baseten

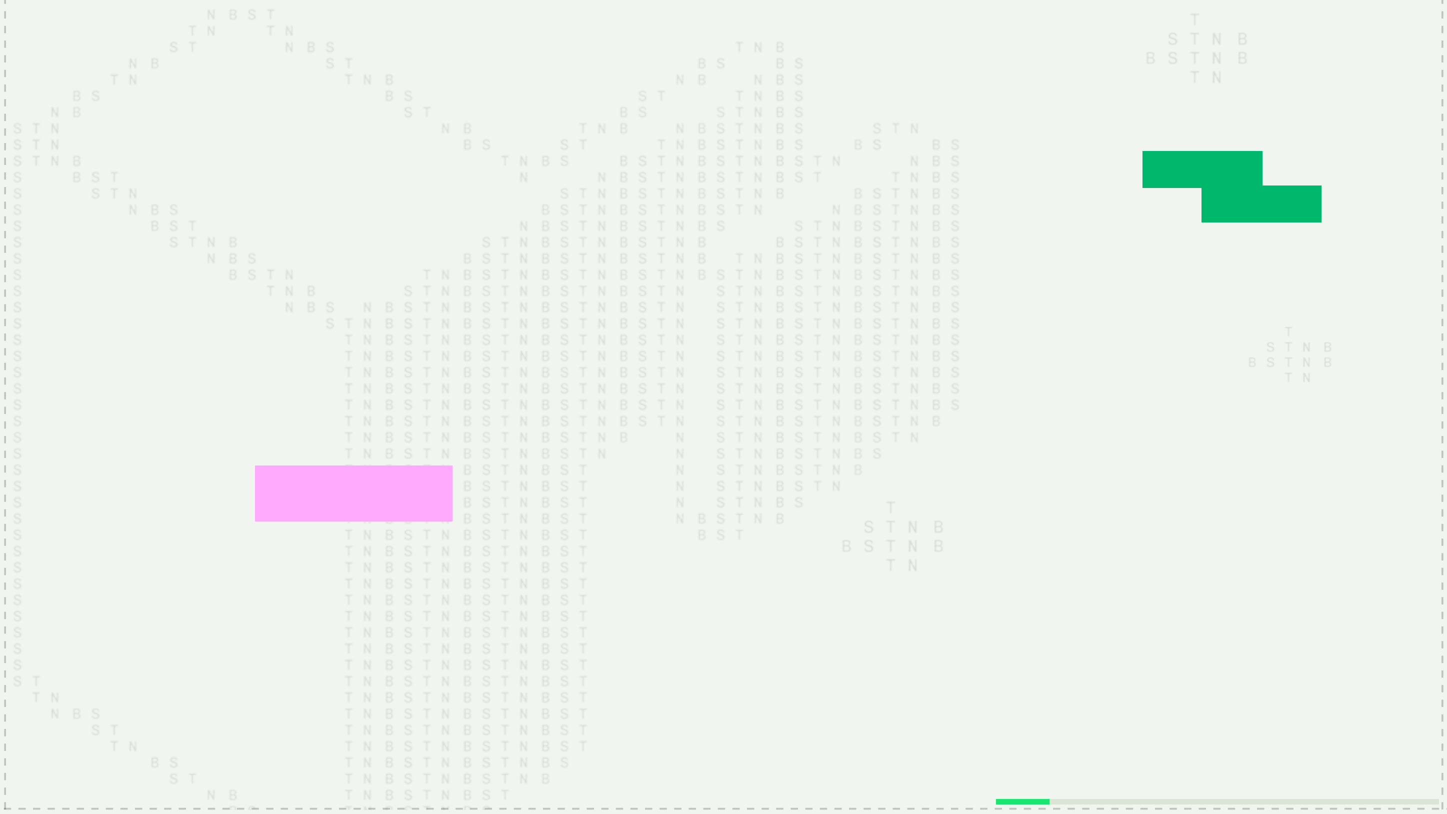
When to use RL vs. SFT IRL

Why reinforcement learning matters (Pt. 2)

When to use RL vs. SFT IRL

It’s not as if RL is some free upgrade you sprinkle on top of SFT and call it a day. RL comes with very real tradeoffs, both statistical and engineering. At a high level, SFT teaches distributions while RL teaches strategies. SFT is doing behavioral cloning: “given this input, imitate this pattern of outputs as closely as possible.” RL, in contrast, is optimizing for returns: “over the course of an entire trajectory, push the model toward sequences that score well under some reward.” That distinction sounds elegant in theory. In practice, it means SFT is incredibly data-efficient and stable, while RL is often the opposite.

One way to see this is through the lens of information flow. In RL, each generation (or rollout, often a small batch of candidate outputs for a single input) gets a scalar reward at the end. You produce a 8192-token answer, you run your verifier, and you get a single number back. In the limiting case, that’s basically one bit of information per rollout: “good” or “bad.” In fact, this is the maximum amount of information you receive, and only occurs when you get the rollout “right” 50% of the time, and “wrong” 50% of the time. By contrast, SFT gets about n bits of information per sequence, where n is the number of tokens. After every prediction, we tell the model exactly what the next token should have been. The gradient touches every step.



CONTINUED >

When to use RL vs. SFT IRL

So when you choose RL, you're deliberately squeezing a very rich space of behavior through an extremely narrow feedback channel. You are, as Andrej Karparthy says, sucking supervision through a straw. That doesn't mean RL is wrong, but it does mean you're throwing away a lot of information you could have used. Sometimes you don't have a choice: you don't have enough high-quality demonstrations to fine-tune on, so you cling to whatever signal you can get. That's the regime where RL makes sense. But in many cases, people reach for RL not because they lack signal, but because they're discarding most of the signal they already have.

Take something as simple as code execution. A naïve RL setup says: run the unit tests, give +1 if they pass, 0 if they fail, backprop through that. But a failing test doesn't just say "no"; it comes bundled with a stack trace, an exception type, sometimes a failing line number, logging, and so on. There is a huge amount of structure in that failure. If all you do is map "all of that" → "single scalar," you've lobotomized your own supervision.

Alternative Methods

This is where methods like iterative SFT and RGT come in: instead of treating the verifier as a black-box reward oracle, you use it to generate better training examples. The model proposes candidates, your verifier filters them or annotates them, and you fold the surviving or enriched outputs back into SFT. You've taken what looks like an RL problem and turned it back into a supervised learning problem, but with much sharper data.

There are other variations on this same idea. Process reward models don't just say "this final answer is good," they try to assign signal at intermediate steps: is this chain-of-thought step coherent, is this lemma usage valid, does this search action move you closer or further from the goal? Instead of a single reward at the end of the trajectory, you're sprinkling feedback along the path. In spirit, this is still moving you toward "RL," but the key move is the same: make the supervision denser and more structured, rather than resigning yourself to an impoverished scalar at the end.

Hidden Engineering Costs of RL

Even when you do want “real” RL, you pay a large engineering tax. A basic requirement is that the model doing rollouts for inference is logically separate from the model you’re updating with gradients. That usually means different GPU pools, different serving stacks, and a lot of shuttling data and weights back and forth: prompts and sampled tokens flow from the inference GPUs to the trainer, logprobs and value estimates may need to be logged, gradients get computed on another set of devices, and periodically you have to ship updated weights back to the inference side.

RL also has a lot of instability problems. People have shown that slight numerical imprecision in different inference engines can cause model training to blow up. They’ve had to come up with a lot of tricks to manage the various sources of “off-policiness”, such as truncated importance sampling, which introduce additional hyperparameters and complexity. RL has famously suffered a whole host of other problems related to hyperparameters. We often constrain the updates in some way so that the gradient doesn’t change the base model too much (either through KL divergence constraints, or clipping, or the like), but this leaves a sensitive goldilocks zone: control too much, and the model will cease to explore and entropy will collapse; control too little, and the model can drift significantly from the base-model and start to get into some really nonsensical areas.

Why Use RL

So if RL is noisy, sample-inefficient, and infrastructure-heavy, why does anyone do it? For the frontier labs, the reason is obvious: they've more or less saturated the space of clean supervised data in domains like code and math. There simply aren't that many untouched examples of human-written proofs or gold-standard code left to scrape. However, there is an effectively unbounded stream of problems they can generate and automatically verify. Verification is much easier than generation. They can synthesize endless math problems, run checkers, and train models to optimize solve rate. They can generate code tasks and reward models can tell them which solutions are correct. In that regime, optimizing returns with RL makes sense; it's the only way to squeeze more value out of a domain where new demonstrations are scarce but verifiable tasks are abundant.

If you're fine-tuning your own model, the translation of this is straightforward: y**ou should only reach for RL when you don't have good supervised examples to train on, but you do have a reliable learning signal that can tell you whether a given sample is good or not.** A classic example is training a model to use a search tool. It's very hard to write down the "correct" query for every possible user question ahead of time. But it's easy to synthesize questions sourced from a given document and reward the model if its retrieval hits that document. The policy space is messy; the verification is cheap. That's an RL-shaped problem.

However, once you admit "I have a verification signal," you're not confined to RL. You can generate a bunch of candidates with a strong model (even a closed model like GPT-5.1 or Gemini-class models), filter those candidates with your signal, and then train on the survivors with standard SFT. One step further is using the evaluation signal to refine the initial generation itself. Iterative SFT and RGT push this further, where instead of just accepting or rejecting candidates, it tries to make every sample useful by turning verifier feedback into richer annotations or corrections, then distilling that back into SFT. In other words, you can often get most of the benefits people ascribe to RL by just being smarter about the data and supervision pipeline.

CONTINUED ›

Why Use RL

This still leaves the fact that vanilla SFT is not some silver bullet either, no matter how good your examples are. There are two core problems. First, traditional SFT is, in a precise sense, off-policy. During training, we use teacher forcing: we always feed the model the ground-truth previous tokens from the dataset and ask it to predict the next one. At inference time, though, the model conditions on its own previous outputs. That means it's now seeing prefixes that are slightly (and eventually quite) different from those in the training data. The distribution of states the model experiences at test time drifts away from the distribution it was trained on. That's exactly what "off-policy" means: you're learning a policy under one state distribution and deploying it under another.

Second, even within that framework, the feedback in SFT is very sparse in token space. We act as if there is exactly one correct token at each position and push the model's probability mass onto that token. In reality, there is usually a whole set of tokens that would be acceptable, often with a natural ordering of preference. SFT doesn't capture that; it just says "make this one as likely as possible and everything else as unlikely as possible." You're discarding information about the shape of a good distribution over tokens.

Distillation

That’s where distillation enters. In distillation, a smaller student model tries to approximate the full logit distribution of a larger teacher, rather than just its argmax. Instead of only seeing “the correct next token was ‘therefore’,” the student learns that the teacher thought ‘thus’, ‘so’, ‘hence’ were all moderately plausible, and ‘banana’ was not. You get a much richer supervisory signal, because you’re trying to match an entire distribution, not a single point. On-policy distillation takes this idea a step further: you let the student generate a rollout, then query the teacher for logits over each position in that student-generated trajectory. This is an extremely efficient way to train the student, because every token in its own behaviour is getting shaped by a teacher distribution. It’s “distribution matching” but now on the states the student actually visits, not just on curated dataset prefixes.

Of course, standard distillation and on-policy distillation usually assume you have white-box access to the teacher: you can get logits, and ideally you share a tokenizer or at least can align tokenizations. Recently, people have started to push into black-box on-policy distillation: a kind of GAN-style setup where the student learns from a proprietary, API-only model. You no longer get the logits, but you can still extract relative preferences over sequences or detect when the teacher “prefers” one candidate over another. It’s a way of turning an opaque frontier model into a shaping signal for your own model, without full RL and without ever seeing the teacher’s internals.

Recap

Zooming out, SFT and RL are not competing ideologies; using both is the standard paradigm in modern LLM training. SFT teaches distributions: how to follow instructions, imitate patterns, and stay within the support of sensible text. RL teaches strategies: how to choose sequences that score well under some long-horizon objective. RLHF, GRPO, and their cousins are all just different attempts to push models toward human preferences or task-specific metrics that are hard to express as clean demonstrations. For big frontier models, that's often about optimizing for multidimensional notions like helpfulness, harmlessness, and honesty, where correctness is messy and context-dependent.

But here's the part people don't like to say out loud: in most practical settings, RL is the wrong starting point. If you have the ability to collect in-distribution, high-quality demonstrations, you should squeeze SFT, distillation, and iterative SFT-style pipelines for everything they're worth before you even contemplate spinning up a full RL stack. Most teams that say they "need RL" are actually staring at a data problem: their demonstrations are weak, their reward signals are underutilized, and their evaluation is vibes-driven rather than verifiable. RL will not fix that; it will just make it harder to debug.

Takeaway

Our strong opinion is that for 90% of teams, the shortest path to delighting users is not a shiny RL pipeline; it's boring, high-quality, in-distribution data, a verifier that actually encodes what "good" means, and a training loop that treats every scrap of signal as precious. SFT should be your default, distillation and iterative SFT your power tools, and RL you only pull out when you know you're up against a ceiling those methods cannot reach.