and the RL. Instead of calling its convenient trainer classes, you will use `transformers` to load a model and a tokenizer, then write the training loop, the loss, and the update from scratch. That is what we mean by pure PyTorch here. You are not reimplementing the transformer, just removing the trainer abstraction so the mechanics are visible.

A single install line covers the dependencies.

```
pip install torch transformers
```

> **NOTE**
>
> The training loop, the loss function, and the optimizer step are written out by hand in this chapter. The model architecture and the tokenizer come from `transformers`. Writing a transformer from scratch is a different exercise and not the point here. The point is to see how the training signal reaches the weights.

One GPU and one notebook

The whole chapter runs on one GPU with 16 GB of memory, the kind a free or low-cost cloud notebook provides. The model is `Qwen/Qwen3-0.6B`, a 600 million parameter model. At this size the weights, the optimizer state, and the activations fit in memory with room to spare, and a training step takes seconds rather than minutes. Alammar and Grootendorst[2]survey the open-model landscape and the size trade-offs behind a choice like this.

A small model is a deliberate choice. You are not trying to produce a strong model in this chapter. You are trying to see the loops work and to watch a

measurable change in performance. A small model makes the loop fast, and a fast loop makes the change easy to observe. The same code scales to larger models by adding the sharding and generation infrastructure that later chapters cover, but with that scaling comes abstraction.

Three settings make runs repeatable. Set a seed so sampling and shuffling are deterministic. Load the model in `bfloat16` to halve the memory of the weights. Keep generation short during development so each step returns quickly.

```
import torch
from transformers import AutoModelForCausalLM, AutoTokenizer

model_name = "Qwen/Qwen3-0.6B"
tokenizer = AutoTokenizer.from_pretrained(model_name)
model = AutoModelForCausalLM.from_pretrained(model_name,
dtype=torch.bfloat16).to("cuda")
```

This is the starting point for both SFT and RL loops, which load the model the same way. What follows is where they diverge.

> **TIP**
>
> Keep `max_new_tokens` small (8 to 64) while you develop. Short generations make each step fast, and a fast step keeps iteration tight.

A Minimal SFT Loop

Supervised fine-tuning trains a model to reproduce demonstrated outputs. Given pairs of prompts and completions, the model learns to produce those completions. The objective is the same next-token prediction used in pre-training, and only the data changes. This section builds the loop in four steps: the objective, the data format, the implementation, and a probe that confirms the model changed.

Next-token prediction as supervision

A token is a chunk of text, often a word or word-piece, that the tokenizer maps to an integer. A language model reads a sequence of tokens and outputs one score, called a logit, for every token in its vocabulary. Softmax turns those logits into a probability distribution over the next token. For a sequence of length T, the model produces T distributions, one after each position, each predicting the token that should follow. Alammar and Grootendorst[3]give clear visual explanations of tokens and next-token prediction for readers who want a fuller introduction.

Figure 2-1 shows this for a four-token sequence. Each position reads the tokens up to it and emits one distribution over the vocabulary, and the most probable token is the model's prediction.

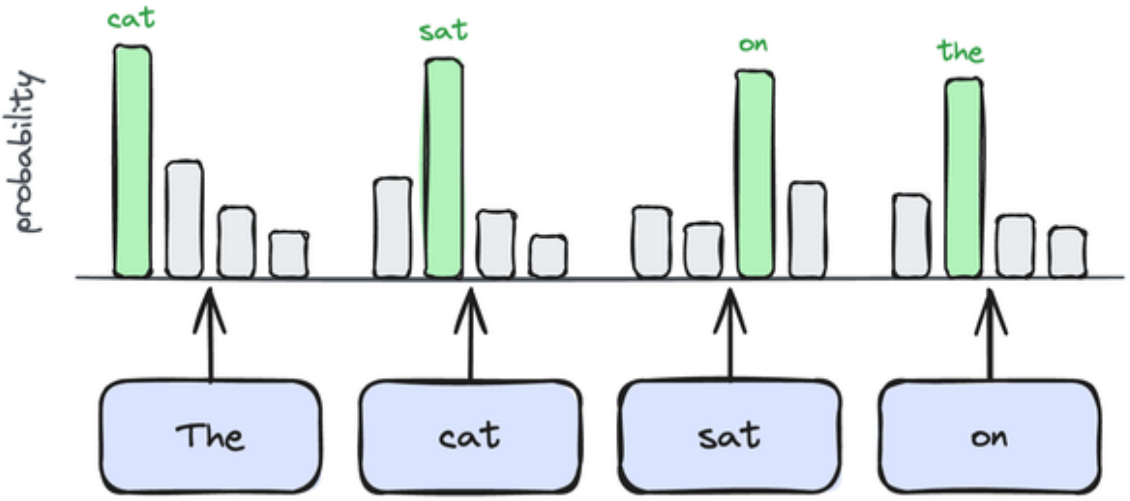


Figure 2-1. Next-token prediction.

Training uses these predictions as the supervision signal. At each position the model has a predicted distribution and a known correct next token. The loss measures how much probability the model put on the wrong tokens. The standard loss is cross-entropy, which is the negative log probability the model assigned to the correct token (Géron, *Hands-On Machine Learning with Scikit-Learn and PyTorch*, O'Reilly, 2026, gives the textbook derivation of cross-entropy and softmax). Minimizing it over the demonstrations is maximum likelihood estimation: it makes the tokens the data actually contains more probable.

When the model is confident and correct, the loss is near zero. When it is confident and wrong, the loss is large.

This is the entire learning signal in SFT. There is no separate notion of a good or bad answer, only the next token the data says should come next and how much probability the model gave it. Pre-training applies this objective to a broad text corpus. SFT applies the same objective to a curated set of demonstrations. The mechanism does not change, only the data.

Because the signal is per token, the model gets one gradient for every token in the completion. Each position is trained on the true tokens before it, not on what the model itself would have generated. This setup is called teacher forcing. A ten-token answer provides ten supervised predictions, and this density is why SFT is efficient: a single example teaches the model at every position at once.

The SFT data format: prompts, completions, masks

An SFT example has two parts: a prompt and a completion. The prompt is the input, such as a user's question. The completion is the output you want, such as the assistant's answer. You want the model to learn the completion given the prompt. You do not want it to learn to generate the prompt, because at inference time the prompt is supplied by the user.

This is where masking comes in. You concatenate the prompt and the completion into one sequence and feed it to the model. Then you tell the loss to ignore the prompt positions and score only the completion positions. The tool for this is a label array that mirrors the input tokens but replaces the prompt tokens with a sentinel value, `-100`, which PyTorch's cross-entropy treats as "skip this position". The completion tokens keep their real values and contribute to the loss.

Figure 2-2 shows the alignment. The input row holds prompt tokens followed by completion tokens. The label row holds `-100` over the prompt and the real token ids over the completion. The model predicts the token to the right of each position, so the labels are read with a shift of one.

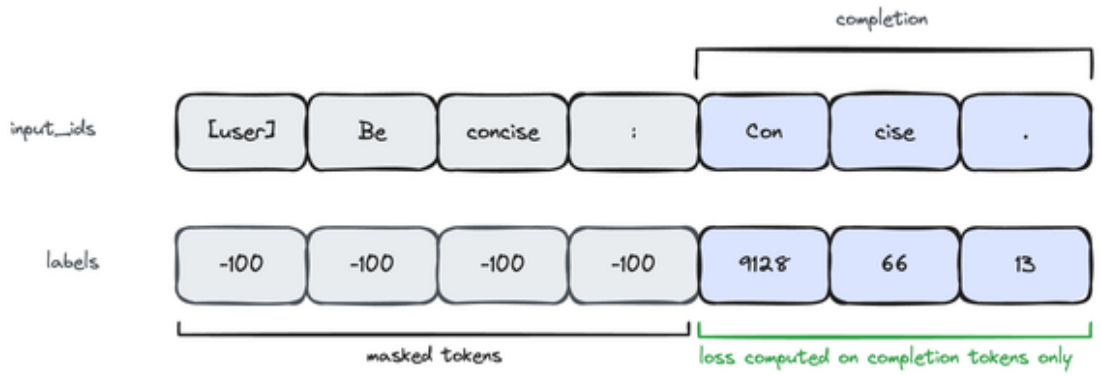


Figure 2-2. The SFT label mask. The prompt is replaced with `-100` so the loss falls only on completion tokens.

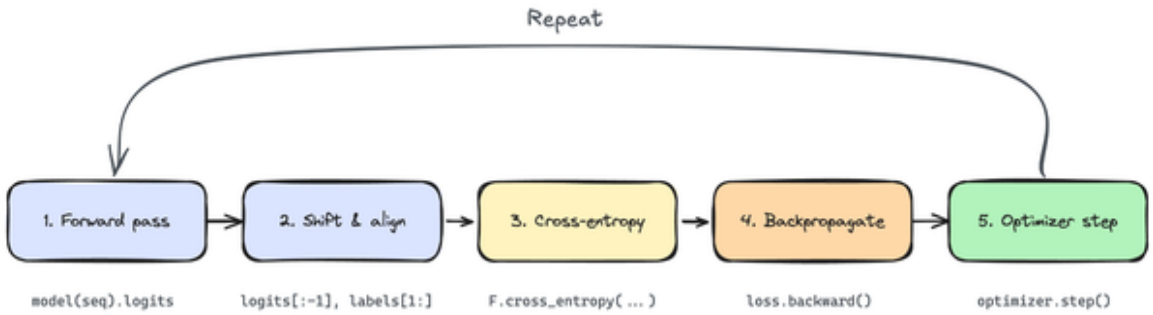
The code applies the template to one short conversation and builds the labels based on an assistant mask.

```
messages = [
    {
        "role": "user",
        "content": "Summarize in one word: always keep answers
concise."},
    {
        "role": "assistant",
        "content": "Concise."
    },
]

enc = tokenizer.apply_chat_template(
    messages,
    return_dict=True,
    return_assistant_tokens_mask=True,
    return_tensors="pt",
).to("cuda")

input_ids = enc["input_ids"]
assistant_masks = torch.tensor(
    enc["assistant_masks"],
    device="cuda"
)

labels = input_ids.clone()
labels[assistant_masks == 0] = -100
```

The `messages` list holds the conversation, one dict per turn. `apply_chat_template` renders it into a single token sequence and inserts the special tokens that separate the user and assistant turns. With `return_dict=True` it also returns the `assistant_masks`, a vector that indicates which tokens belong to the assistant's response. The labels start as a copy of `input_ids`, and the masking line blanks out the non-assistant tokens so cross-entropy skips them.

Implementing the loop in pure PyTorch

The loop has five steps. First, run the sequence through the model to get logits, the raw scores it assigns each candidate next token before softmax turns them into probabilities. Second, shift the logits and labels so each position lines up with the token it should predict. Third, compute cross-entropy on the unmasked positions. Fourth, backpropagate. Finally, step the optimizer.



The logits come out of the model with one distribution per input position. Position `t` predicts the token at position `t+1`, so you drop the last logit and the first label to align them. The `reshape` flattens the batch and time dimensions into one long list of predictions and targets, which is the form cross-entropy expects. The `ignore_index=-100` argument drops the masked positions from the average.

```
import torch.nn.functional as F

optimizer = torch.optim.AdamW(model.parameters(), lr=1e-5)
model.train()
```

```
for step in range(30):
    logits = model(input_ids).logits           # (1, T,
vocab_size)

    shift_logits = logits[:, :-1, :]           # drop last
position
    shift_labels = labels[:, 1:]               # drop first label

    loss = F.cross_entropy(
        shift_logits.reshape(-1, shift_logits.size(-1)),
        shift_labels.reshape(-1),
        ignore_index=-100,                     # skip padding
tokens
    )

    loss.backward()
    optimizer.step()
    optimizer.zero_grad()
```

That is a complete SFT step. `loss.backward()` fills in the gradient of the loss for every weight, and `optimizer.step()` takes one step of gradient descent against it. The optimizer here is `AdamW`: gradient descent with a per-parameter adaptive step size and decoupled weight decay (Géron, *Hands-On Machine Learning with Scikit-Learn and PyTorch*, covers backpropagation and the Adam/AdamW optimizer in detail).

The cross-entropy loss is written out completely so the shift and the mask are not hidden. The gradient of cross-entropy with respect to the logits is the predicted distribution minus the one-hot target, so the error sent back at each position is predicted minus correct.

> **NOTE**
>
> The `transformers` library provides a shortcut: calling `model(input_ids, labels=labels).loss` computes the shifted cross-entropy internally. This chapter writes the loss out by hand so the shift and mask are visible.

What changed in the model by probing before and after

A loss that goes down tells you the optimizer is working. It does not directly show what the model learned. To see the change, probe the model before and after training on two things: the probability it assigns to the demonstration, and the text it generates from the prompt.

The first probe is the demonstration's loss. Before training, compute the cross-entropy of the target sequence under the model, then compute it again after training. The same number the loop minimizes is a direct read on how likely the demonstration is for the model. Training drives it down.

```
@torch.no_grad()
def sequence_loss(model):
    logits = model(input_ids).logits[:, :-1, :]
    return F.cross_entropy(
        logits.reshape(-1, logits.size(-1)),
        labels[:, 1:].reshape(-1),
        ignore_index=-100,
    ).item()
```

The second probe is generation. Build the prompt on its own, then decode it greedily before and after training and compare the text.

```
prompt_ids = tokenizer.apply_chat_template(
    messages[:1],                       # the user turn, with a
generation prompt
    add_generation_prompt=True,
    return_tensors="pt",
).to("cuda")

@torch.no_grad()
def greedy(model):
    out = model.generate(prompt_ids, max_new_tokens=8,
do_sample=False)
    return tokenizer.decode(out[0, prompt_ids.shape[1]:],
skip_special_tokens=True)
```

Run both probes around the loop. Before training, the loss on the demonstration is high and the greedy decode produces something other than the target. After thirty steps, the loss falls toward zero and the greedy decode converges on the target word. The weights moved, and the move is visible in the model's output, not just in the loss curve.

Figure 2-3 suggests one way to present this. A small before-and-after comparison makes the abstract loss number concrete.

Figure 2-3. Probing the model before and after SFT.

The loss the loop minimizes is also a read on the demonstration's probability, and the generated text shifts to match it.

This before-and-after probe is the habit to carry through the book. Whatever you train, measure the model on a fixed input before and after, and look at the output, not only the loss. It is the fastest way to catch a loop that is technically running but learning the wrong thing.

The Minimal RL Loop

SFT needs a completion for every prompt which is written and validated for the model to imitate. Reinforcement learning removes that requirement. Instead of a target completion, you supply a reward function that scores any completion the model produces. The model generates its own candidates, scores them, and shifts toward the ones that score well. This section builds that loop, using the same model and optimizer as before.

Policy, reward, and rollout as three simple functions

Reinforcement learning has a vocabulary that sounds heavier than the code. Three terms carry the loop. The policy is the model: it maps a prompt to a distribution over completions, and it is what you are training. The reward is a function from a completion to a number, where higher means better. The rollout is the act of sampling completions from the policy, the model's attempts at the task. Winder[4] translates these terms and the surrounding RL vocabulary for practitioners.

Figure 2-4 shows how the three connect. The policy produces rollouts. The reward scores them. The scores become an update that changes the policy. Then the cycle repeats with the improved policy.

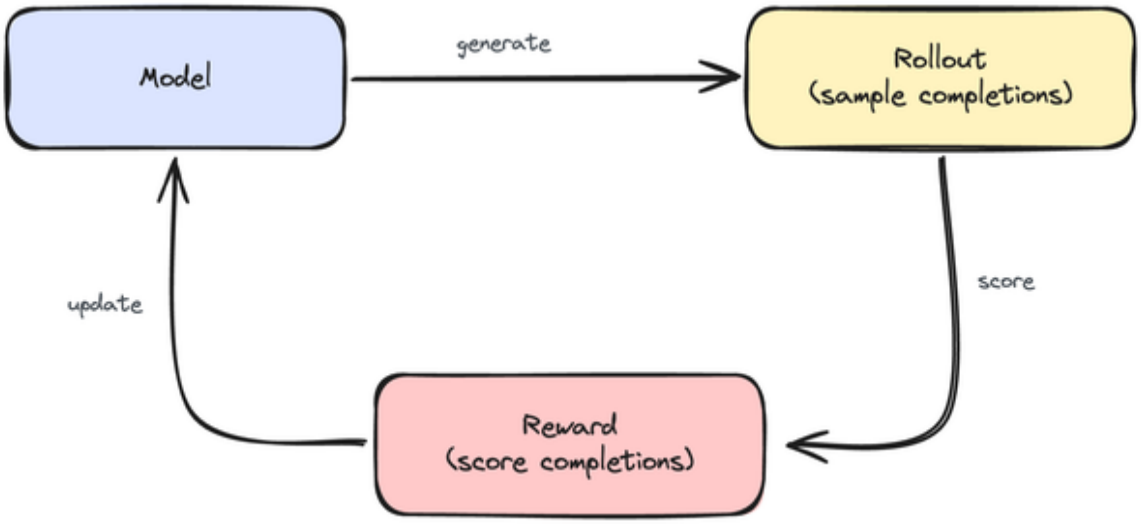


Figure 2-4. The RL loop as three functions.

SFT has a fixed target and asks the model to match it. RL has no target. Instead, there is a score and the model proposes the candidates itself. RL learns from answers the model wrote, weighted by how good they were.

This difference is what lets RL exceed its starting point. SFT cannot teach behavior that is not in the demonstrations. RL can find a high-reward completion the model stumbled into during sampling, then reinforce it. The catch is that the model has to stumble into it at least sometimes, and the reward has to recognize it. Both points return later in the chapter and are a cornerstone of understanding post-training.

A tiny environment: a verifiable text task

RL needs a reward, and a good first reward is one you can check by rule rather than judgment. A verifiable task is one where a program can decide whether the output is correct. Math problems with known answers and code that must pass tests are the canonical examples from Chapter 1. They need no human rater and no second model.

The task here is simpler still: a formatting task. The model must put its answer inside `<answer>` and `</answer>` tags. A short function checks the output and returns a score. This keeps the environment self-contained, so the whole loop runs in one notebook with nothing to install beyond the model.

```
import re

def reward(text):
    score = 0.0
    if "<answer>" in text:
        score += 1.0
    if "</answer>" in text:
        score += 1.0
    if re.search(r"<answer>.+?</answer>", text, re.DOTALL):
        score += 1.0
    return score
```

Table 2-1 lists the four reward values and the kind of output that earns each.

Table 2-1. The scoring system from the formatting reward.

Reward	Example	
0		An empty attempt
0	`42`	An unformatted attempt
1	`<answer>42`	A stray opening tag
3	`<answer>42</answer>`	A well-formed block

The grading matters, and the next subsection explains why: the update needs different completions in a group to receive different scores. A reward that returns the same value for everything teaches nothing.

Implementing policy gradient from scratch

A policy in reinforcement learning is the mapping from a prompt to a distribution over completions. Policy gradient is a family of methods that improve the policy by following the gradient of expected reward, stepping the parameters in the direction that makes higher reward more likely.

The policy gradient increases the probability of completions that scored above average and decreases the probability of those that scored below. This means you need a baseline to compare against. GRPO gets its baseline cheaply: for each prompt it samples a group of completions and uses the group's own mean reward as the baseline. The advantage of a completion is its reward minus the group mean, divided by the group's standard deviation. Completions above the group mean get a positive advantage and are reinforced, while completions below it get a negative advantage and are suppressed. The "Group Relative" in GRPO is exactly this group baseline[5] .

Classic policy-gradient methods like proximal policy optimization (PPO) train a second network that estimates the expected reward of a state to act as the baseline[6]. That network doubles the memory and adds its own training problem. GRPO replaces this with the mean reward of the sampled group, which you already have. The trade-off is that you must sample several completions per prompt to estimate the mean, which costs generation time. For tasks where generation is cheap relative to a second network, that is a good trade, and it is part of why GRPO scaled well for reasoning models.

First, look at GRPO as pseudocode before tackling the real thing. Iterate over the samples, generate completions for a group, score their rewards, turn scores into advantages, then push the policy toward the high-advantage completions.

```
for each step:
    completions = sample G times from policy(prompt)
```

```
    rewards     = [reward(c) for c in completions]

    advantage   = (rewards - mean(rewards)) / (std(rewards) +
eps)

    for each completion c with advantage A:
        logprob = sum of log policy(token | prefix) over c's
tokens
        loss    += -(A * logprob)                          #
reinforce if A > 0
```

The loss `-(A * logprob)` is the whole rule. When the advantage is positive, minimizing the loss raises the completion's log probability. When it is negative, minimizing the loss lowers it. The size of the advantage sets how hard the model pushes. This is the REINFORCE estimator (Williams, "Simple statistical gradient-following algorithms for connectionist reinforcement learning," 1992), the basic policy-gradient update, with a group baseline.

Figure 2-5 illustrates the advantage on one group. Some completions beat the group mean and are pulled up. Others fall short and are pushed down. The mean itself moves nothing.

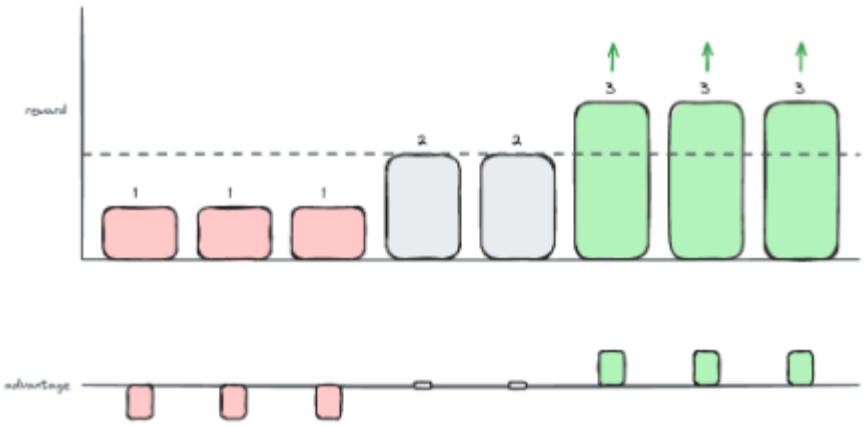


Figure 2-5. Group-relative advantage in GRPO.

The implementation has to handle generation, masking, and log-probabilities over real tensors, but it still follows the same process.

```
import torch
import torch.nn.functional as F

optimizer = torch.optim.AdamW(model.parameters(), lr=1e-5)

prompt = "Name a primary color. Put the answer in answer tags."
prompt_ids = tokenizer.apply_chat_template(
    [{"role": "user", "content": prompt}],
    add_generation_prompt=True,
    return_tensors="pt",
).to("cuda")

G = 8

for step in range(200):
    # rollout G times on a single sample
    with torch.no_grad():
        seq = model.generate(
            prompt_ids.repeat(G, 1),
            do_sample=True, temperature=1.0, top_p=1.0,
            max_new_tokens=64,
pad_token_id=tokenizer.pad_token_id,
        )
    prompt_len = prompt_ids.shape[1]
    completion_ids = seq[:, prompt_len:]
    texts = tokenizer.batch_decode(completion_ids,
skip_special_tokens=True)

    # reward and group-relative advantage
    rewards = torch.tensor([reward(t) for t in texts],
device="cuda")
    advantages = (rewards - rewards.mean()) / (rewards.std() +
1e-4)
```

Two pieces of bookkeeping turn the sampled tokens into a loss.

The first is a completion mask. The generated sequences have different lengths, and the shorter ones are padded. The loss should only count real completion tokens, so you keep tokens up to and including the first end-of-sequence token and drop the rest.

The second is the per-token log-probability. You run the sequence back through the model, line up each position with the token it predicts, and read off the log probability the policy assigned to the token that was sampled.

Below, the completion mask is implemented by finding the first end-of-sequence token with `torch.where` and creating a binary mask tensor. The per-token log probability is implemented by gathering the log probabilities and summing over the mask.

```
    # completion mask: keep tokens up to and including the first
EOS
    is_eos = completion_ids == tokenizer.eos_token_id
    has_eos = is_eos.any(dim=1)
    first_eos = torch.where(
        has_eos,
        is_eos.float().argmax(dim=1) + 1,
        torch.full_like(has_eos, completion_ids.size(1),
dtype=torch.long),
    )
    idx = torch.arange(completion_ids.size(1), device="cuda")
    comp_mask = (idx.unsqueeze(0) <
first_eos.unsqueeze(1)).float()

    # per-token log-probs of the completion under the current
policy
    logits = model(seq).logits[:, prompt_len - 1 : -1, :]
    logp = F.log_softmax(logits, dim=-1)
    token_logp = logp.gather(-1,
completion_ids.unsqueeze(-1)).squeeze(-1)

    # policy-gradient loss with the group baseline
    seq_logp = (token_logp * comp_mask).sum(dim=1)
    loss = -(advantages * seq_logp).mean()

    loss.backward()
    optimizer.step()
    optimizer.zero_grad()
```

Run it and the mean reward climbs over a few hundred steps as the model learns to emit the tags. The early steps depend on chance: the base model has to produce at least one well-formed completion in a group before there is anything to reinforce. Once it does, the advantage points the policy toward that behavior and the reward compounds.

If every completion in a group earns the same reward, the advantage is zero for all of them and the step changes nothing. This is the most common

reason a GRPO run stalls: You need variance within the group. A reward that is too easy (everything scores high) or too hard (nothing scores) gives you none. Graded rewards and a high enough sampling temperature both help keep variance alive.

The minimal loss above takes one gradient step per rollout. The example script reuses each rollout for several steps, which is cheaper because generation is the slow part. To do that safely, it borrows the clipped surrogate from PPO[7] . It records the log-probabilities at sampling time, then bounds how far the ratio between the new and old probabilities can move on each reuse.

```
coef_1 = torch.exp(per_token_logps - old_per_token_logps)
coef_2 = torch.clamp(coef_1, 1 - epsilon, 1 + epsilon)
per_token_loss = -torch.min(coef_1 * advantages, coef_2 *
advantages)
loss = (per_token_loss * completion_mask).sum() /
completion_mask.sum()
```

When you take a single step per rollout, the ratio is exactly one and this reduces to the plain policy gradient above. The clip is only worthwhile when you reuse a rollout because the group-advantage computation is identical at a single step.

```
mean = rewards.view(-1,
num_generations).mean(1).repeat_interleave(num_generations)
std  = rewards.view(-1,
num_generations).std(1).repeat_interleave(num_generations)
advantages = (rewards - mean) / (std + 1e-4)
```

Comparing the updates: SFT vs RL on the same task

Run both methods on the formatting task and the difference in their updates becomes concrete. With SFT, you write a correct example, `<answer>red</answer>`, and train the model to reproduce it token for token. The gradient pushes the model's distribution toward that exact string. With RL, you write no example. The model samples its own attempts, the

reward scores them, and the gradient pushes toward whichever attempts scored above the group mean.

The two gradients have the same shape but a different source. SFT minimizes `-logprob(target)` for a target you supplied. RL minimizes `-(advantage * logprob(sample))` for samples the model supplied. Set the advantage aside and the RL update is an SFT update toward the model's own output. The advantage is what reweights those self-generated targets by quality. Figure 2-6 places the two side by side.

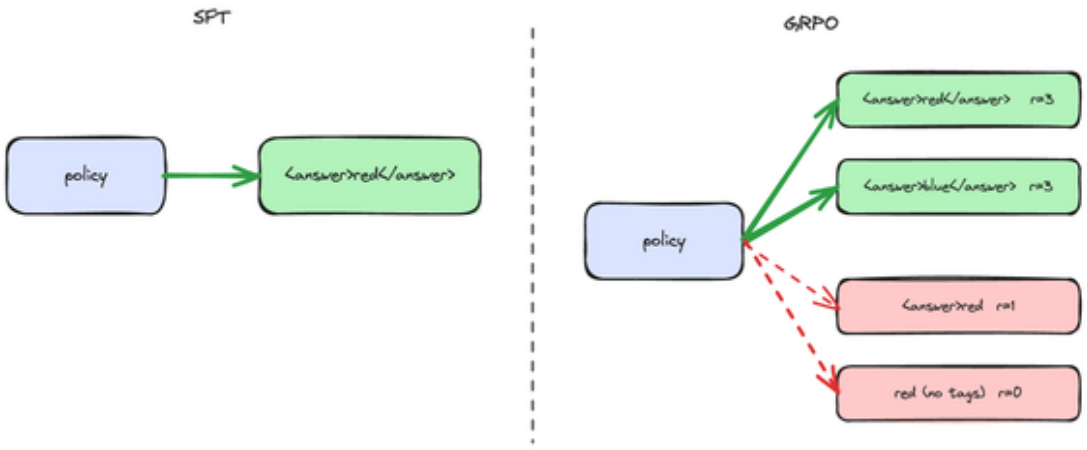


Figure 2-6. The same task in SFT and GRPO.

The practical consequences follow from the source. SFT is stable and sample-efficient: every token is a clean gradient, and one pass over good data moves the model reliably. But it needs that good data, and it cannot teach what the demonstrations do not contain. RL needs only a verifier, not demonstrations, and it can discover behavior no one demonstrated. But it is noisier, it depends on the model sampling something worth reinforcing, and it can collapse if the reward has no variance or rewards the wrong thing.

On a formatting task either method works, and SFT is the easier choice. The picture changes when correct answers are hard to write but easy to check, which is the case the next section builds into a general rule.

Mental Model

You have now run both the SFT and RL loops and seen their updates. This section turns that experience into a mental model you can carry to harder problems. The three subsections frame the core contrast, the asymmetry in what each method costs, and a rule for choosing between them.

Imitation vs optimization

SFT is imitation. It shows the model what to do and trains it to copy. The ceiling is the data. A model trained by SFT can become as good as its demonstrations and no better, because the only signal it ever sees is "produce this token here." If the demonstrations are mediocre, the model learns mediocrity faithfully. If they are excellent, the model approaches them but does not surpass them, because nothing in the objective rewards doing better than the examples.

RL is optimization. It does not show the model what to do. It scores what the model already does and pushes toward the higher scores. The ceiling is the reward and the model's own reach during sampling. RL can find a strategy that appears in no demonstration, as long as the model samples it occasionally and the reward recognizes it. This is how DeepSeek-R1-Zero developed chain-of-thought reasoning from reinforcement alone[8], without being shown a single worked example. No demonstration told it to reason step by step. The reward favored correct answers, and longer reasoning produced more correct answers.

The flip side is that optimization finds whatever the reward measures, including unintended shortcuts. If the reward can be satisfied without doing the intended task, the model will find that path. Imitation does not have this failure mode, because it copies behavior rather than chasing a score. The two methods fail in opposite ways: imitation is capped by its data, and optimization is only as honest as its reward. Chapter 6 returns to reward hacking and its mitigations.

Data is cheap, reward is expensive

The two methods move their cost to different places, and seeing where helps you budget a project. SFT spends its cost on data. You need prompts paired with good completions, and the completions have to be written or collected. For many tasks this is tractable: you can mine existing logs, hire annotators, or generate candidates with a stronger model and filter them, as the approach Chapter 1 described in the Llama 3 recipe.

RL inverts this. The data is just prompts, which are cheap, and often you reuse the same prompts many times across rollouts. The cost moves to the reward. A reward that captures what you actually want, resists shortcuts, and returns useful variance is hard to build. On verifiable tasks you get the reward almost for free, because a checker already exists: a math answer is right or wrong, a test passes or fails. Off verifiable tasks you have to construct the reward, often by training a reward model on human preferences, which is a data and modeling problem of its own.

This asymmetry explains much of the field's shape. Verifiable domains like math and code saw the fastest RL progress because the expensive part, the reward, came for free. Open-ended domains like helpful dialogue need preference data and reward models. When you plan a post-training project, ask first where your signal comes from.

Where each technique shines

Neither method is the better one in general. Instead, they solve different problems, and most real pipelines use both. SFT is the right tool when you can demonstrate the behavior: a format to follow, a style to adopt, a tool-call syntax to emit, a cold start that gives the model a competent baseline before any reinforcement. It is fast, stable, and predictable, and it is almost always where a pipeline begins.

RL is the right tool when you can score the behavior but not easily demonstrate the best version of it. Mathematical reasoning, code that must pass tests, and multi-step agentic tasks all fit: the optimal path is hard to write by hand, but success is easy to check. RL is also the tool for pushing

past the ceiling SFT imposes, because it optimizes against an outcome rather than copying an example. Table 2-2 summarizes the contrast.

Table 2-2. When to reach for SFT and when to reach for RL.

Dimension	SFT	GRPO
Training signal	A target completion you provide	A reward score over the model's own samples
Data	Prompts with good completions	Prompts
Manual cost	Collecting quality completions	Designing a reward that resists shortcuts
Ceiling	The quality of the demonstrations	The reward and the model's sampling reach
Regression	Inheriting the data's limits	Exploiting flaws in the reward
Best for	Format, style, tool syntax, cold start	Math, code, agentic tasks, verifiable outcomes

The DeepSeek-R1 recipe from Chapter 1 is the canonical combination: SFT first, to give the model a competent starting point and a sane output format, and RL second, to optimize reasoning against verifiable rewards. SFT supplies the floor, while RL raises the ceiling. You now have working versions of both halves.The next chapter puts the SFT half to work, starting with where its data comes from.

Summary

This chapter built post-training end-to-end in miniature. You set up a single model on a single GPU, then implemented the two loops that the rest of the book expands. The SFT loop is next-token prediction on a templated conversation, with padding masked so the loss falls on every real token. The RL loop samples a group of completions, scores them against a reward, and pushes the policy toward the ones that beat the group average. You probed a model before and after SFT to see the weights move, and you watched a GRPO reward climb on a verifiable task.

A solid mental model of post-training techniques is the goal of this chapter and the part to carry forward. SFT imitates and is capped by its data. RL optimizes and is capped by its reward and the model's reach. Data is the cost of imitation, and a good reward is the cost of optimization. Pipelines combine the two: SFT for the floor, RL for the ceiling.

Everything in this chapter was deliberately minimal. Stripping out scaling, logging, and tooling is what lets each loop fit in under 100 lines and run in seconds, and a fast loop is what makes the change in the model easy to observe. The next chapter slows down and returns to the first loop. It looks at where SFT data comes from, how to build and clean it, and how chat templates and loss masking shape what the model learns. The loop stays the same. The data and infrastructure around it are where the real work begins.

References

- Alammar, J., and Grootendorst, M. *Hands-On Large Language Models*. O'Reilly, 2024.
- DeepSeek-AI. "DeepSeek-R1: Incentivizing Reasoning Capability in LLMs via Reinforcement Learning." 2025. *https://arxiv.org/abs/2501.12948*.
- Géron, A. *Hands-On Machine Learning with Scikit-Learn and PyTorch*. O'Reilly, 2026.
- Pointer, I. *Programming PyTorch for Deep Learning*. O'Reilly, 2019.

- Schulman, J., Wolski, F., Dhariwal, P., Radford, A., and Klimov, O. "Proximal Policy Optimization Algorithms." 2017. *https://arxiv.org/abs/1707.06347*.
- Shao, Z., et al. "DeepSeekMath: Pushing the Limits of Mathematical Reasoning in Open Language Models." 2024. *https://arxiv.org/abs/2402.03300*.
- Tunstall, L., von Werra, L., and Wolf, T. *Natural Language Processing with Transformers*, Revised Edition. O'Reilly, 2022.
- Williams, R. J. "Simple Statistical Gradient-Following Algorithms for Connectionist Reinforcement Learning." *Machine Learning* 8 (1992): 229–256.
- Winder, P. Reinforcement Learning: Industrial Applications of Intelligent Agents. O'Reilly, 2020.

---

1 For a thorough treatment, see Pointer, *Programming PyTorch for Deep Learning*, O'Reilly, 2019.

2 Alammar and Grootendorst, *Hands-On Large Language Models*, 2024, O'Reilly.

3 Alammar and Grootendorst, *Hands-On Large Language Models*, 2024, O'Reilly.

4 Winder. *Reinforcement Learning: Industrial Applications of Intelligent Agents*, O'Reilly, 2020.

5 Shao et al., "DeepSeekMath," 2024; DeepSeek-AI, "DeepSeek-R1," 2025.

6 Schulman et al., "Proximal Policy Optimization Algorithms," 2017; Winder, *Reinforcement Learning*, covers PPO and actor-critic methods)

7 Schulman et al., 2017

8 DeepSeek-AI, "DeepSeek-R1," 2025

About the Author

Ben Burtenshaw is a Machine Learning engineer at Hugging Face working on open source tooling for post-training, reinforcement learning, and evaluation. His work sits at the intersection of machine learning engineering and practical education for the community. He has helped build and teach the ecosystem around tools like TRL (Transformers Reinforcement Learning), OpenEnv, and the Hugging Face Hub, and regularly turns frontier training workflows into hands-on material the broader community can use.