Distilling 100B+ Models 40x Faster with TRL

On-policy distillation with large teacher models, efficient rollout generation, and optimized teacher server setup

AUTHORS

Carlos Miguel Patiño, Kashif Rasul, Edward Beeching, Lewis Tunstall

AFFILIATION

Hugging Face

PUBLISHED

Apr. 12, 2026

Table of Contents

TRL now supports on-policy distillation with 100B+ parameter teacher models and trains up to 40x faster thanks to three key features we've implemented in the latest release:

- Utilizing a generation buffer to take advantage of vLLM's batching capabilities for generation.
- Batching requests when sending sequences to the external vLLM server that hosts the teacher model.
- Encoding logprobs in binary to reduce the transmission payload between the teacher server and student client.

This means you can now distill models in the Qwen3.5 or Gemma4 families across any scale. Here's a snippet with how to do it:

DistillationTrainer Example

A common scenario with exciting releases on the Hub is that we see a new model with amazing performance in a benchmark we're interested in. We then use hf-mem to estimate the resources required to run inference with the model and find out that you need more than 400GB of memory. Regardless of your level of GPU wealth, 400GB of memory is something difficult and expensive to spin up if you are going to use the model frequently.

The good news is that we can extract the capabilities of that large model and transfer them into a smaller model that we can fit in 8GB of memory. The process of extracting those capabilities is called distillation, and we'll use it to extract capabilities from a model that requires 400GB of memory (`Qwen/Qwen3-235B-A22B-Instruct-2507`) to one that's 50 times smaller (`Qwen/Qwen3-4B`). Let's see how distillation works and how we can use it to extract knowledge from a teacher model.

The Distillation Setup

The goal of distillation is to transfer knowledge or behavior from a powerful teacher model to a smaller student model. One common approach is to have the teacher generate answers for a set of prompts and then train the student to match either the generated tokens (hard targets) or the teacher's output distribution, such as its log-probabilities (soft targets). This is often called off-policy distillation, because the student is trained on data generated by another model's policy rather than its own.

Off-policy distillation works well, but we can usually do better by letting the student learn from its own mistakes instead of just imitating data generated by another model. To do that, we first let the student generate rollouts and keep track of its log-probabilities at each training step. Then we ask the teacher to generate the log-probabilities for those exact same rollouts and measure how different the student's log-probabilities are from the teacher's. We call this on-policy distillation (Agarwal et al., 2023) because we are generating tokens using the model we are training. In practice, this on-policy setup helps the student recover from bad moves and learn from them, instead of only imitating the teacher on trajectories the student might never produce on its own.

You can check the diagram below to see the main differences between the off- and on-policy distillation.

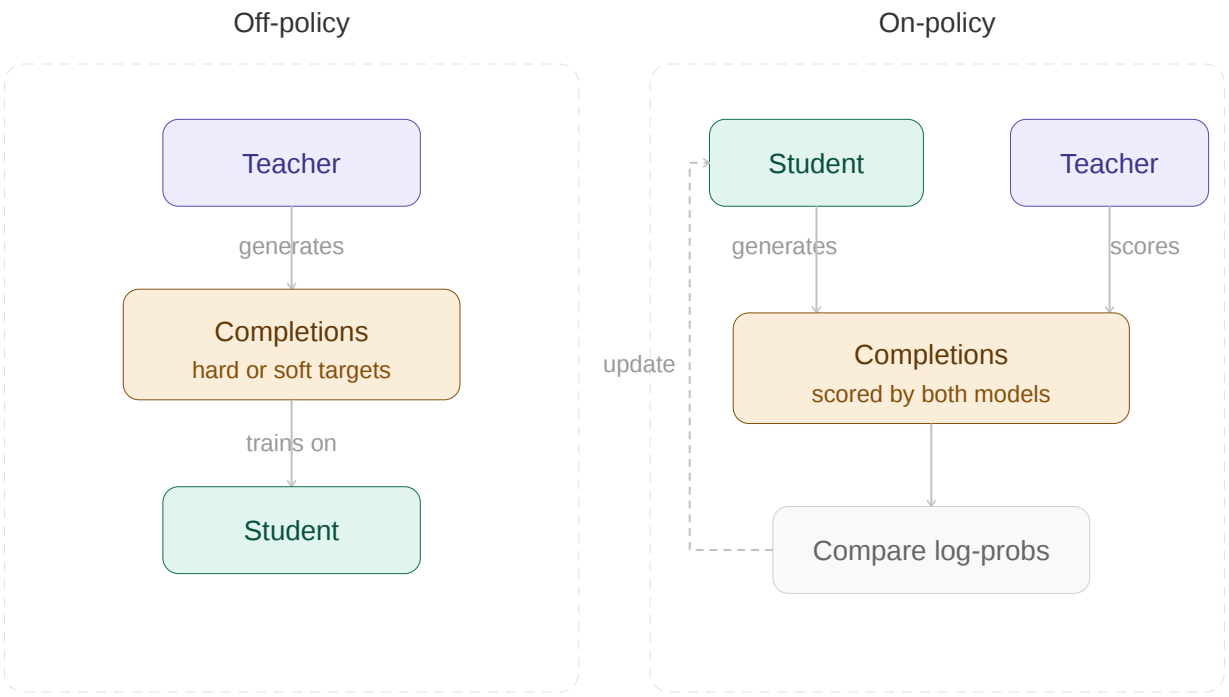


Figure 1 · Comparison between off- and on-policy distillation.

Off- and on-policy describe where the training data comes from, but we can also classify distillation methods based on how we compare the teacher and student distributions. In both cases, the comparison is usually done with the Kullback-Leibler (KL) divergence. That comparison can use either forward KL, which treats the teacher as the reference distribution, or reverse KL, which instead focuses on the student's log-probabilities. This image below shows how we calculate each type of divergence and highlights the differences in what each loss penalizes.

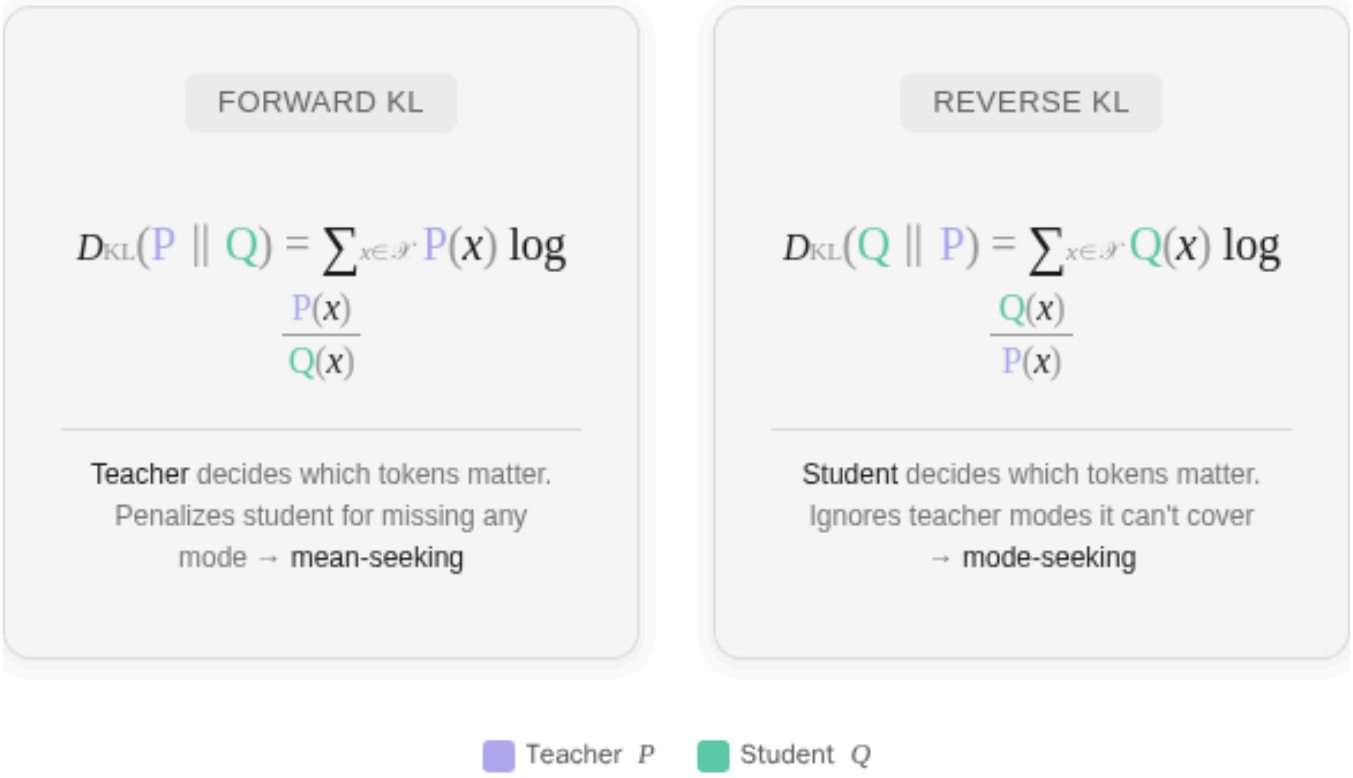


Figure 2 · Graphical explanation of the main differences between forward and reverse KL.

Both forward and reverse KL compare the teacher and student distributions, and they give us a way to measure how different the two models are at each step of the completion. The catch is that computing KL over the full vocabulary is expensive. For every generated token, we would need the log-probability for every token in the vocabulary at that position. For a 1k-token sequence with a Qwen model, that means storing roughly 1k x 150k = 150 million log-probabilities for a single sample. That is why, in practice, people usually approximate the KL using only the top-k log-probabilities at each step.

Aside from the theoretical differences between forward and reverse KL, which are well explained elsewhere (Jones, 2023; Ko et al., 2025), they also lead to different engineering trade-offs when you use a top-k approximation. In forward KL, you take the token IDs with the highest log-probabilities under the teacher and compare the student against the teacher on those same IDs. In reverse KL, you instead use the token IDs with the highest log-probabilities under the student. The diagram below illustrates the difference between these two choices using the simple case where we keep only the top-1 log-probability to compute the KL loss. Despite using the same models and the same completion sequence, the forward and reverse KL will have different results.

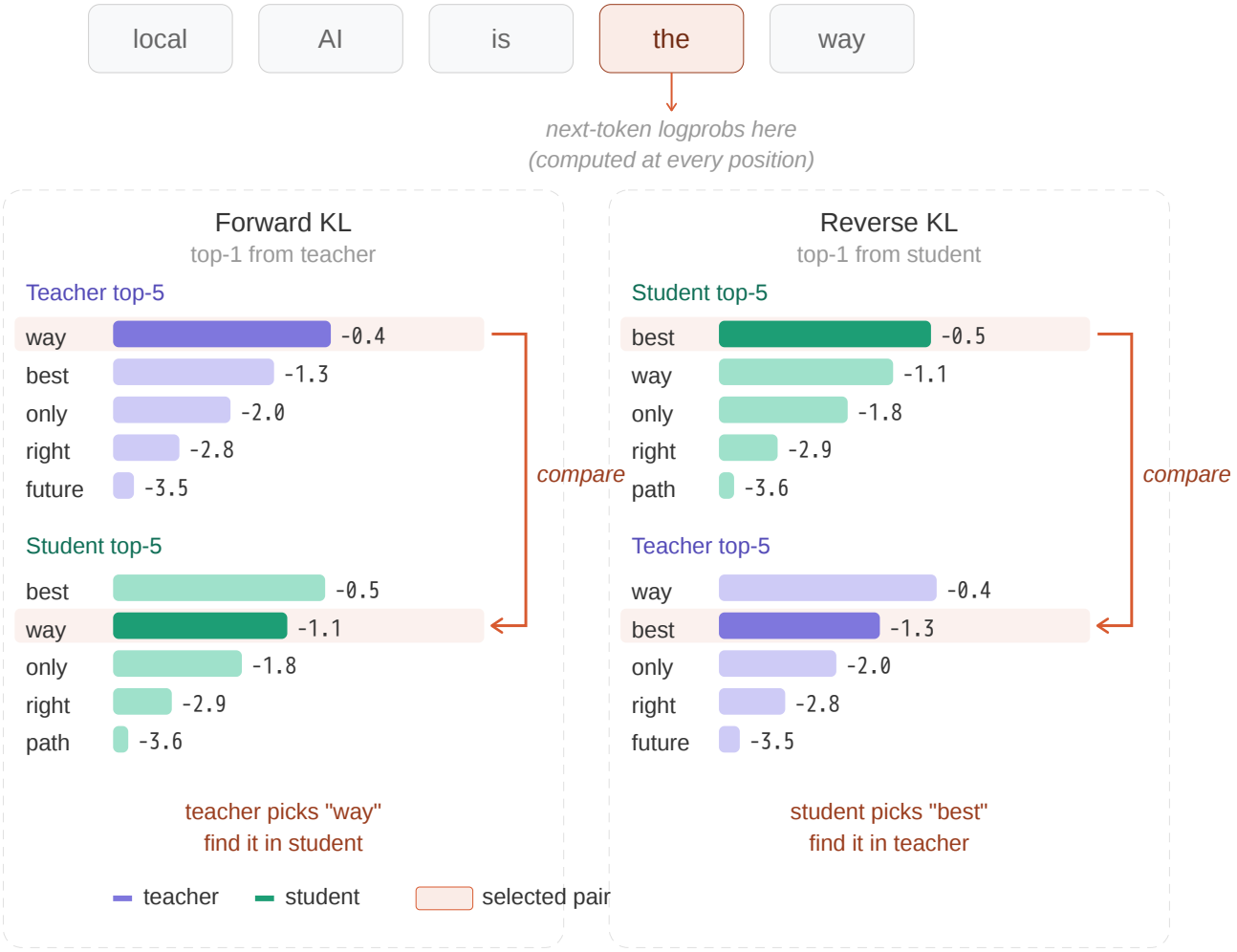


Figure 3 · Diagram showing the differences between the forward and reverse KL when using only the top-1 logprob.

Our GOLD work (Hugging Face H4, 2024) and Thinking Machines (Thinking Machines, 2024) showed how on-policy distillation performs better than off-policy distillation, so we'll focus the conversation on what is necessary to implement that kind of distillation at scale.

Engineering Challenges of Distillation

On-policy distillation looks simple at a high level: let the student generate rollouts, ask the teacher to score them, and train the student to close the gap. In practice, though, making that loop efficient is a pretty tricky systems problem. We started exploring on-policy distillation as a way to distill between any teacher and student with our work on GOLD. In the process, we found that on-policy distillation introduced a few engineering bottlenecks that made training surprisingly slow, even for relatively small models. In particular:

- Student rollout generation: training required small microbatches, so we were not benefiting from the batching capabilities of inference engines.
- Teacher logprob extraction: inference engines like vLLM and SGLang are optimized for generation, not for serving the token-level log-probabilities needed for distillation.
- Logprob transfer: once the teacher was moved to an external server, sending those logprobs back efficiently became a bottleneck of its own.

The next sections will walk you through the main bottlenecks we ran into and the changes that made the biggest difference.

Using a Generation Buffer Without Breaking On-Policy Training

Although we were using vLLM to generate student rollouts, we were not getting the full benefit of its batching engine. GPU memory was already heavily used by the activations needed for backpropagation, so in practice we had to keep `per_device_batch_size=1`. That meant prompts were sent to the generator one at a time at every micro-step, even when using gradient accumulation.

The main optimization was to decouple the training microbatch size from the generation batch size. To do that, we accumulate prompts in a buffer over a window matching the number of gradient accumulation steps. For example, with 64 gradient accumulation steps, we collected 64 prompts and generated all of their rollouts in one call instead of making 64 separate single-prompt calls. This let us recover the batching efficiency of the inference engine without increasing the training microbatch size.

You can use the animation below to try different settings and see how the buffer speeds up generation by taking advantage of the inference engine's batching capabilities. You can notice that the forward and backward steps are still done sequentially from the elements in the buffer, but we leverage parallelization in the generation phase.

Completion Buffer: Why Batched Generation Speeds Up Training

Comparing sequential vs. buffered completion generation across gradient accumulation steps

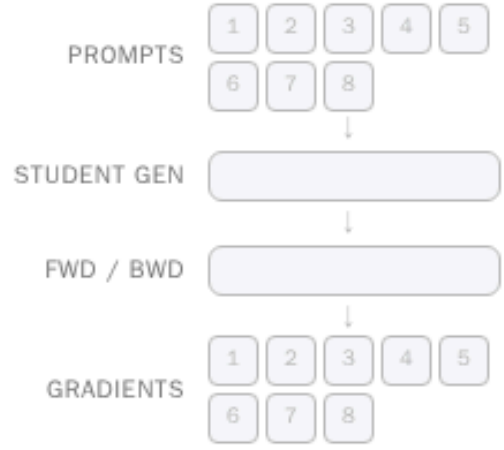
Generating (sequential) Generating (batched) Buffered completion Forward / Backward Done

Start Animation Reset Grad Accum: 8 Speed: 1x

Without Completion Buffer

—

Generate one completion → train → repeat sequentially



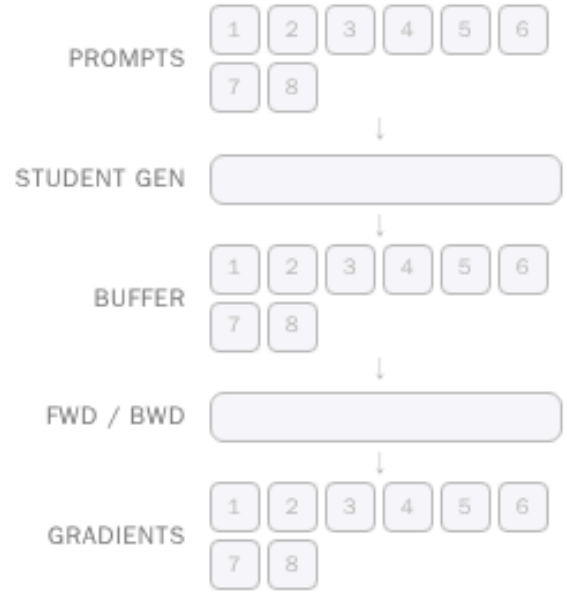
PER-STEP TIMELINE (GENERATION + TRAINING)

Generation	Training	Total
—	—	—

With Completion Buffer

—

Generate all completions (batched) → buffer → train sequentially



PER-STEP TIMELINE (GENERATION + TRAINING)

Generation	Training	Total	Speedup
—	—	—	—

Figure 4 · Interactive animation showing generation buffer batching vs sequential generation.

The nice part is that the buffer does not break the on-policy setup. Because the student weights stay fixed until the optimizer step, every rollout in that buffer is still generated by the same policy. In other words, we get much better throughput without going off-policy (which opens a whole new can of worms).

The effect is especially noticeable as the number of gradient accumulation steps grows, as we show in the table below.

grad_accum	Sequential Time (s)	Batched Time (s)	Tokens/s (seq)	Tokens/s (batch)	Speedu
2	2.50	1.30	204.7	393.2	1.9x
8	10.05	1.38	203.8	1,482	7.3x
32	40.26	1.71	203.5	4,801	23.6x
64	80.43	1.93	203.7	8,494	41.7x

Including the buffer enabled fast distillation for teachers and students in the 8B scale. However, remember that our goal was to use 235B teachers. Loading teachers of that size on the same GPUs used for training wasn't feasible, so we started working on supporting using a server to get the teacher logprobs.

Improving Teacher Server Latency by 17x and Throughput by 2x

Including the generation buffer made distillation fast enough for teacher and student pairs around the 8B scale. But our real goal was to use much larger teachers, including models in the 100B+ range. At that scale, colocating the teacher and student on the same GPUs was no longer practical, so we moved the teacher to an external vLLM server and had the training

workers query it for log-probabilities. That solved the memory problem, but it introduced the challenge of handling many concurrent requests and returning large volumes of logprob data.

To make that setup practical, we focused on two optimizations:

- batching requests on the teacher server
- reducing the size of the logprob payloads sent back to the student.

The plots below show the overall impact of our optimizations, and the next two sections dive into the details of how we achieved these improvements.

Server-side Optimization: Tail Latency

p99 latency (seconds) at sustained 32 concurrent workers

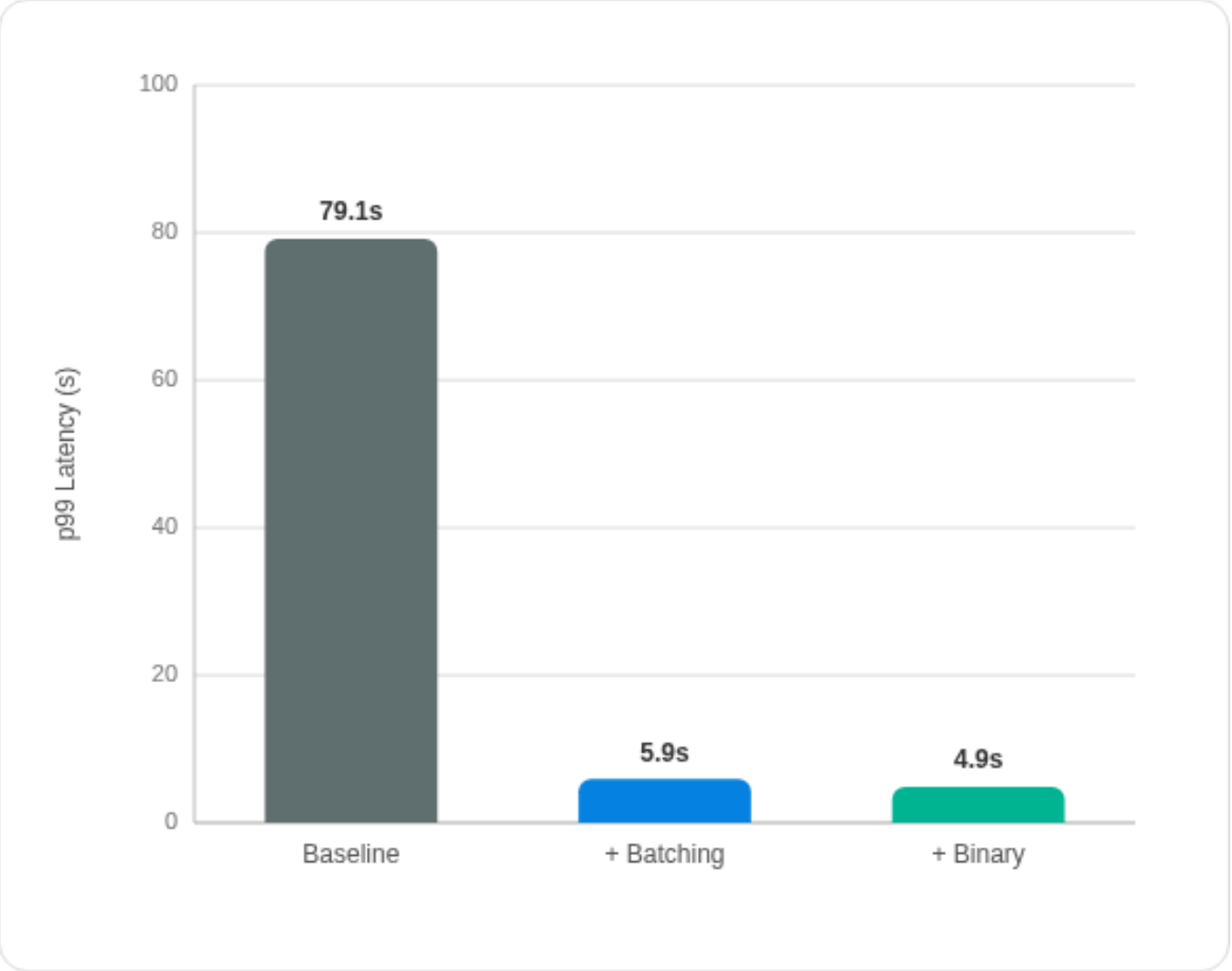


Figure 5 · Plot showing latency improvements from teacher server optimizations.

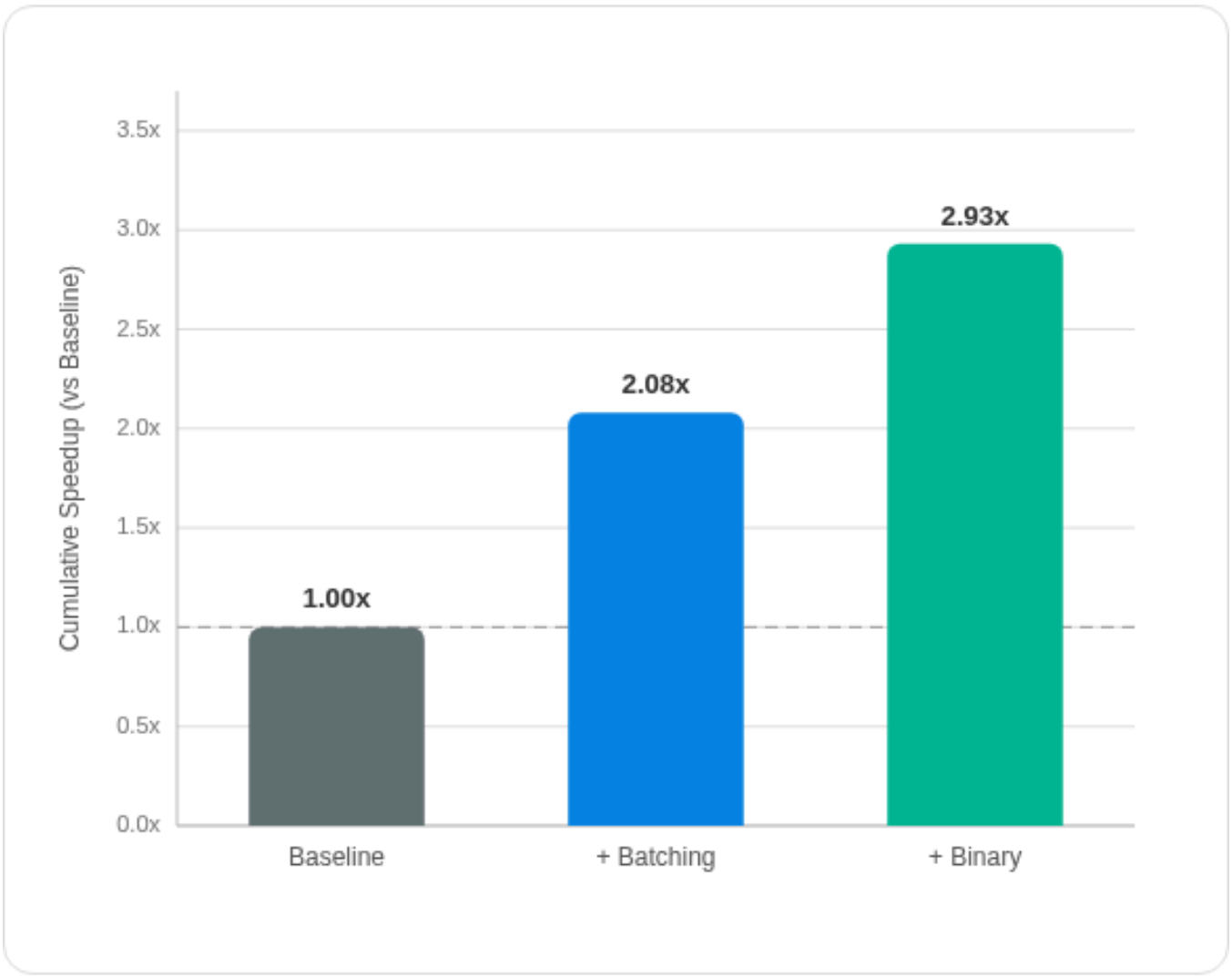


Figure 6 · Plot showing throughput improvements from teacher server optimizations.

Request Batching for 10x Latency Improvement

Before we added request batching, our data parallel (DP) workers were often sitting idle even though the teacher server still had capacity left. The problem was that requests were effectively being handled one at a time, so we were not benefiting from vLLM's internal batching on the server side. This became especially painful when several workers queried the teacher at once. Instead of being processed together, requests piled up in a queue and tail latency grew quickly. You can see that effect on the left side of the animation below.

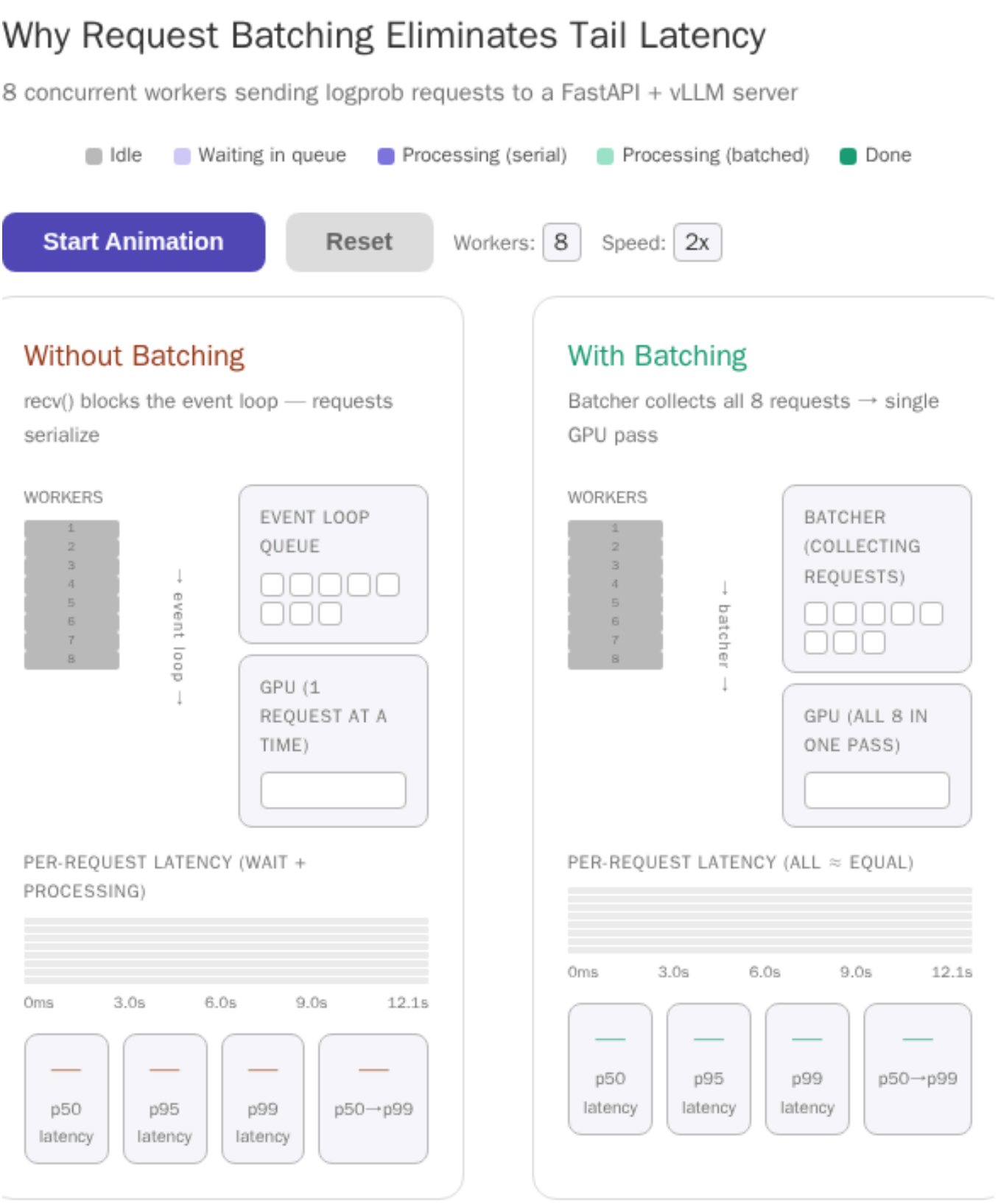


Figure 7 · Animation comparing request handling with and without batching.

This mattered a lot in practice because our main training setup used several training GPUs, each sending requests to the teacher server concurrently. To handle that more efficiently, we added a small batching layer on the server. Incoming requests were placed into an `asyncio.Queue`, and every few milliseconds, or whenever the queue reached a maximum size, we drained the queue and sent the combined batch to vLLM in one call. After that, we split the

results back into the individual responses expected by each worker. The animation on the right side shows the effect of batching the requests on stabilizing the latency for the requests in the queue.

We also added a token budget to keep the server stable on long sequences. In our implementation, that meant batching requests for up to 5 ms (`_BATCH_WAIT_S`) or until reaching a sequence limit (`_MAX_BATCH_SEQS`), while capping the total token load with `_MAX_BATCH_TOKENS = max_model_len * dp_size`. As the plot below shows, this one change reduced tail latency by about 10x.

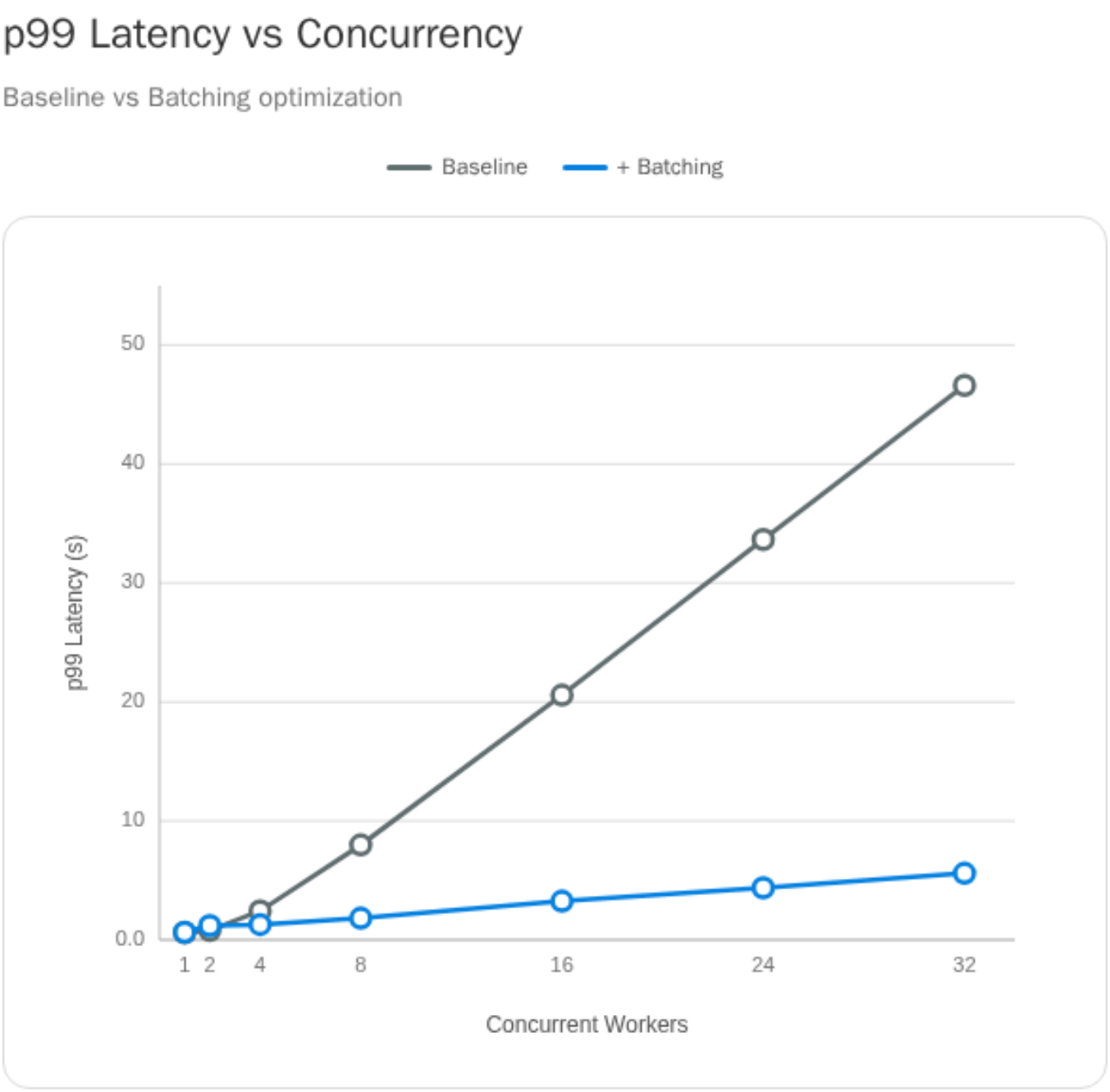


Figure 8 · Plot showing tail latency reduction from request batching.

Request batching reduced latency a lot, but we were still far from matching the performance of the colocated setup. At that point, the bottleneck shifted from getting the logprobs ready on the

server to transferring the logprobs to the client.

Binary Encoding for 5x Smaller Payloads

Once request batching made the teacher server fast enough, the next bottleneck was no longer computing the logprobs on the server, but sending them back to the training workers efficiently. This was a very different workload from standard text generation, because we needed to send back the top-k log-probabilities and token IDs for every position in the sequence. That added up quickly, especially for long rollout and many concurrent workers.

To make those responses cheaper to transfer, we changed how the server encoded the logprob data. Instead of returning nested Python lists in JSON, we packed the log-probabilities and token IDs into pre-allocated NumPy arrays with shape `(batch, max_completion_len, top_k)`. We then base64-encoded those arrays so they could still be sent safely in a JSON response. The final payload looked like the one below, and we serialized it with `orjson` while returning a raw `starlette.responses.Response` to bypass Pydantic validation overhead.

```
{
    "logprobs_b64": "str",
    "token_ids_b64": "str",
    "shape": [B, T, K],
    "completion_lengths": ["int"]
}
```

The biggest win came from avoiding the original list-of-lists representation, which carried a lot of Python and JSON overhead. This binary encoding reduced payload size by about 5x and cut latency by another 25%. It also made decoding much faster on the client side. Instead of rebuilding the response with a double Python loop, we could read the NumPy arrays directly, which made decoding about 25x faster.

Method	Mean Time (ms)	Speedup
Python double loop	436.4	1.0x
Numpy vectorized	17.0	26x

Scaling to Multiple Workers and Sequence Lengths

We also wanted to make sure these optimizations were still useful for different training scales, so we ran a set of tests with different numbers of workers requesting logprobs from the teacher server. The plot below shows that the improvements hold up well as concurrency increases.

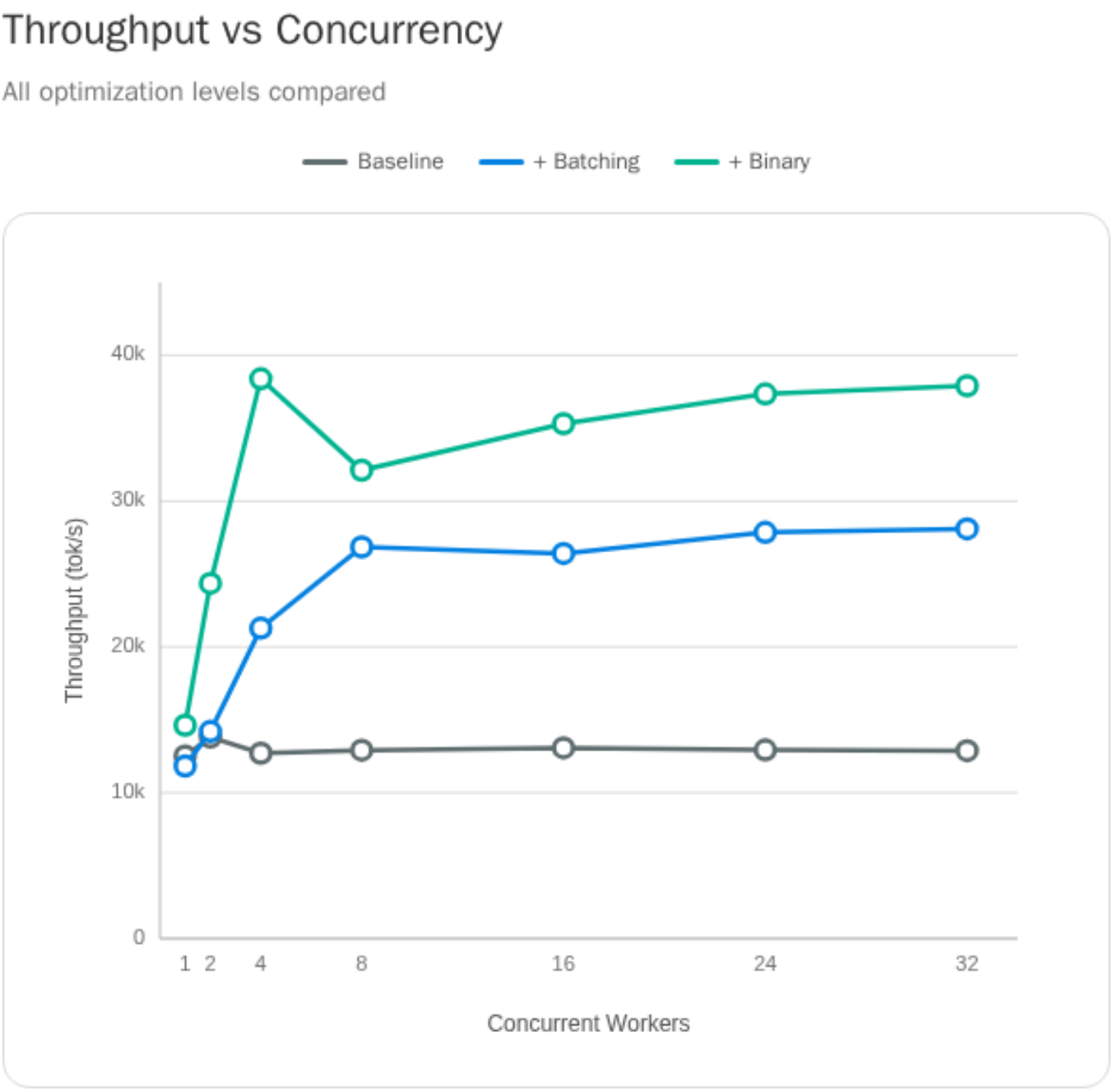


Figure 9 · Plot showing throughput scaling across different numbers of concurrent workers.

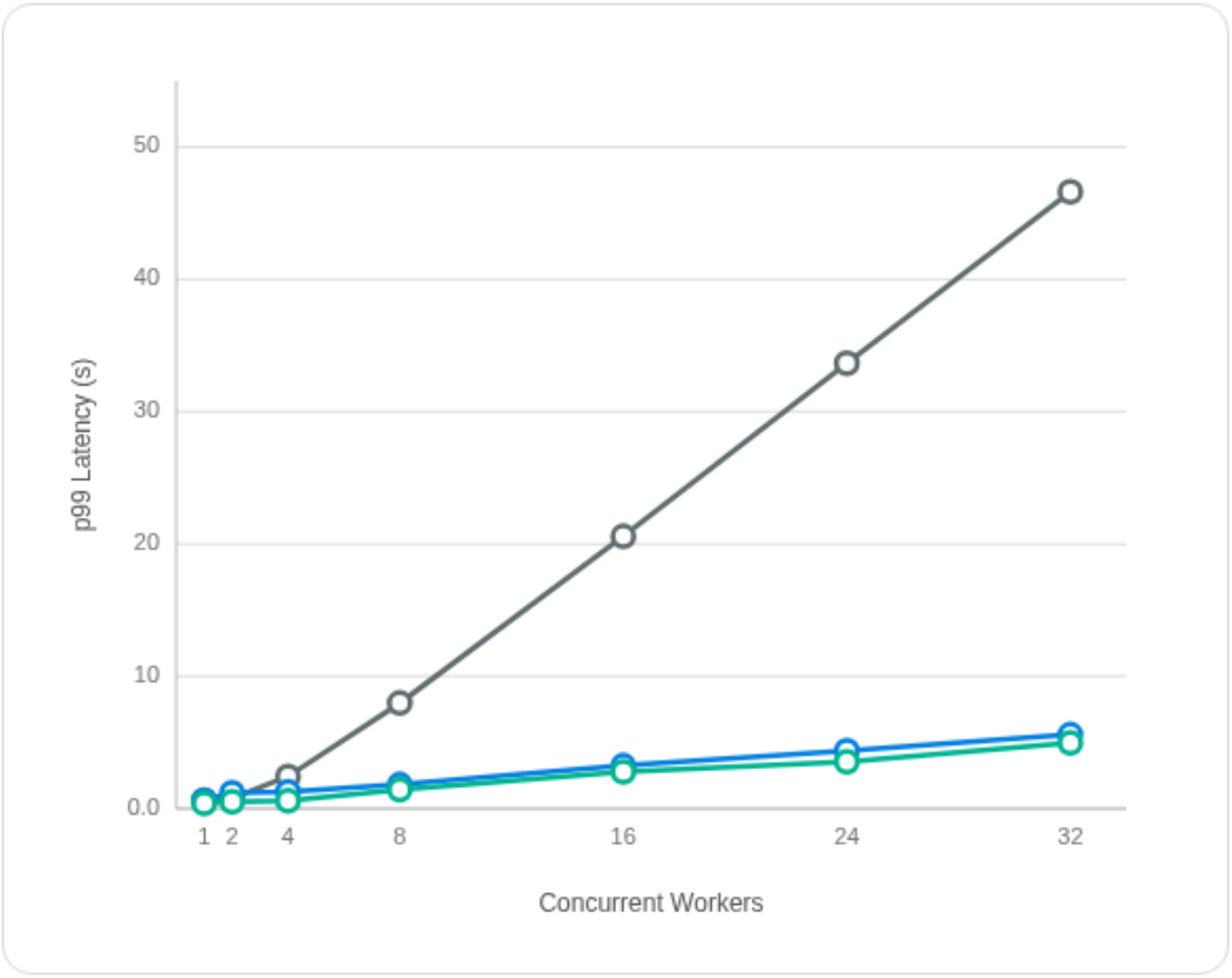


Figure 10 · Plot showing latency scaling across different numbers of concurrent workers.

One interesting detail is the throughput peak at around 4 concurrent workers. That is mostly because our benchmarking server was configured with DP=4, so there was a natural 1:1 match between the number of GPUs sending requests and the number of workers processing them. With fewer requests, the server is not fully utilized. With more, a queue starts to build up.

That queueing effect shows up even more clearly in the tail-latency plot. As we saw in the request batching animation, the baseline approach does not scale well once the queue grows, while the batched implementation stays much more stable under heavier traffic.

We also benchmarked how latency and throughput behaved across sequence lengths. The plots below show how our optimizations have better throughput than the baseline when scaling to longer sequences, and also latency doesn't increase as drastically as the baseline.

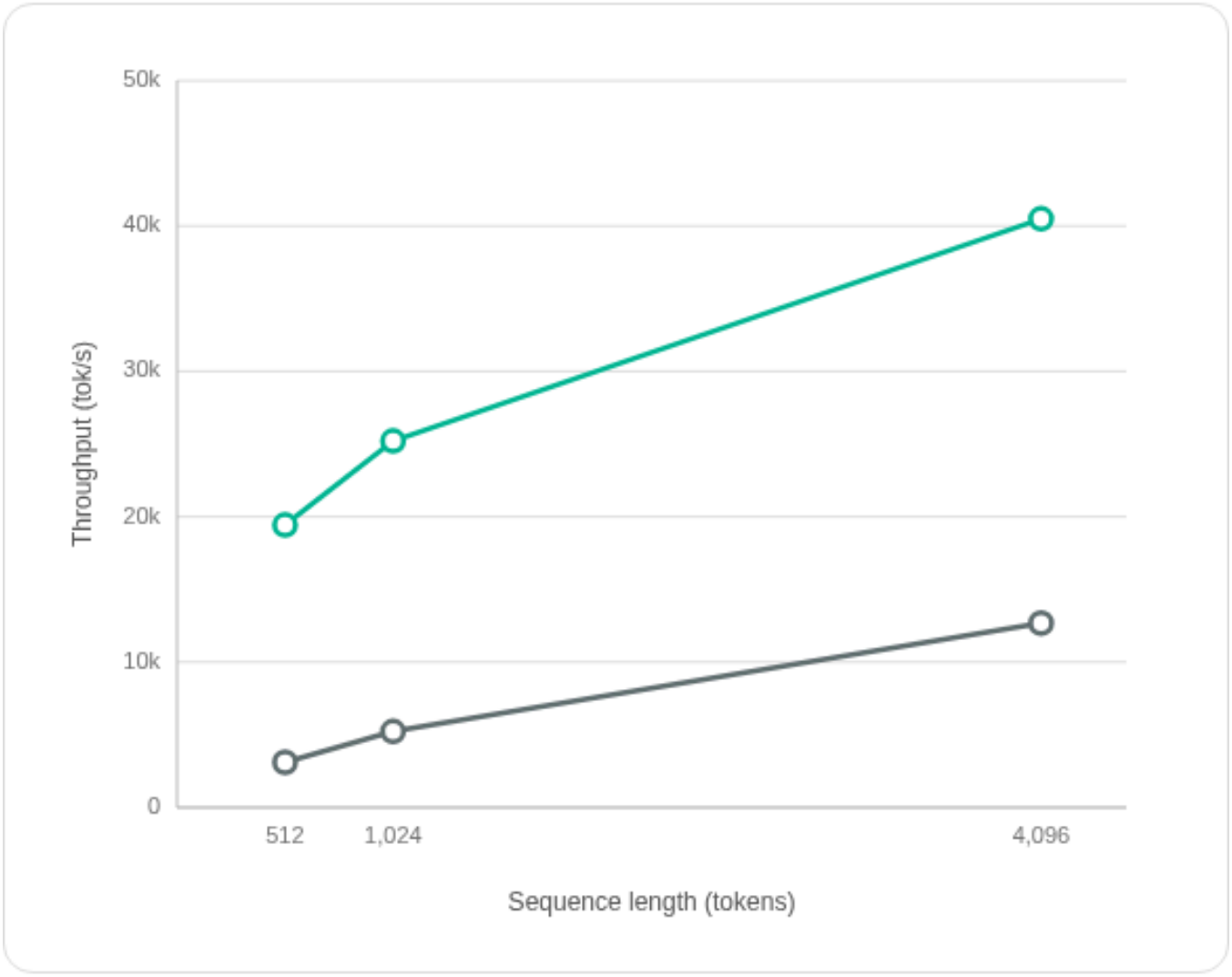


Figure 11 · Plot showing throughput comparison across sequence lengths.

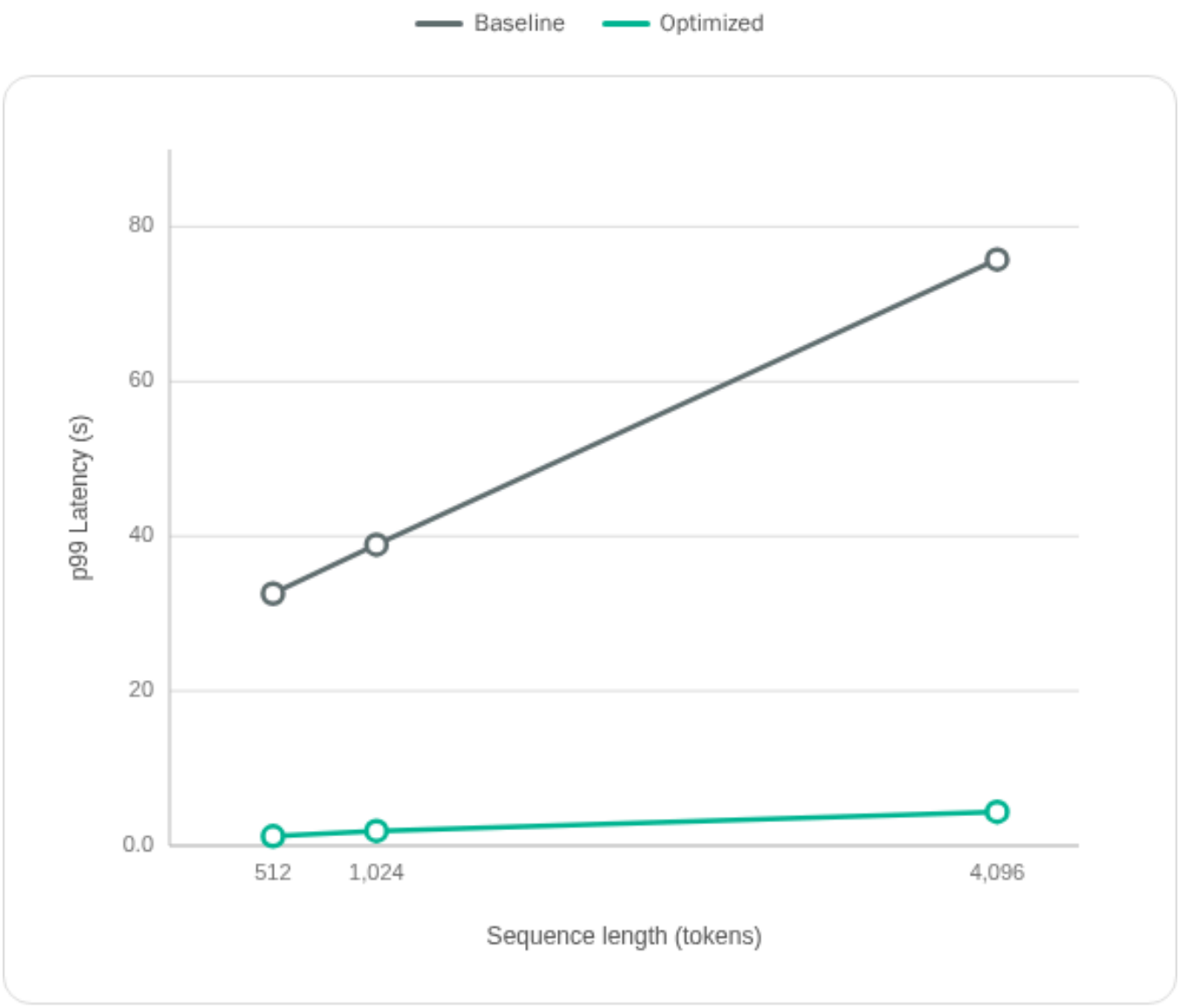


Figure 12 · Plot showing latency comparison across sequence lengths.

We finally reached a setup where we can run distillation with larger teachers! Let's use distillation to teach a model how to make fire.

Distilling a Model for a Camping Trip

After all the discussion about batching, payloads, and teacher servers, imagine you are packing for a camping trip. On your packing list is a small local model that can help with survival questions even without an internet connection. You decide to try the Gemma 4 models that

Google recently released, to decide if they give good answers about topics related to your camping trip.

When testing the models, you notice that `google/gemma-4-E2B-it` refuses to answer questions about how to make fire, while `google/gemma-4-31B-it` gives a more useful response by warning about the risks first and then explaining the steps to make a fire. You can see the responses from both models below.

Answer from google/gemma-4-E2B-it

Answer from google/gemma-4-31B-it

This is a nice opportunity for distillation because the larger model responds the way we want, but the smaller one is the model we would actually want to run locally. In theory, we could just use the 31B model directly, but that would require around 60GB of VRAM. For something like an offline camping assistant that runs on your device, a much smaller model is far more practical. So the goal here is to transfer that useful behavior from `google/gemma-4-31B-it` into `google/gemma-4-E2B-it`, along with other bushcraft and survival skills.

For training, we used HuggingFaceTB/CoT_Reasoning_Bushcraft_Survival, which is a reformatted version from the mattwesney/CoT_Reasoning_Bushcraft_Survival dataset covering questions about wilderness survival and bushcraft.

After only 150 distillation steps, the E2B student was already giving detailed and practical instructions for how to build a fire. It was also 2x faster while using half of the GPUs the 31B model used to generate a similar answer!

Answer from distilled google/gemma-4-E2B-it after 150 steps

Fire-making is a fun example, but the bigger point is that distillation lets us move useful behavior from large, expensive models into smaller models that are much easier to run in resource-constrained settings. That opens the door to specialized models that run locally on your phone or in the browser and can still handle the tasks you care about, which will likely matter more and more as agentic systems become part of everyday products.

Improving Math Reasoning Skills

The Gemma experiment worked well, but it still did not get us to the 100B+ teacher scale we were targeting at the beginning. To test that setting directly, we moved to the Qwen family and used distillation to transfer math skills into a 4B student.

We used prompts from the DeepMath-103k dataset to distill math capabilities into the `no-think` version of `Qwen/Qwen3-4B`. To study how teacher size affects the student, we compared the results when using `Qwen/Qwen3-30B-A3B-Instruct-2507` and `Qwen/Qwen3-235B-A22B-Instruct-2507` as teacher. Since our goal here was to improve math capabilities, we used AIME25 as the main evaluation benchmark.

The plot below shows that distillation substantially improves the 4B student on AIME25. In total, the student gains more than 39 points by learning from more capable teachers, which is a strong signal that domain-specific reasoning skills can be transferred effectively through distillation. Just as importantly, this experiment let us validate the full training setup with a teacher above 100B parameters.

At the same time, the results also show that scaling the teacher does not automatically translate into equally large gains for the student. Even though the 235B teacher performs almost 10 points better than the 30B teacher on AIME25, the distilled students end up at very similar levels. This is a sign of the capacity gap between teacher and student. If the teacher is much larger than the student, the student may no longer be able to absorb all of the extra capability. (Gu et al., 2023; Mirzadeh et al., 2020; Xu et al., 2025).

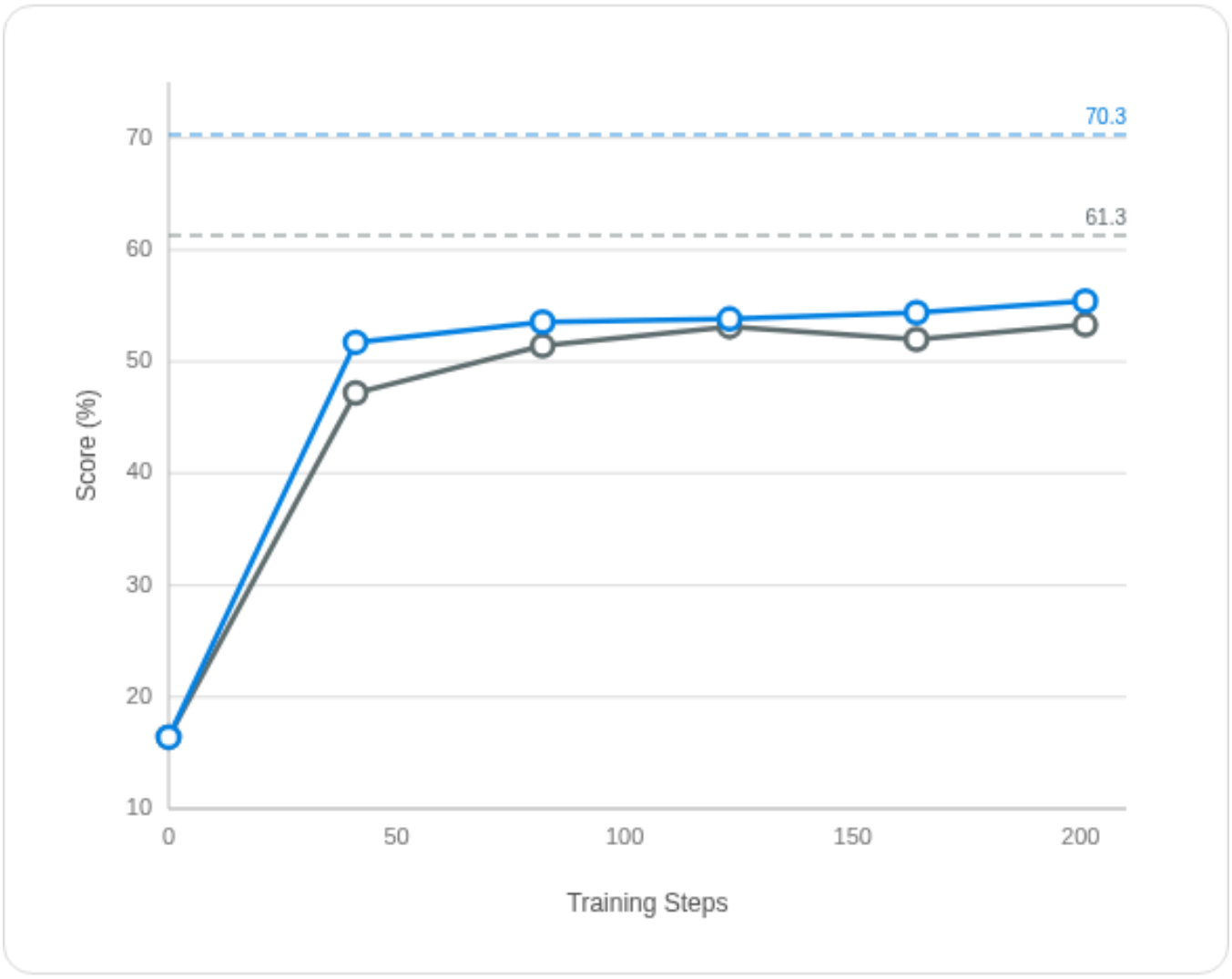


Figure 13 · Plot showing AIME25 results comparing distillation with 30B and 235B teachers.

We also tracked GPQA Diamond during training to check whether improving math reasoning came at the expense of other, out of distribution, capabilities. The plot below suggests that the answer may depend on the size of the teacher. When we distill from the 30B teacher, GPQA stays relatively stable throughout training. With the 235B teacher, GPQA drops by about 10% after step 40. Our current hypothesis is that this is another effect of the capacity gap, and that pushing too hard on one domain may make it harder for the student to retain performance elsewhere.

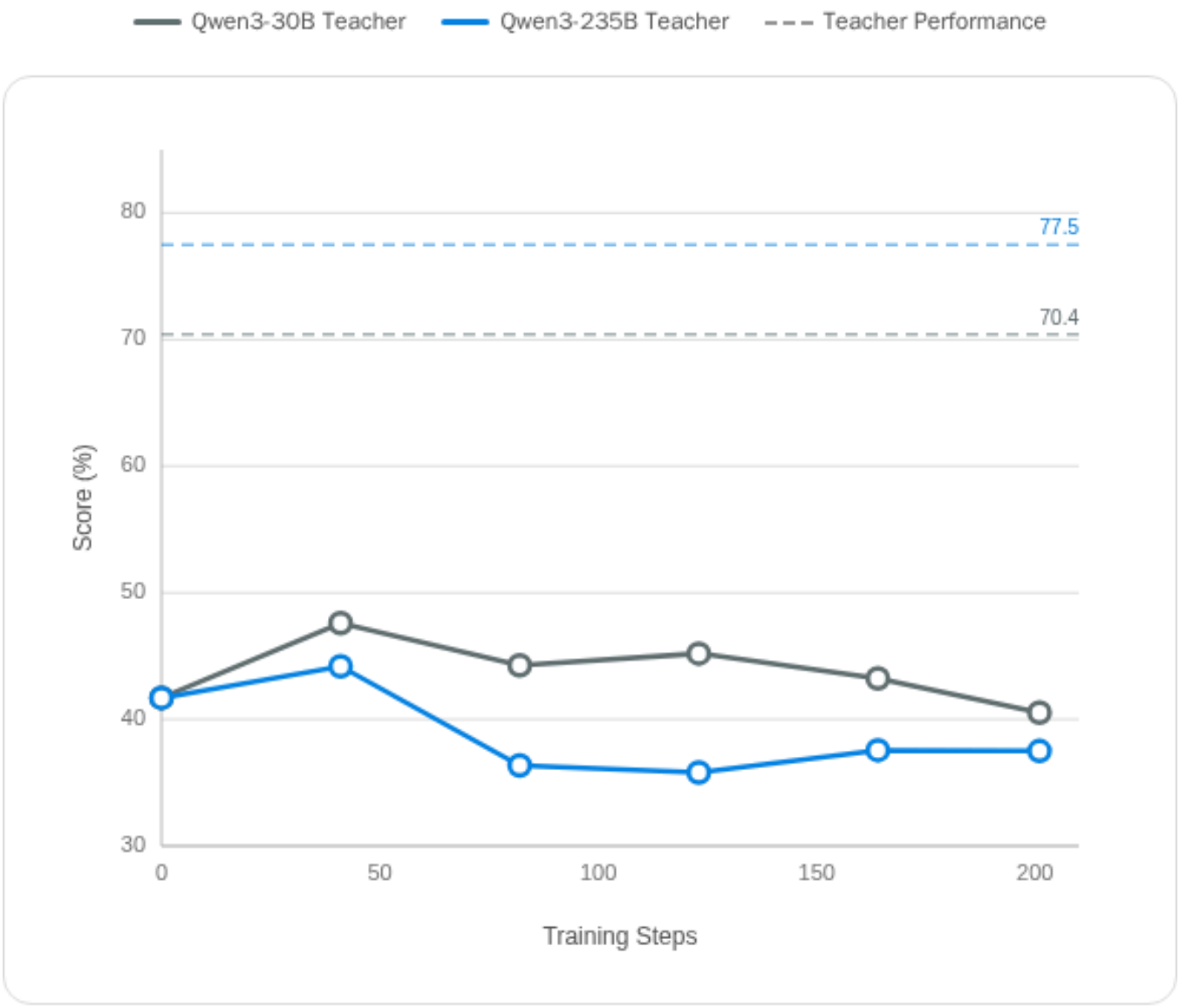


Figure 14 · Plot showing GPQA Diamond performance during distillation training.

Get Started with Distillation

Distillation is one of the most practical ways to transfer great capabilities from a large model into a small model you can actually use. With our `DistillationTrainer` in TRL, it is now much easier to run on-policy distillation with large teachers, efficient rollout generation, and a teacher server setup that can keep up with training.

So if there is a model on the Hub whose behavior you would love to transfer into something smaller, this is a great time to try it. Start with one of the examples in this post or in the docs,

swap in your own teacher and student, and watch while your student learns.

Citation

For attribution in academic contexts, please cite this work as

```
Carlos Miguel Patiño, Kashif Rasul, Edward Beeching, Lewis Tunstall (2026). "Distilling 100B+
Models 40x Faster with TRL".
```

BibTeX citation

```
@misc{patiño2026_distilling_100b_models_40x_faster_with_trl,
  title={Distilling 100B+ Models 40x Faster with TRL},
  author={Carlos Miguel Patiño and Kashif Rasul and Edward Beeching and Lewis Tunstall},
  year={2026},

}
```

References

1. Agarwal, R., Vieillard, N., Stanczyk, P., Ramos, S., Geist, M., & Bachem, O. (2023). GKD: Generalized Knowledge Distillation for Auto-Regressive Sequence Models. *arXiv Preprint arXiv:2306.13649*. https://arxiv.org/abs/2306.13649↑

2. Gu, Y., Dong, L., Wei, F., & Huang, M. (2023). Knowledge Distillation of Large Language Models. *arXiv Preprint arXiv:2306.08543*. https://arxiv.org/abs/2305.12129↑

3. Hugging Face H4. (2024). *GOLD: Generalized On-policy Language model Distillation*. Hugging Face Spaces. https://huggingface.co/spaces/HuggingFaceH4/on-policy-distillation↑

4. Jones, A. C. (2023). *KL(q||p) vs KL(p||q)*. Blog post. https://andrewcharlesjones.github.io/journal/klqp.html↑

5. Ko, Y., Shi, L. X., Sferrazza, C., Zhu, Y., Hausman, K., Sadigh, D., & Finn, C. (2025). Forward KL Regularized Preference Optimization for Aligning Diffusion Policies. *arXiv Preprint arXiv:2510.18874*. https://arxiv.org/abs/2510.18874↑

6. Mirzadeh, S. I., Farajtabar, M., Li, A., Levine, N., Matsuda, A., & Ghasemzadeh, H. (2020). Improved Knowledge Distillation via Teacher Assistant: Bridging the Gap Between Student and Teacher. *Proceedings of the AAAI Conference on Artificial Intelligence*, *34*(04), 5191–5198. https://arxiv.org/abs/1902.03393↑

7. Thinking Machines. (2024). *On-Policy Distillation*. Blog post. https://thinkingmachines.ai/blog/on-policy-distillation/↑

8. Xu, Z., Liu, Z., & Sun, M. (2025). On the Distillation of Reasoning Capabilities into Small Language Models. *arXiv Preprint arXiv:2501.16937*. https://arxiv.org/abs/2501.16937↑