

Reflection releases Beam, a 501B-parameter open model

Source copy kept by wheresthe.ai for the story "Reflection releases Beam, a 501B-parameter open model" (<https://wheresthe.ai/d/us-reflection-releases-beam-a-501b-parameter-open-model-2026-10/>).

Source: <https://huggingface.co/blog/reflection-beam>

Text saved 18 Oct 2026 (the page as published).

SHA-256 of the saved text: d62fbd011c14e82a15d03a9d861505c6ad710f52a334ae4b7e4d7066f5838f30

Back to Articles a]:hidden">

Welcome Beam: a 501B open model you can own, customize, and scale

Published October 16, 2026

Update on GitHub Upvote 29

+23

ben burtenshaw burtenshaw Follow

merve merve Follow

Pedro Cuenca pcuenq Follow

:last-child]:mb-0">

Beam by Reflection is out on Hugging Face! It is the first model from the new US lab, Reflection, and it's open! It's a mixture-of-experts (MoEs) model with 501B parameters (23B active per token), supports text as input, has a context window of 1M tokens , and is released under the Apache 2.0 license.

Beam is focused on coding and agentic use cases. It is for teams that want a frontier-scale model on their own hardware, infrastructure, or cloud, and want to customize it to their own use cases. Beam ships with its own dedicated open source Rust based harness that delivers the best performance from the model.

Beam has a native transformers implementation, as well as downstream libraries like vLLM and SGLang. We've worked with the Reflection team to bring you inference, SFT (supervised fine tuning), and RL (reinforcement learning) recipes for you to build with this model for your needs. On top of day-0 support in transformers, SGLang and vLLM, this post shows you how to run it, how to post-train it, and what we built with it.

TL;DR

501B-parameter MoEs LLM with 23B active per token. 52 layers, 256 experts per MoE layer, 8 routed per token.

Text in, text out with context window of 1M tokens.

Checkpoint formats: BF16 , FP8 , and NVFP4

Reasoning that can stay in context across turns (preserve_thinking), tool calling with a dedicated grammar. The model supports five reasoning values: "low", "medium" (default), "high", "xhigh", and "max".

Response template in the tokenizer.

Speculative decoding drafter model with DFlash2

Day-0 libraries: transformers, SGLang, vLLM. Full-parameter RL with Miles.

Ships with a Rust-based custom harness.

Apache 2.0.

Build with Beam

Reflection is going to give out \$10,000 Hub credits to the best projects that use Beam. All you have to do is make a thing, and share it on the Hub:

You could make a demo that uses the Beam model on Spaces

You could quantize it for your favorite hardware

You could generate a synthetic dataset for your favorite problem

You could build an RL environment with or for Beam

You could benchmark Beam and share the results.

You could surprise us!

If you build something with Beam, then share it on the Hub for a chance to win \$10,000. All you have to do is build the space, model, or dataset repo and share it on the model's repo page . We'll collect all the links, select the best, and send out the credits to your Hub accounts.

What is Beam?

Beam is a text-only model built for long-context (1M context length) reasoning and agentic work. It is a sparse MoE model, where each token is computed by a small fraction of the network, so it runs at the speed of a much smaller model while keeping the capacity of a large one.

Attention

The decoder has 52 layers with a hidden size of 5,120. Most layers use sliding-window attention over the previous 512 tokens; every sixth layer, and the last one, attend to the full context. That gives 43 sliding-window layers and 9 global ones.

Inside the attention block (both layer types)

The two kinds of layers handle position differently. Sliding-window layers use RoPE, since relative position matters most within a short window. Global layers use no positional encoding . Instead, each query is multiplied by a scaler that grows slowly with the number of visible tokens, $(1 + (\text{pos} + 1) / 2048) ** 0.1$. It acts as a temperature: the softmax keeps its precision as the context lengthens instead of flattening out. The implementation calls this scalable-softmax query scaling.

Attention uses grouped-query attention with 48 query heads and 8 key-value heads, with RMSNorm on queries and keys. A learned sigmoid gate, computed from the layer input, scales every channel of every head before the output projection.

Mixture of Experts

The first 3 layers use a dense SwiGLU MLP. The other 49 replace it with a Mixture-of-Experts block of 256 experts, each a SwiGLU MLP with an intermediate size of 2,560, or about 39M parameters. The experts hold 98.4% of the model's parameters.

The router follows DeepSeek-V3's group-limited scheme. It scores all 256 experts with a sigmoid, splits them into 8 groups of 32, keeps the 4 best groups (ranked by the sum of each group's top two scores), then picks the top 8 experts inside them. Beam uses the same group sizes as DeepSeek-V3 but no shared expert. Expert outputs are combined with normalized router weights and scaled by 2.5.

Only 8 experts run per token, so the model computes each token with about 23B parameters. All 256 experts stay in memory, so deployment needs the memory of a 501B model but runs at the speed of a 23B one. The BF16 checkpoint is about 1 TB in safetensors format. The NVFP4 checkpoint stores most parameters in 4-bit precision for NVIDIA Blackwell GPUs, some parameters are kept in BF16 for accuracy. There are also FP8 checkpoints on the hub which are ideal for Hopper and Ampere series chips.

If you want to deploy Beam yourself, check out the vLLM or SGLang recipes below.

Residual Stream

The residual connections are the most distinctive part. A standard transformer adds each sublayer output to the stream with weight 1, so the stream grows with depth.

Beam counts the input embedding as step 1. At step k it adds the sublayer output with weight $1/\sqrt{k}$ and keeps the existing stream with weight $\sqrt{(k-1)/k}$. The squares of the two weights sum to one, so the stream stays at a constant scale from the first layer to the last. After all 104 additions, every sublayer output and the input

embedding carry the same weight in the final hidden state.

Every sublayer is also normalized before and after, and the final logits pass through a tanh softcap at ± 30 , computed in FP32. The implementation fixes where it computes in FP32: the router, the SwiGLU product, the norms, RoPE, the query scaling and the softcap.

Long Context

Hybrid attention works well for long contexts. Each layer caches 2×8 KV heads $\times 128$ dims = 2,048 values per token. If all 52 layers kept every token, the cache would grow by 208 KiB per token. Only the 9 global layers keep the whole sequence. The 43 sliding-window layers hold a fixed 512-token window each and stop growing. So the cache grows by 36 KiB per token, roughly a sixth of the naive rate.

Context If all 52 layers were global Beam

32K tokens 6.5 GiB 1.2 GiB

128K tokens 26 GiB 4.6 GiB

Full context (1M) 208 GiB 36.08 GiB

These numbers are per sequence, in BF16, with the transformers dynamic cache.

Benchmark Results

Here are the benchmark results across different precision levels, as provided by the Reflection team.

Beam (BF16) Beam (FP8) Beam (NVFP4)

Agentic Coding / Terminal

DeepSWE v1.1 43.3% 43.0% 42.7%

SWE Bench Pro v2-Hard 76.5% 78.4% 78.6%

SWE Bench Pro v1 66.1% 65.9% 65.6%

Terminal Bench v2.1 80.1% 78.9% 78.2%

SWE Atlas Codebase QnA 31.8% 32.2% 31.4%

SWEBench Multilingual 78.3% 78.1% 77.5%

SWEBench Verified 81.6% 80.2% 80.7%

Reasoning

AIME 2026 97.7% 98.0% 97.5%

HLE (no tools) 36.0% 36.0% 34.8%

HLE (with tools) 51.1% 51.1% 51.1%

SciCode 49.2% 48.3% 46.9%

CriPT AA 16.3% 17.4% 15.4%

GPQA Diamond 90.6% 90.3% 89.3%

LiveCodeBench 88.3% 88.1% 87.4%

Tool Calling / Search

AutomationBench - public split only 39.3 38.2 39.8%

MCP Atlas 80.8% 81.5% 80.5%

Tau3 Banking 36.1% 36.8% 35.4%

BrowseComp w/ context management 76.8% 76.7% 75.6%

DeepSearchQA w/ context management 80.5% 79.6% 81.1%

General Capabilities

AA-LCR 78.2% 79.4% 78.7%

LongBench v2 67.5% 67.6% 67.2%

IFBench 79.6% 80.7% 79.2%

AA Omniscience (index) - public split only 10.8 8.8 9.8

Hardware Requirements

The most practical way to serve the model is serving the NVFP4 checkpoint, which retains most of the quality of the bf16 checkpoint. NVFP4 stores some weights in 4 bits, and this way, it shrinks the 501B parameters in BF16 from 1 TB to ~280 GB. Blackwell GPUs run NVFP4 natively but GPUs older than Blackwell like the H200 can't. The FP8 checkpoints are suitable for GPUs older than Blackwell.

A B200 has 180 GB of HBM3e, so two of them (360 GB) can hold the Beam weights, with little room left for activations and the KV cache. Plan for four B200s (720 GB), which leaves plenty of room for long contexts and concurrent requests. You can also serve it on a full 8xB200 node, or on two B300s at 288 GB each, or even 8xRTX Pro 6000. If you don't have Blackwell hardware, use Inference Providers (serverless) to test Reflection.

Inference Support

Beam comes with day-0 support in transformers, SGLang and vLLM.

Transformers

Install the latest version of transformers.

```
pip install -U "transformers>=5.20.1"
```

The easiest way to run Beam is the text-generation pipeline:

```
from transformers import pipeline
```

```
pipe = pipeline("text-generation", model="reflection/Beam-1.0-501B-A23B", device_map="auto")
```

```
messages = [ {"role": "user", "content": "Write a limerick about Python exceptions." } ] output = pipe(messages, max_new_tokens= 2000 ) output[ 0 ][ "generated_text" ][ - 1 ]
```

The pipeline returns the whole chat. The last message is Beam's reply, already parsed by the response template.

Going one level lower, use AutoModelForCausalLM and AutoTokenizer .

```
from transformers import AutoModelForCausalLM, AutoTokenizer
```

```
model_id = "reflection/Beam-1.0-501B-A23B" tokenizer = AutoTokenizer.from_pretrained(model_id)
```

```
model = AutoModelForCausalLM.from_pretrained( model_id, dtype= "auto", device_map= "auto" )
```

Chat

```
messages = [ {"role": "system", "content": "You should only answer with a number." }, { "role": "user", "content": "What is 17 * 23?" }, ]
```

```
inputs = tokenizer.apply_chat_template( messages, add_generation_prompt= True , tokenize= True , return_dict= True , return_tensors= "pt", reasoning_effort= "medium" # by default ).to(model.device)
```

```
input_len = inputs[ "input_ids" ].shape[ - 1 ]
```

```
outputs = model.generate(**inputs, max_new_tokens= 2000 )
```

```
tokenizer.parse_response( outputs[ 0 ][ input_len:], prefix=inputs[ "input_ids" ][ 0 ] )
```

Beam's tokenizer carries a response template. parse_response reads it and turns the raw output into a message dict, with the reasoning, the answer, and any tool calls in separate fields. This avoids the usual regex for splitting

reasoning from answers from tool calls. And the rules ship with the tokenizer, so the parsed structure matches what Beam was trained to produce. The chat template can pre-write part of the assistant turn, which the parser needs to see.

Tool calling

Complete tool-calling example

```
def get_weather ( city: str , unit: str = "celsius" ): """ Get the current weather for a city.
```

Args: city: The city to look up. unit: Temperature unit, "celsius" or "fahrenheit". """ ...

```
tools = [get_weather] messages = [ { "role" : "user" , "content" : "I land in Paris tonight. Do I need a coat?" } ]
```

```
inputs = tokenizer.apply_chat_template( messages, tools=tools, add_generation_prompt= True , tokenize= True ,  
return_dict= True , return_tensors= "pt" , ).to(model.device)
```

```
input_len = inputs[ "input_ids" ].shape[ - 1 ] outputs = model.generate(**inputs, max_new_tokens= 1000 )
```

```
message = tokenizer.parse_response( outputs[ 0 ][input_len:], prefix=inputs[ "input_ids" ][ 0 ] , tools=tools )  
message.get( "tool_calls" )
```

Pass tools to `parse_response` as well. Beam writes tool-call arguments inline, in its own XML-style grammar, so a bare 7 could be a number or a string. With the tool schemas, the parser casts each argument to its declared type.

Multi-GPU and multi-node

Beam's config defines an expert-parallel plan. Expert parallelism puts each expert on one device. Attention, the dense layers, the embeddings and the LM head are replicated on every device. That is 8B parameters, or 16 GB in BF16. Load the model with a `DistributedConfig` :

```
import os
```

```
from transformers import AutoModelForCausalLM from transformers.distributed import DistributedConfig
```

```
model = AutoModelForCausalLM.from_pretrained( "reflection/Beam-1.0-501B-A23B" ,  
distributed_config=DistributedConfig( tp_size= int (os.environ[ "WORLD_SIZE" ]), ep_size= int (os.environ[  
"WORLD_SIZE" ]), tp_plan={} ), )
```

Launch the script with `torchrun` across 16 H200 GPUs. Each one then holds 78 GB of weights. With 8 GPUs each would hold 139 GB, which leaves no room on a 141 GB H200. The number of devices must divide the number of experts, 256.

vLLM

Beam has day-0 support in vLLM. The command below serves the BF16 checkpoint with tensor parallelism across 16 GPUs (two 8x H200 nodes) and exposes an OpenAI-compatible API:

```
vllm serve reflection/Beam-1.0-501B-A23B --tensor-parallel-size 16 --served-model-name beam
```

For faster generation, add DFlash2 speculative decoding. A small drafter model proposes up to 15 tokens ahead, and Beam checks them all in one forward pass, so each step can accept several tokens. The cost is a little extra memory for the drafter:

```
vllm serve reflection/Beam-1.0-501B-A23B \ --tensor-parallel-size 16 \ --speculative-config '{ "method": "dflash",  
"model": "reflection/dflash2-drafter-preview", "num_speculative_tokens": 15 }'
```

Both commands start an OpenAI-compatible server, so you can point any OpenAI client at `http://localhost:8000/v1` . For additional serving configurations, see the proposed Beam recipe for vLLM .

SGLang

Beam also has day-0 support in SGLang. We recommend serving it with DFlash2 speculative decoding, which both commands below turn on. On Blackwell, the NVFP4 checkpoint fits on 4 B300s:

```
python -m sglang.launch_server \ --model-path reflection/Beam-1.0-501B-A23B-NVFP4 \ --served-model-name  
beam \ --tp 4 \ --attention-backend trtlm_mha \ --page-size 64 \ --context-length 1048576 \ --mem-fraction-static
```

```
0.85 \ --enable-unified-memory \ --speculative-algorithm DFLASH \ --speculative-draft-model-path reflection/  
Beam-1.0-501B-A23B-DFlash2 \ --speculative-dflash-block-size 8 \ --tool-call-parser beam --reasoning-parser  
beam \ --host 0.0.0.0 --port 30000
```

On Hopper GPUs, use the FP8 checkpoint on 8 H200s instead. Only three flags change: the checkpoint, the tensor-parallel size (8 instead of 4), and the attention backend (fa3 instead of trtllm_mha, which needs Blackwell):

```
python -m sglang.launch_server \ --model-path reflection/Beam-1.0-501B-A23B-FP8 \ --served-model-name beam  
\ --tp 8 \ --attention-backend fa3 \ --page-size 64 \ --context-length 1048576 \ --mem-fraction-static 0.85 \ --enable-  
unified-memory \ --speculative-algorithm DFLASH \ --speculative-draft-model-path reflection/Beam-1.0-501B-  
A23B-DFlash2 \ --speculative-dflash-block-size 8 \ --tool-call-parser beam --reasoning-parser beam \ --host 0.0.0.0  
--port 30000
```

Inference Providers

Beam is available on Hugging Face Inference Providers, powered by Baseten, Fireworks, and Together, with more to follow soon.

```
import os from openai import OpenAI
```

```
client = OpenAI( base_url= "https://router.huggingface.co/v1" , api_key=os.environ[ "HF_TOKEN" ] , )
```

```
completion = client.chat.completions.create( model= "reflection/Beam-1.0-501B-A23B:auto" , messages=[{ "role" :  
"user" , "content" : "What is the capital of France?" }], )
```

```
print (completion.choices[ 0 ].message)
```

Inference Endpoints

We've set up a Beam recipe for you to one click deploy in Inference Endpoints from the catalog in FP8 or NVFP4 , or install with the hf cli like so:

```
hf endpoints catalog deploy --repo reflection/Beam-1.0-501B-A23B --name beam
```

What we tested

We are sharing some vibe evals of Reflection Beam-501B-A23B on five tasks, once each at low and high reasoning effort through the Reflection platform. These are vibe evals, not benchmarks. If you're planning on using Beam, this is a great way to kick the tires and see how it behaves on specific use cases, and importantly, where it wins and loses.

Below is an overview of the prompts we ran and the output traces from the hub. The complete inputs, outputs, API usage, task fixtures, graders and conversation traces are in this public Hugging Face bucket .

Category	Prompt Description	Low reasoning (tokens)	High reasoning (tokens)
----------	--------------------	------------------------	-------------------------

Multi-step tool calling	Fulfill ORD-742: compare stock, reservations, surcharges and arrival dates; reserve the cheapest eligible warehouse.	✓ 544	✓ 811
-------------------------	--	-------	-------

Long-context retrieval over a code base	Derive Click command configuration and two invocation results from three complete source files; cite six implementation symbols.	● 1,613	* ✓ 5,007
---	--	---------	-----------

Terminal task in a sandbox	Inspect and repair a CSV ledger CLI; fix all bugs, preserve tests, run tests and a CLI check.	✓ 582	✓ 1,548
----------------------------	---	-------	---------

Math reasoning (AIME-style)	How many eight-element subsets of {1, ..., 16} have a sum divisible by 17? Give a justification.	● 829	† ✓ 2,809
-----------------------------	--	-------	-----------

Structured output	Convert an out-of-order revision log into strict JSON: merge patches, cancellations, nulls, dates, currencies and totals.	✓ 1,040	✓ 2,088
-------------------	---	---------	---------

* Low's long-context response got all code behavior and all six source citations correct. However, the response did not correspond to the parsers format, so this is partly prompt ambiguity.

† Low returned the correct answer, but omitted the requested justification. High supplied a correct roots-of-unity proof, independently reviewed.

Don't take our word for it. Dig into the bucket , rewrite for your use case or project and push Beam to the limit.

Post-training Beam

You can fine tune Beam for your own use cases. Here, we share some examples in TRL and Miles to get you started on supervised fine-tuning and reinforcement learning.

SFT with TRL

The quickest way to adapt Beam to your own data is supervised fine-tuning with TRL's SFTTrainer. To keep memory manageable, we train LoRA adapters with PEFT rather than updating all 501B parameters: We used TRL's SFTTrainer to adapt Beam to a simple goal: keep its reasoning in the language requested by the user. We fine-tuned on Multilingual-Thinking with LoRA, which trains a small set of adapter weights while keeping the base model frozen.

The setup combines model sharding at load time, expert parallelism, FlashAttention 4 and sequence packing. LoRA SFT fits on a single node with 8 B300 GPUs, reaching around 19,000 tokens per second. The corresponding H100 setup uses four nodes.

After one epoch of Multilingual-Thinking, Beam reasoned in the requested language on 16 out of 16 held-out prompts, compared with 3 out of 16 for the base model. This small evaluation measures language adherence, rather than the correctness of the reasoning itself.

Training loss (left) and mean token accuracy (right) during a separate 500-step run on a reasoning subset of open-r1/Mixture-of-Thought, using one 8-GPU B300 node. These training metrics are from the official TRL example PR .

GRPO with Miles

Reflection and RadixArk built a recipe that trains the model with GRPO in Miles, in BF16. You can update the full model or use LoRA to train adapters while keeping the base weights frozen. Megatron trains, SGLang generates the rollouts, and Miles publishes the new weights to the rollout engines after each update. We reviewed the recipe.

Workload Reference configuration

Inference / eval, BF16 16x H200 per engine

Full-parameter GRPO, BF16 (Miles) 96x H200: 64 train, 32 rollout. Or 64x H200 colocated.

LoRA GRPO, BF16 (Miles) 32x H200: 16 train, 16 rollout.

Full-parameter GRPO, BF16, 128-token context (Miles) 48x H200: 32 train, 16 rollout. The setup Reflection and RadixArk first validated on the real checkpoint.

The recipe runs GRPO on DAPO-Math-17K and evaluates on AIME25. Build the image, download the data, and submit the job to a Ray cluster of twelve 8x H200 nodes:

```
docker build -f docker/Dockerfile.beam -t miles-beam .
```

```
python scripts/run_beam.py prepare \ --model-id reflection/Beam-1.0-501B-A23B \ --model-dir /root/models/Beam \ --data-dir /data/datasets
```

```
export MASTER_ADDR=10.0.0.1 export RAY_ADDRESS=http:// ${MASTER_ADDR} :8265
```

```
python scripts/run_beam.py execute \ --model-dir /root/models/Beam \ --data-dir /data/datasets \ --output-dir /root/shared_data/beam-run \ --num-rollout 100
```

What happens in a rollout:

Sample: 32 prompts are passed to the model with 8 responses each. Two SGLang engines serve the rollouts on 16 GPUs each.

Score: the boxed answer is checked against the label, with a soft penalty for responses near the length limit.

Update: one optimizer step on all 256 samples. Group-relative advantages with truncated importance sampling, no KL or entropy penalty.

Train in parallel: tensor parallelism 8, expert parallelism 8, data parallelism 8, with sequence parallelism and full recompute.

Optimizer: Adam at a constant 1e-6. Gradients in BF16, master weights in FP32, moments in BF16. The optimizer state streams through node-local NVMe, so each trainer node needs 2 TB of scratch.

Evaluate: AIME25 every five rollouts, eight samples per problem. A checkpoint is saved every twenty.

The next rollout is generated while the current one trains, so responses are one update behind. Add `--resume` to continue from the last checkpoint. `execute-colocate` runs the same recipe on eight nodes, with every GPU training and serving in turn.

Train with LoRA

You can also run GRPO with LoRA on four 8× H200 nodes: 16 GPUs for training and 16 for rollouts. The recipe trains rank-16 adapters on Beam’s attention, dense MLP, and expert layers while keeping the base weights frozen. Only adapter weights are synchronized with SGLang, and the optimizer state stays in GPU memory, removing the need for NVMe optimizer offloading.

```
python scripts/run_beam_lora.py execute \
--model-dir /root/models/Beam \
--data-dir /data/datasets \
--output-dir /root/shared_data/beam-run \
--num-rollout 100
```

Build with Beam: Demos

If you build something with Beam, then share it on the Hub for a chance to win \$10,000. All you have to do is build a space, model, or dataset repo and share it on the model’s repo page . To inspire you, here are a few things we built.

Hey Beam, build a native local MacOS application!

At Hugging Face we love open and local models that run native to their hardware, but building native apps on Mac can be hard! Let’s see if Beam can help us out.

We asked Beam to build a native MacOS application that turns speech to text, and conveniently drops it at your cursor. We want to use MLX to get the most out of our Mac Metal and the Parakeet Model for great speech-to-text.

Build BeamSpeak, a native macOS menu bar dictation app with a compact recording pill and no main window or account.

Use `⌘⇧Space` to start and stop microphone recording, showing elapsed time, microphone level, and transcription status. Run speech recognition entirely locally using a Hugging Face model, preferably Parakeet v3 through MLX, and keep the model loaded between recordings. Paste the transcript into the original app if it remains focused and Accessibility permission is granted; otherwise copy it to the clipboard, preserving newer clipboard changes and never pressing Enter. Handle permissions, cancellation, silence, errors, a two-minute recording limit, and temporary audio cleanup. Deliver source code, a runnable BeamSpeak.app, setup instructions, and verification of offline transcription, the physical shortcut, microphone capture, and text delivery.

We used Beam with the Pi coding harness to build the initial native macOS menu bar app, running Parakeet with MLX. The saved run includes follow-up prompts; the final source also includes later Codex repairs. The prompt above is a condensed version of the build requirements.

You can check out the agent trace of how Beam built this in pi in this trace . If you want to try the demo out, download the source code or read the setup instructions .

Hey Beam, how do you scale on multiple agents?

We wanted to see whether Beam could make faster progress by splitting a programming task across agents. We paired Beam with the open-source Pi coding agent and compared three setups: 1. one agent working alone, 2. five persistent peers collaborating through a shared Git repository, and 3. a lead agent delegating work to asynchronous subagents. The harness ran on Hugging Face jobs with inference through Reflection’s platform API.

For the tasks, we used ProgramBench: agents must recreate a program from its compiled executable and

documentation, without seeing its source code or the evaluator's tests. Each setup had the same 8M API-token ceiling and two-hour solver limit per task, shared across its agents. We saved implementation snapshots and evaluated them against fixed hidden-test suites from the ProgramBench-Tests dataset .

Our score-versus-latency plot shows the average fraction of tests passed. The time axis uses fixed reference rates for model tokens plus measured tool time, rather than elapsed wall time. We initially ran five tasks. The current figure shows a post-hoc subset of jq, xsv, FIGlet, and hexyl; ripgrep was excluded across all three setups because grading its saved artifacts was too slow. The published results retain all 15 original episodes.

You can inspect the runs in the Trackio dashboard , explore the published result data , and reuse the study harness .

Hey Beam, finetune a decision model for my code base!

Decision models are all the rage at the moment because they're super fast and practical for many use cases. We started with Beam in Pi after Beam's quota ran out to train and evaluate a custom decision model for developer incident triage.

We wanted to see whether a coding agent could turn an incident-triage policy into a small, fast decision model. The task was to read a developer incident and choose the responsible team, its severity, and whether someone should be pinged immediately. Using Unsloth, we fine-tuned Qwen3.5-0.8B to score these choices directly.

First, the workflow built a synthetic dataset containing 2.3k incidents split by incident cluster for training, calibration, and testing. We trained a LoRA adapter and decision head for three epochs, with a 4-bit backbone, on one H100 80GB on the Science Slurm cluster.

For a baseline, we prompted the unadapted model with the full triage policy and ticket, without examples, and asked it to generate the three decisions as JSON. On the same 320 synthetic test tickets, the fine-tuned decision model got all three answers correct on 76.25%, versus 4.69% for prompted zero-shot. Mean ticket latency fell from 1,218.06 ms to 102.58 ms, an 11.9× speedup.

If you want to try out this experiment for yourself, check out the artifact bucket which contains the data, code, checkpoints and results.

Wrapping Up

Beam is a 501B open model that you can run on your own hardware and change. Start with the BF16 checkpoint, or try one of the quants FP8 , NVFP4 . If you build something with Beam, share it in the discussions tab .

Models mentioned in this article 3

Datasets mentioned in this article 2

Spaces mentioned in this article 1

More Articles from our Blog

llm rl trl

Async GRPO with LoRA across HF Jobs: a bucket, a proxy, and no NCCL

53 September 10, 2026

llm rl trl

Shipping a Trillion Parameters With a Hub Bucket: Delta Weight Sync in TRL

+4

50 May 27, 2026

Community Well why don't you build something that can run on consumer GPU like Qwen 3.8 Flash Next for example using strata ? </p>\n", "updatedAt": "2026-10-16T18:35:13.302Z", "author":

{ "_id": "6313c832c093ff968e0d7ced", "avatarUrl": "/

avatars/1eb4ad0105ad152c175a8f01be57819d.svg", "fullName": "openroom", "name": "openroomxyz", "type": "user", "isF { "language": "en", "probability": 0.8631231188774109 }, "editors": ["openroomxyz"], "editorAvatarUrls": ["/

avatars/1eb4ad0105ad152c175a8f01be57819d.svg"],"reactions":
[],"isReport":false}}],"status":"open","isReport":false,"pinned":false,"locked":false,"collection":"community_blogs"},"cor
["burtenshaw","merve","pcuenq"],"primaryEmailConfirmed":false,"discussionRole":0,"acceptLanguages":
["en"],"withThread":true,"cardDisplay":false,"repoDiscussionsLocked":false,"hideComments":true}">

openroomxyz 1 day ago

Well why don't you build something that can run on consumer GPU like Qwen 3.8 Flash Next for example using strata ?

Reply

Edit Preview

Upload images, audio, and videos by dragging in the text input, pasting, or clicking here . Tap or paste here to upload images

Comment · Sign up or log in to comment

Upvote 29

+17

Models mentioned in this article 3

Datasets mentioned in this article 2

Spaces mentioned in this article 1