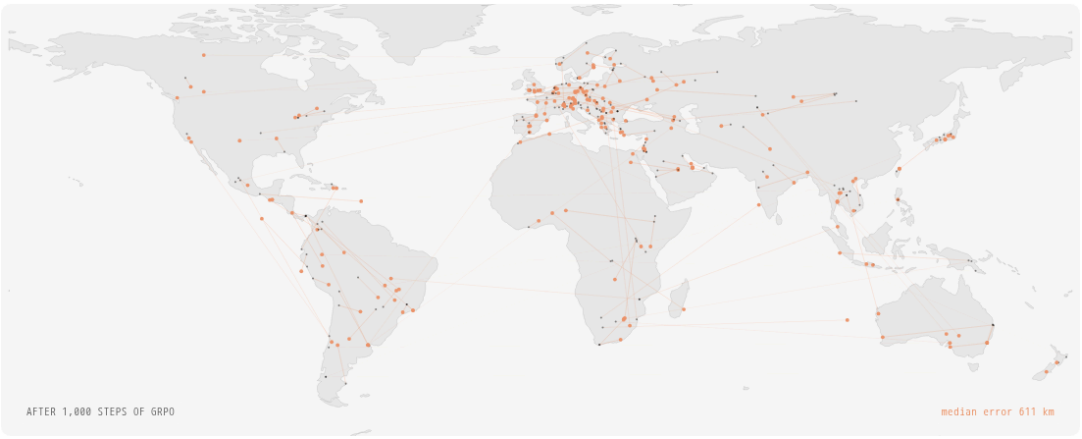
How to turn a game into an RL environment

the technical intuition



Curating the dataset, designing the environment, and shipping it with OpenEnv, then training a 4B with TRL until it outscored gpt-5.4-mini and claude-haiku-4.5 at GeoGuessr.

AUTHOR

Adithya S Kolavi

AFFILIATION

Hugging Face

PUBLISHED

September 8, 2026

CODE

HuggingEnvs ★

Table of Contents

Introduction

There are plenty of RL environments on the internet. Repositories, gyms, benchmark suites, papers with an appendix describing the observation space. Very few of them explain how one got *made*. How you go from a vague idea to a working environment, and then keep iterating on it until a model can be trained to do well at it.

That gap matters, because a finished environment hides almost everything interesting about building it. Presented as a clean API it looks inevitable. What the agent may do, what the reward pays for, what a tool is allowed to reveal. Every one of those is a judgement call, and most of ours were wrong the first time. Building one takes a lot of intuition and considerably more iteration than the finished thing admits to.

So this is the path for one idea, in the order we actually walked it, with the dead ends left in.

The idea was to make a vision-language model play GeoGuessr. Everything else follows from that. Curating a dataset that makes the task learnable, building the environment, deploying it so other people can use it, running evals we could trust, and finally training a model that performs well on it. Built with OpenEnv, trained with TRL.

It worked well enough to write down. A Qwen3.5-4B with a LoRA adapter finished ahead of `gpt-5.4-mini` and `claude-haiku-4.5` on the held-out split, ahead of every Qwen3.5 up to 397B, and behind only `claude-sonnet-5`. Ten hours on four A100s, around $100, and it runs on a single GPU if you want to try it more cheaply. None of that came from a clever algorithm. It came from the decisions in the sections below, which is why those are the subject rather than the number.

It also ended up somewhere we did not expect. The trained model does not look around more carefully; it looks *less* and answers in one turn, and it is more accurate that way. It is not a random guess, and it is not the reward being gamed. Working out which of those it was took us longer than the training did, and the evidence is in “What it actually learned”.

What GeoGuessr is

You get dropped somewhere on Earth at street level with no metadata. Turn your head, walk down the road, zoom in on a sign. When you think you know where you are, drop a pin and commit. You are scored on how many kilometres you were off.

Easier to try than to read about. This is the environment itself, which is the same thing a model sees:

Geoguesser Environment

Try Environment | MCP Playground

reset(split=) | reset(index=) · 0 to 3451 | load episode | random episode

train | 0

nd: look around, walk, drop a pin, commit. Runs on a shared Space, so a cold start takes a minute. Open it in a new tab for m

One episode

Figure 2 · Gather evidence, test a candidate on the map, then commit. Every action costs a little reward, and the pin never tells you whether you are close.

Models are bad at this, and bad in a way we found interesting. There is no obvious training set, nobody has optimised against it, and the skill it needs is properly visual and properly sequential. You cannot solve it by thinking harder about one image; you have to go and look at the right thing.

Why this idea and not another

Four things make a task suit an environment. Check them against whatever idea you have before you write any code, because together they decide whether the project is tractable at all.

- The reward is continuous and verifiable. Distance in kilometres from a known coordinate. No rubric, no LLM judge, nothing to argue with. Every guess gets a gradient rather than a pass or a fail, which matters more than it sounds like it should.
- It is multi-turn. Looking, walking and testing a candidate on the map are separate actions with separate costs. A single-shot version of this is a different and much less interesting problem.
- It is not saturated. The best model we measured still misses by a median of 324 km, so there is real headroom to move.
- It is cheap to simulate. The whole environment runs on CPU from a local cache, so a rollout can be debugged for pennies before a GPU is involved.

This one happened to be cheap. Most RL environments are not, and are heavily resource dependent.

How this goes

Here is the order we worked in, and the order we would suggest:

1. Curate the data. Where the imagery comes from and what its licence lets you do. We looked at three sources and two of them fall apart the moment you try to train on them.
2. Build the eval before the training. A small frozen split and a sweep of models that already exist. That tells you the ceiling, how noisy your measurements are, and which model to train.
3. Design the environment. What tools exist, what they cost, and, the part that nearly went wrong here, what they must never reveal.
4. Deploy it. Running it in one place is what lets the trainer and the eval talk to exactly the same thing. OpenEnv also makes it shareable. Push it to the Hub and anyone can try the environment themselves, in a browser, with nothing installed.
5. Design the reward, then fix it. Mine was wrong the first time in a way we only caught by replaying 200 episodes through it.
6. De-risk before you spend. Simulate the rollout with no gradient, then overfit four tasks. Both are cheap, and both catch bugs that look identical to a model that cannot learn.
7. Train, then read the behaviour. Not just the reward curve. The reward went up for a reason we had not intended.

That sequence is not specific to GeoGuessr. The order, and the checks at each stage, are what we would apply to any environment, and only the details change.

What is specific is how much iteration it took. Three configurations, four runs counting the 2B we trained alongside run 2, and they came out very differently on the same environment, the same data and the same base model. Which settings caused that, and why they mattered as much as they did, comes later.

> Everything here is reproducible. The environment is a playable Space, the task splits and 22 GB of imagery are published, both trained adapters are on the Hub, every training run sits in one dashboard, and the code ships with the exact commands. The raw episode records behind every number are published too, so any table in this article can be regenerated rather than taken on trust.
>
> Full inventory in Reproduction.

Data and evals come first

Before writing a line of environment code, the question is where the imagery comes from. Get this wrong and everything downstream is wasted work. We looked at three options.

	look around	walk	bulk cache	licence	verdict
OSV-5M	no	no	yes	CC BY-SA 4.0	0 of 40 probed sequences h
Google Street View	yes	yes	no	ToS-restricted	fine for a live demo, unusabl
Mapillary	yes	yes	yes	CC BY-SA 4.0	the one that survives all four

OSV-5M was the obvious first stop. 5.1 million geolocated street-level images, openly licensed, already a research dataset. It even carries a `sequence` column, so in principle its rows are free seeds for movement. we probed 40 of its sequences to check.

Zero of them contained a spherical image. OSV-5M sampled *perspective* imagery, which is what most of the underlying source actually is. You cannot look around in a perspective photo, so the whole “turn your head” half of the game is impossible. A clean negative result, and worth the hour it took.

The Google Street View API is what the real game uses, and it has far better coverage than anything else. It is also the wrong tool for this. The terms of service do not permit the kind of bulk caching that training needs; you cannot mirror tens of thousands of panoramas to a bucket and read them eight times per step for ten hours. It is usable for a demo or a small held-out eval where you fetch live and discard, and we would treat it that way. It is not usable as a training corpus, and we were not going to build this on a licence we would have to hope nobody checked.

Mapillary is the answer. Community-contributed street-level imagery, CC BY-SA 4.0, with a real API, sequence graphs so movement is possible, and, critically, spherical 360° panoramas among the perspective ones. Faces and plates arrive pre-blurred.

It comes with its own surprises, all of which cost time:

- `camera_type` returns `spherical`, not the `equirectangular` the docs say. Filtering on the documented value matches nothing at all.
- Panorama coverage is thin and clustered. At 45 sampled Street-View coordinates, only 7 had a full 360° panorama. Expect a Europe-weighted distribution, and say so rather than hiding it.
- `/images` search is not a bulk endpoint. `limit` does not cap the scan, and any bbox denser than about ±0.0005° fails with “reduce the amount of data”. Discovery ends up being thousands of tiny bounding boxes.
- Image URLs are expiring signed CDN links. You cannot store them in a task index and fetch later; you resolve to a local cache once and read from disk forever after.

Harvesting, and the split

The pipeline walks Mapillary’s vector coverage tiles, collects candidate sequences, picks tasks with country balancing, and mirrors every frame each task might need. That last part matters. Frames in a sequence sit about 3.3 m apart, so walking down a road touches many images, and all of them have to be local or a rollout dies mid-episode on a network call.

The output is a pool of 3,673 tasks, from which 3,452 go to training and 200 are held out for eval.

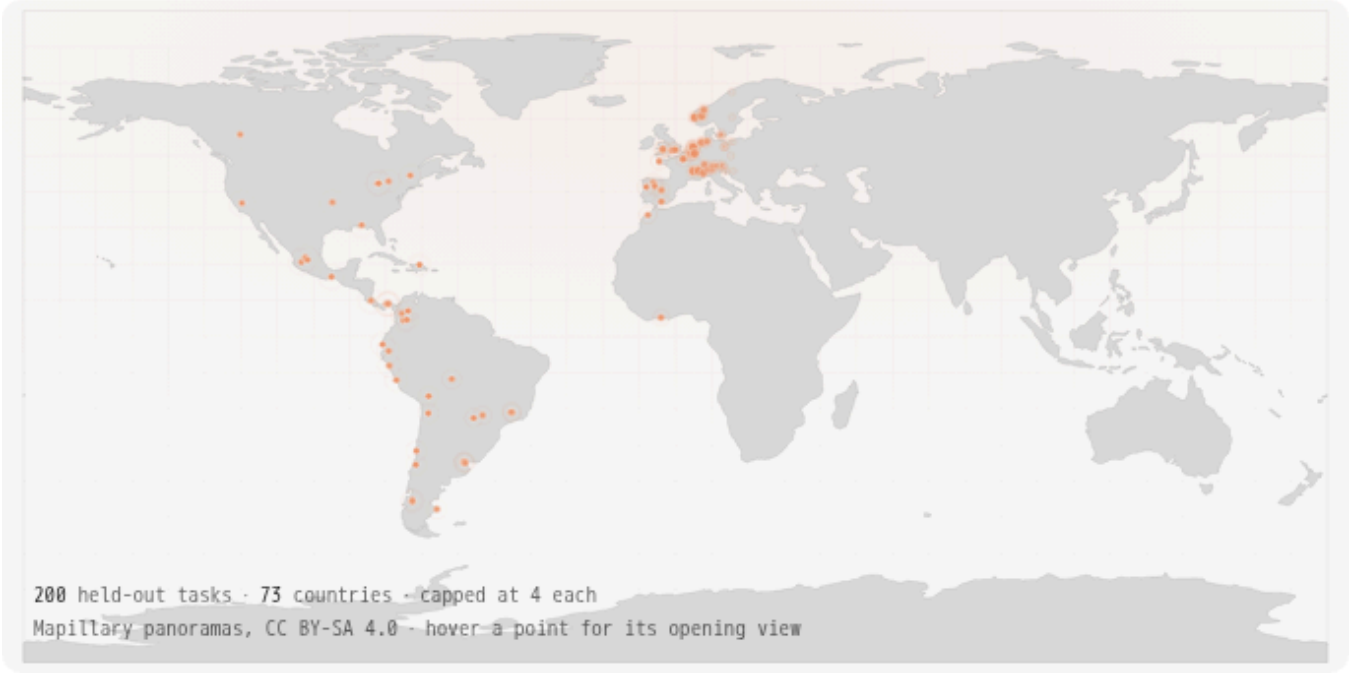
The contamination rules are enforced once, before the split, which is the only place they can be enforced correctly:

- no Mapillary sequence appears in both splits

- no training task sits within 1 km of an eval task

> That second rule we took from the OSV-5M paper, and it is the one people get wrong. At 3.3 m frame spacing, holding out one image while keeping its neighbour holds out nothing. The unit of contamination is the sequence and the neighbourhood, not the image.

These are the 200 locations that ended up in the eval split. Every number in this article is scored on them.



The 200-task eval split is small on purpose. It is small enough to sweep fifteen models across in an afternoon, and frozen so that every number in this article is comparable to every other. Imagery is 22 GB and lives in a Storage Bucket rather than git; the task index is metadata only and is committed, so a change to the benchmark is visible in review.

Sweep the field before you train anything

With an eval split and no trained model, the first useful thing is a baseline sweep. We ran fifteen models across all 200 tasks. Anthropic and OpenAI over their APIs, the larger open models through Hugging Face inference providers, and the small ones self-hosted. For those we spin up a Hugging Face Job serving vLLM, point the harness at it, sweep, and tear it down.

Run pass@k, not pass@1. A single attempt per task on 200 tasks is noisy enough to invent rankings that do not exist. We use pass@4. Four independent passes over the same tasks, each recorded with its own attempt index, pooled afterwards. It costs four times as much and it is worth it. In this project a comparison read at pass@1 said our model had tied Sonnet 5; completed to pass@4, Sonnet moved from 0.6798 to 0.6952 and the tie became a real loss.

Fifteen models, before any training

#	MODEL	REWARD	MEDIAN KM	≤ 25 KM	≤ 200 KM	TURNS	NEVER GUESSED	FABRICATE
1	claude-sonnet-5	0.638	304	29	83	5.6	0	0
2	gpt-5.4-mini	0.446	786	16	36	6.2	0	0
3	Qwen3.5-122B-A10B novita	0.437	654	21	48	9.1	0	7
4	claude-haiku-4-5	0.432	902	9	46	10.3	0	0
5	Qwen3.5-397B-A17B novita	0.338	1,212	10	34	10.9	0	0
6	Qwen3.5-27B deepinfra	0.333	1,154	10	28	10.6	0	4
7	Qwen3.5-9B local	0.332	1,214	8	29	9.5	3	12
8	Qwen3.5-35B-A3B deepinfra	0.332	1,206	6	20	7.7	2	1
9	Qwen3.5-4B local	0.331	1,200	7	21	7.9	1	32
10	Qwen3.5-9B deepinfra	0.311	1,478	6	27	9.4	2	6
11	Qwen3-VL-4B-Instruct local	0.293	1,428	7	19	10.9	0	0
12	Qwen3.5-2B local	0.282	1,989	1	11	8.1	11	26
13	gpt-5.4-nano	0.257	2,104	3	15	10.6	0	0
14	Qwen3-VL-2B-Instruct local	0.253	1,542	4	16	11.8	24	0
15	Qwen3.5-0.8B local	0.159	4,779	0	4	7.4	16	29

15 models, 200 tasks each, 3,000 episodes, scored on the environment's own reward curve. **never guessed** counts episodes that ran out of turns without submitting, which is an instruction-following failure rather than a geography one. **fabricated** counts replies that wrote the environment's own side of the exchange, inventing tool results for turns that never happened. The highlighted row is the model this project went on to train.

Figure 4 · Reward, accuracy at two thresholds, turn count, and the two failure modes that the reward column hides.

> This first sweep is scored on the environment's own reward curve, `exp(-distance_km / 1492.7)` minus a per-action cost, which is a different scale from the pass@4 numbers everywhere else in this article. The Reward section explains why the two exist and what mixing them cost us.

Read the tails, not the mean. Sonnet's 0.638 against second place's 0.446 is a big gap, but the informative number is accuracy. 83 of 200 guesses inside 200 km, against 36 to 48 for everyone else, and 29 inside 25 km. It does that in the fewest turns of any model, 5.6, with almost no pinning. A mean reward can be dragged around by a handful of catastrophic guesses; the threshold columns cannot.

Scale is not predictive here. Ranks 5 through 10 span 9B to 397B-A17B and sit within 0.027 of each other. A 397B mixture-of-experts scores 0.338 against 0.331 for a local 4B. The one large model that separates is 122B-A10B, and its advantage is in the tails again. It has the best median distance of any non-Sonnet model at 654 km, and 21 hits inside 25 km. Whatever this task needs, parameters are not buying much of it, which is exactly the shape of task where RL on a small model has room to work.

The reward column hides two different failures, and they need different fixes, so count them separately.

The first is never guessing. Qwen3-VL-2B-Instruct ran out of turns without submitting in 24 of 200 episodes and scored a hard zero on each. That is an instruction-following failure, not a geography one, and it turns out to be the number we ended up tuning against.

The second is fabrication. Some replies wrote the environment's own side of the exchange, inventing tool results for turns that never happened:

```
-> Pin 2 placed at 48.75, 16.40 ... 20 actions left

```

That happened in 32 of 200 episodes for the local Qwen3.5-4B, and in 26 and 29 for the 2B and 0.8B. It is absent from every Anthropic and OpenAI model, and from both Qwen3-VL models. So it is a property of those particular self-hosted checkpoints rather than of the task.

Which is how you pick the model to train. Here the sweep was picking a fine-tuning target, and we wanted the smallest model with the latent ability, because that is where RL has the most headroom and the cheapest iteration. Qwen3.5-4B sat mid-field at rank 9, used tools properly at 2.3 pins and 7.9 turns per episode, and fits on one GPU with LoRA. Its fabrication count was

the worst of the fifteen, which we read as a formatting problem rather than a capability one, and fixing it accounted for a good part of the eventual gain. When the model is already fixed, the same sweep pays for itself differently. It tells you which slices of your data are hard, and therefore what to harvest more of and what to drop. This is the cheap version of something every large run does, where data mixture decisions are settled by ablations at a small fraction of the final compute rather than learned from the full run.

The environment

The design goal was fidelity to the real game. Eleven tools, and five of them do the work.

tool	what
`look`	Turn the camera to an absolute heading, 0 = true north. This is how it reads signag
`zoom`	Narrow the field of view without turning. 30° reads a distant sign; 90° almost never
`move`	Walk forward or back along the road. Reports how far it *actually* travelled, which is
`place_pin`	Drop a candidate on the world map. Returns what is at that coordinate: country, su
`submit_guess`	Commit. Terminal, scored on distance.

The other six, `pan`, `view_map`, `list_pins`, `clear_pins`, `measure` and `reverse_geocode`, are conveniences over those. `list_pins`, `clear_pins` and `measure` cost nothing, since they only read back what the agent already knows, but every call still spends one of its twelve turns.

`look` is the one that carries the task. Turning the camera is how a policy reads signage, notices which side of the road the traffic is on, and finds the utility pole whose spacing narrows the country down.

One episode, tool call by tool call

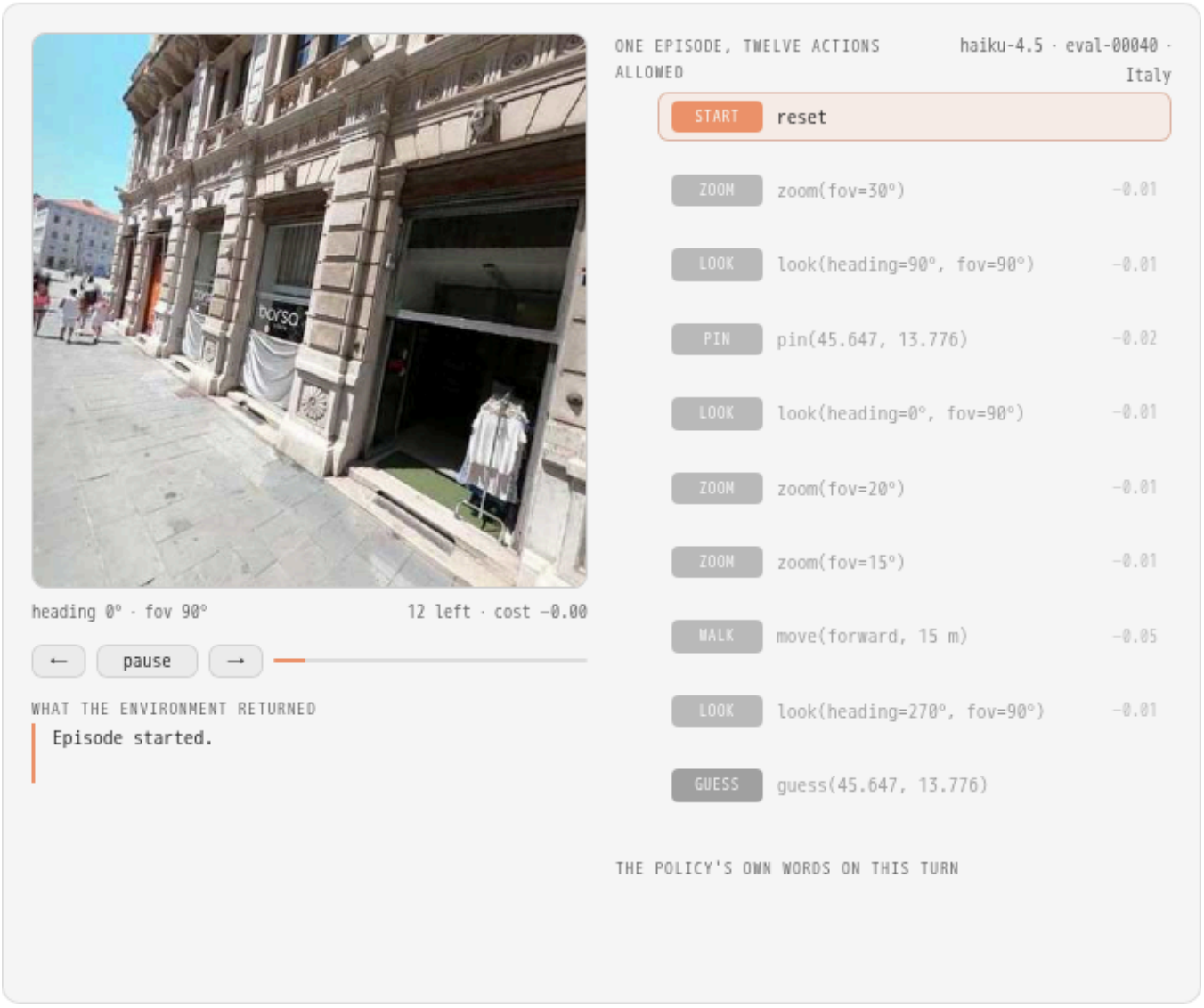


Figure 5 · A real recorded episode replayed from its trace. Every frame is the reprojection the model was actually served, verified against the hash in the record. The left panel is the observation, the right is the call that produced it and the reply the policy wrote.

This is `claude-haiku-4.5` instead of the model we trained, because it uses every tool in one episode: zoom in, turn, pin a candidate, zoom further, walk down the road, turn again, commit. Two details show the environment doing its job. It asked to move 15 m and moved 2, because movement is bounded by where frames actually exist. And the pin at turn 3 came back with the country and the nearest city and nothing about whether that guess was any good. My own trained model's episodes look nothing like this by the end of training, which is what "What it actually learned" is about.

The trap in the pin loop

`place_pin` is where the design could have gone badly wrong. It tells the agent what is at a coordinate. It says nothing about whether that coordinate is close to the truth.

If pin feedback carried any signal about the target, whether a distance, a warmer/colder, or a highlighted region, then the optimal policy is binary search. About twenty pins gets you to metre-level accuracy, and the environment would be measuring bisection rather than geography. So distance and score arrive only from `submit_guess`, and only once.

The same reasoning constrains the map. Detail is a function of zoom alone, never of proximity to the answer. Prefetching high-resolution map tiles around task locations would quietly turn the cache into a ground-truth oracle.

Movement is bounded by what was actually captured, so there are dead ends, and frame spacing varies from 3 m to 57 m between sequences. `submit_guess` is terminal, so an episode is exactly one guess. The pin loop is how a policy checks its own arithmetic before committing.

Why OpenEnv

We built this on OpenEnv. “We used a framework” is usually a throwaway line, so here is what it actually bought.

What we wrote is a Python class with three methods and a set of tools. `reset()` returns the first observation for a `(split, index)`, `step()` applies one typed action, `state` reports where the episode is. The tools are registered on a `FastMCP` server inside the class, the action and observation are Pydantic models, and there is a twelve-line `openenv.yaml` naming the entrypoint and the Space variables. That is the contract.

Everything a trainer actually connects to is generated from it.

One class in, two API surfaces out

Figure 6 · Orange is what create_app() derives from the environment class. The trainer drives the simulation over HTTP, the policy only ever sees tools over MCP, and in production mode the simulation routes are never registered at all.

Every item on that list is something you would otherwise write yourself and get subtly wrong.

The schemas are the API. `Action` and `Observation` are Pydantic models, so `GET /schema` hands a caller the JSON schema for actions, observations and state. My harness derives the tool definitions it gives the model from that endpoint rather than from a copy pasted list, which is how a tool we added mid-project reached the policy with no change to the harness.

Per-session isolation. The server holds an environment *factory*, not an instance, and builds a fresh one per connected session up to `max_concurrent_envs`, each with its own thread pool so a slow synchronous `step()` cannot stall the others. An environment opts in with one class attribute, `SUPPORTS_CONCURRENT_SESSIONS = True`, and the server refuses a cap above one if you have not. Eight rollouts of a GRPO group are eight sessions against one process, and that is a configuration line rather than code we had to write.

Task selection as routes. Declare `list_splits`, `num_tasks`, `get_task` and `get_task_range` on the class and the server exposes them as routes under `/geoguesser_env/…`; leave them out and those routes return `501`. So a trainer can ask a URL how many tasks it has and build a dataset from the answer, without importing anything of ours. That is what makes `--split eval --index 7` mean the same thing to our eval harness, to TRL, and to a stranger with `curl`.

The server enforces the two surfaces. `/reset`, `/step` and `/state` are registered only when the server runs in simulation mode. The agent's surface is `/mcp`, which carries tools and nothing else. So the invariant we care about most, that a policy cannot reroll a task it dislikes, is structural. In production mode the route does not exist. we did not have to trust a prompt for it.

One environment, several backends. The task, reward, parser, map renderer and pin loop do not care where imagery comes from. Changing the source changes *which tools exist*, so a capability gap shows up as an unregistered tool instead of as a different observation schema. A tool that would always fail is simply not offered, because registering an erroring tool just teaches a policy to burn its step budget.

A determinism contract. `reset(split="eval", index=7)` returns a byte-identical observation every time, from any process. A GRPO group is N rollouts of the *same task*, so if the starting observation drifts between them the advantage is measuring noise. Three things make it hold: panorama bytes come from local disk instead of an expiring URL, reprojection is pure numpy with integer sampling, and the initial heading is pinned to the panorama's own `compass_angle` so `look(0)` is true north in every task.

The last one is ours to guarantee. OpenEnv makes `reset(split, index)` addressable; making it return the same bytes every time is on us.

Deploying and sharing it

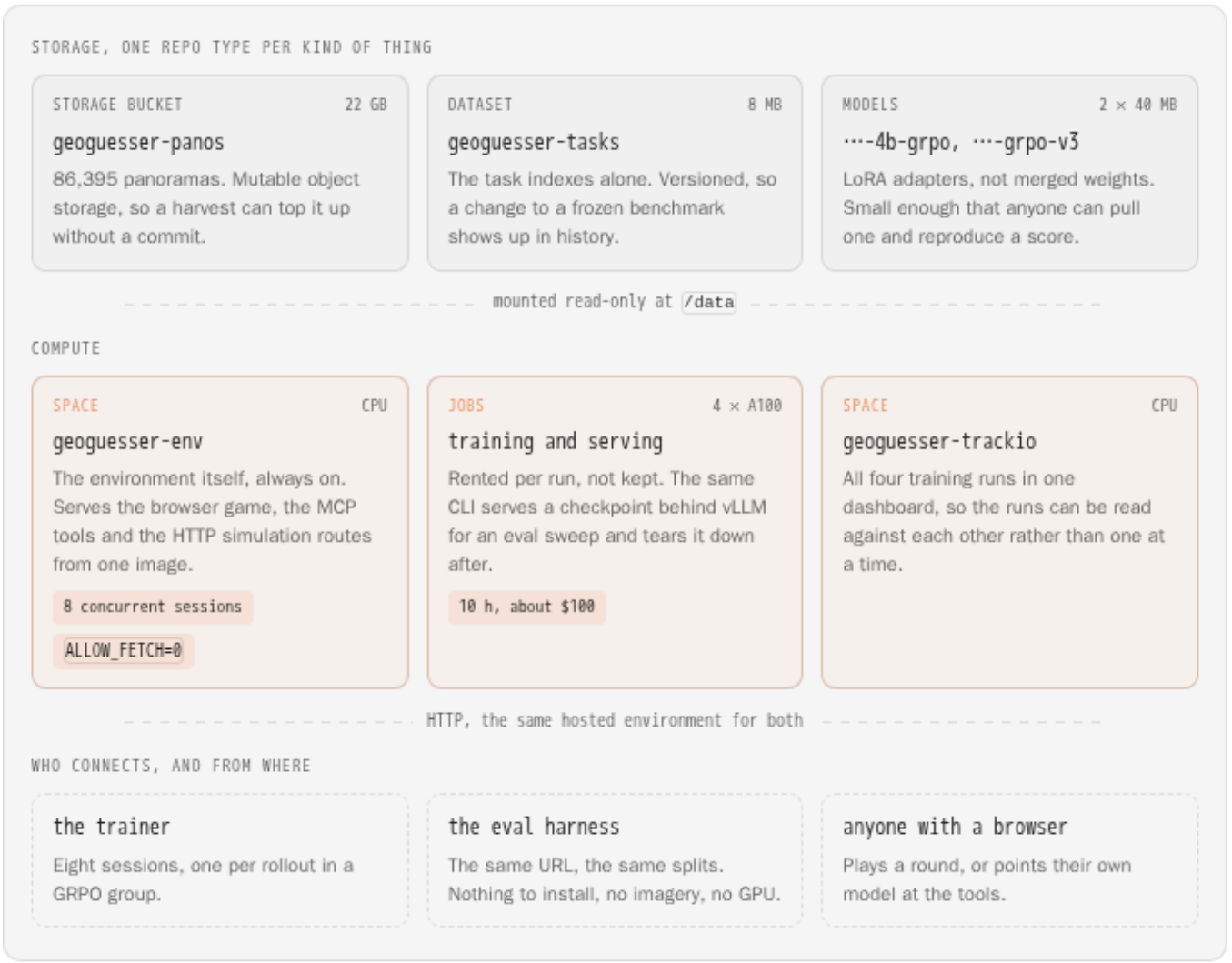
An environment nobody else can run is a private result. A number someone else can regenerate can be checked; a number nobody else can reach cannot. The Hugging Face Hub is the best place we know to fix that, because the environment, the data, the trained adapters and the training curves can all sit next to each other and all be pulled by a stranger holding one URL.

Shipping it also removes a way to fool yourself. If the trainer and the eval harness both talk to one hosted environment over HTTP, neither can be measuring against a subtly different thing than the other.

Let the Hub hold the parts it is good at

Hub infrastructure has a repo type per kind of artifact, and they compose. A bucket for bulk bytes, a dataset for anything that must be versioned, a Space for anything that must be running, model repos for weights, Jobs for GPUs you rent by the hour. One token addresses all of it.

Where each piece lives, and what talks to what



The property worth designing for is the middle row reading the top row the same way everywhere. Locally the indexes and imagery sit in the checkout; on the Space they arrive through the mount. Splits, step budget, street labels and offline enforcement are identical, and checked: the same task returns the same image checksum, reward and distance from either.

Figure 7 · Storage in a Bucket and a Dataset, the running environment on a Space with the bucket mounted read-only, GPUs rented per run through Jobs, and one HTTP URL that the trainer, the eval harness and a browser all reach.

How we used each of them, and why:

what	where it lives	
22 GB of panoramas	Storage Bucket	Space disk is ephemeral and capped well below 22 GB. A bu
the task indexes	dataset repo	8 MB of metadata, and the part that has to be *frozen*. A buc
the environment	Space	One image serves the browser game, the MCP tools and the
the adapters	model repos	Both are LoRA adapters instead of merged weights, so they
the GPUs	Jobs	Rented per run instead of keeping them: ten hours on four A
the curves	Trackio Space	Every run in one dashboard, which is what makes them reac

```
export HF_TOKEN=hf_...
python dataset/deploy_hub.py --all      # bucket, dataset and Space
python dataset/deploy_hub.py --space    # code only, fast iteration
python dataset/deploy_hub.py --verify   # check a live deployment
```

Parity between local and hosted is checked rather than assumed. The same task returns the same image checksum, the same reward and the same distance from either. Part of what makes that true is that the Space runs with `GEOGUESSER_ALLOW_FETCH=0`, so a missing frame raises instead of quietly reaching out to Mapillary. The store is complete by construction, so a miss means the mount is wrong, and you want to hear about that immediately rather than after six hours of a run silently going over the network.

> `openenv push` is deliberately not used here. It cannot attach a bucket volume, and its default excludes would have uploaded 22 GB of panoramas into git.

Once it is up, anyone can play the environment in a browser, and any trainer or eval harness can reach it over HTTP with no checkout, no imagery and no GPU.

The reward

The reward is where a project like this is won or lost, because it is the only thing the model is actually told. The tools, the imagery and the prompt decide what it *can* do. The reward decides what it ends up doing.

It is also the part you should plan to write more than once. Mine went through two versions and the second one is not obviously better on paper; it is better because we could see what the first one did to 200 real episodes. Reward design is empirical. What worked was to pick the obvious curve, run it over recorded episodes, and look at what it rewards and what it cannot distinguish.

The obvious curve here is the game's own. Distance in, score out:

$$\text{game} = \exp\left(-\frac{d}{1492.7}\right), \qquad \text{reward} = \max\bigl(0,\ \text{game} - \text{cost}\bigr)$$

with a charge per action, `0.01` per look or map, `0.02` per pin, `0.05` per move, so the policy has to decide when it has seen enough. Continuous, unambiguous, no judge to game. As a leaderboard score it is fine.

What the trainer optimises is slightly different, and the difference is the interesting part.

$$r = \min\Bigl(1,\ 0.5\,e^{-d/1492.7} + 0.5\,e^{-d/5000}\Bigr) \times \bigl(1 - \min(\text{cost},\ 0.2)\bigr)$$

Two changes. A second, much slower decay scale, and a cost that multiplies rather than subtracts. Both look like fussy details and both change what is learnable. It is easier to see than to read, so this is the same arithmetic you can poke at.

What a guess is worth, and why the two curves disagree

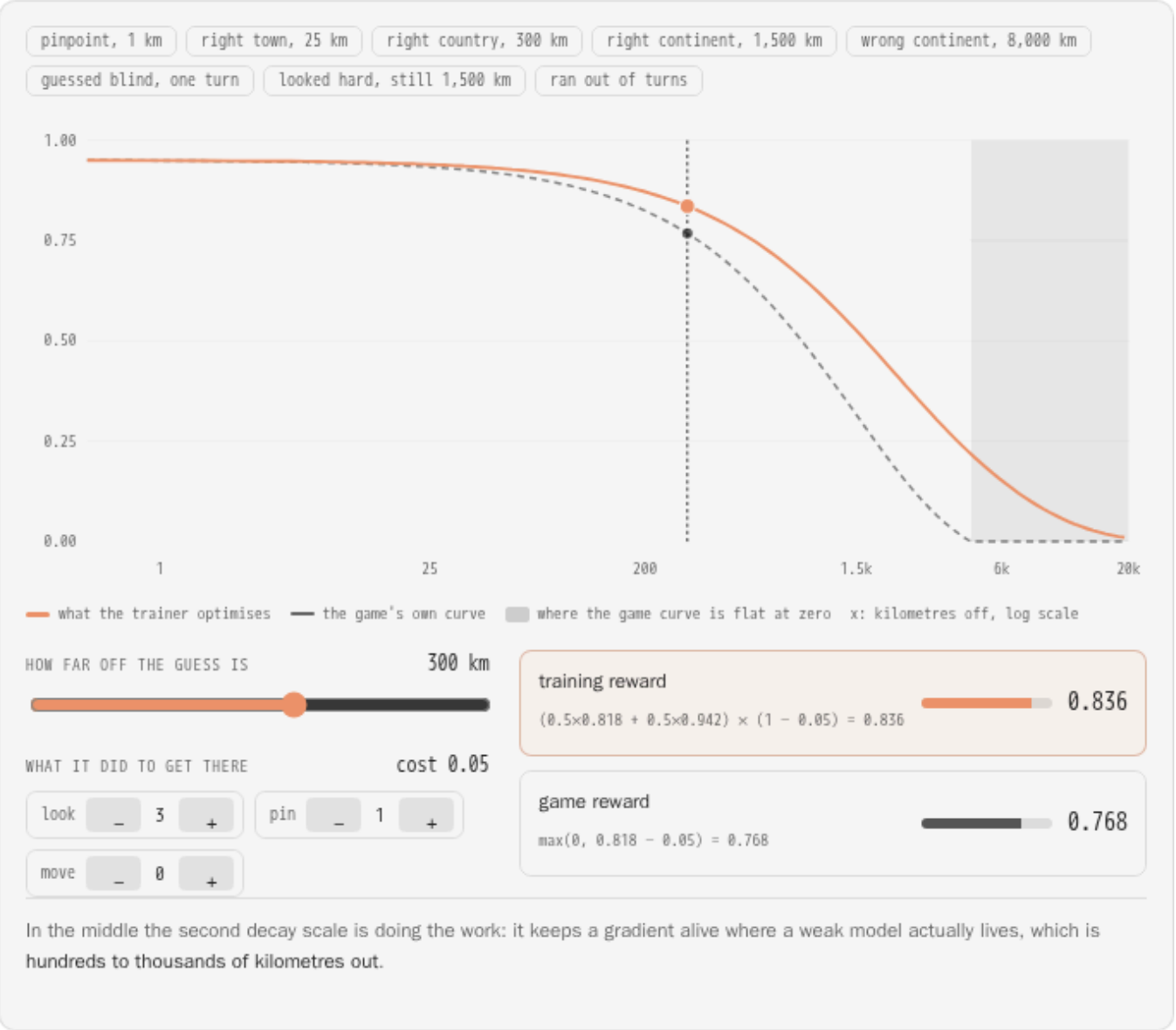


Figure 8 · Pick an episode or move the sliders. The dashed line is the game's curve, the solid one is what the trainer optimises, and the shaded band is where the game curve has already flattened to zero.

The single exponential goes flat where a weak model lives. `exp(-d/1492.7)` is worth 0.018 across the *entire* range from 6,000 to 20,000 km. About a third of an untrained 4B's guesses land there, so getting dramatically less wrong on the wrong continent earns almost nothing. The 5,000 km scale makes that same span worth 0.150. It is telling the model it is getting warmer at the exact point where it most needs to hear it.

Subtracting the cost flattens the far misses into each other. With `max(0, score - cost)` a guess past the cliff clamps to exactly zero, and where that cliff falls depends on what the episode spent: about 4,500 km for a cheap episode, about 3,300 km at the average cost of these models.

> Replaying 200 episodes through the game curve floored 77 of them at exactly 0.0, with no variance between them. A 3,324 km miss scored the same as an 18,723 km miss. A GRPO group drawn from those tasks has zero advantage and teaches nothing, and more than a third of the split was in that state. Multiplying by a positive factor cannot collapse an ordering, which is the whole reason for the change.

One property we would keep in any version of this reward is that any guess anywhere on Earth beats no guess. Running out of turns without answering is the worst outcome, because for a small model it is by far the most common failure, and a reward that leaves it ambiguous will not fix it.

> There are now two reward scales in this project, and on the same guess they differ by more than 10×. The stored episode reward is the environment's curve, while every number we report is recomputed from raw distance through the training curve. Mixing them made run 2's deltas incomparable to run 1's for a while, and a unit test now fails if the two definitions drift apart.

Checks before the run

Three cheap checks before you rent a GPU. Each one has saved us a full run's cost, and none of them needs a GPU.

Being systematic about this pays off because RL failures do not announce themselves. A broken reward, a wrong turn budget and a model that simply cannot do the task all produce the same flat line at zero. You cannot tell them apart from the loss curve, so you have to rule them out beforehand.

First, prove the environment can take the load

If you are training against a hosted environment, open as many concurrent sessions as the run will need, before the run.

`MAX_CONCURRENT_ENVS` defaults to 4. Four training ranks plus generation needs 8. Worse, `duplicate_space` copies a Space's files but *not* its variables, so a duplicate looks correct and

is not. A run pointed at one of those dies about an hour in with `CAPACITY_REACHED: 4/4 sessions active`.

```
from geoguesser_env.client import GeoGuesserEnv
envs = [GeoGuesserEnv(base_url="https://<your-space>.hf.space") for _ in range(8)]
for e in envs:
    e.reset()
print("8/8 concurrent sessions OK")
```

Skipping those six lines cost us a run.

Then simulate the rollout with no gradient

Everything except the backward pass runs on a laptop against any served model with tool calling. Derive the tool schemas the way TRL does, call `reset` with a dataset row, let the model drive the loop, read the reward, and render the conversation through the real chat template to check that images survive a tool result.

This found three bugs that would each have wasted a run. Every one of them produced a reward of exactly 0.0, which is indistinguishable from a model that cannot learn.

bug	what it looked like
two turn budgets disagreeing	0 of 6 episodes ever reached an answer
`done` read from the observation instead of the step result	every episode unterminated, no score ca
a tool omitting a parameter the environment supports	a `TypeError`, a wasted turn, a capabilit

> The first one deserves expanding, because no traceback tells you about it. The environment counted down its own budget of 24 turns while the trainer stopped the rollout at 12. The model believed the live counter in the tool results, paced itself for 24, and was cut off before it ever committed. Every reward exactly zero, no advantage in any group, and from the loss curve it looks precisely like a model that cannot learn the task. Show the model exactly one turn budget.

And overfit four tasks

This is the check we would push hardest on. Take two to four tasks, run many epochs on just those, and watch the reward.

If it does not climb to near the ceiling on four tasks it has seen a hundred times, then one of three things is true: the model cannot represent the task, the reward is broken, or the plumbing is broken. Learning that for a few dollars beats learning it in hour nine. The rollout simulator takes a `--same-task` flag for this, so the group it builds is a real GRPO group over one task.

A healthy first step on real hardware looks like this:

```
reward                 0.1979 -> 0.2123
reward_std             0.1905 -> 0.2858
frac_reward_zero_std   0           <- no group with zero advantage
tools/call_frequency   11
tools/failure_frequency 0
step_time              156 s

```

Watch `frac_reward_zero_std`, not reward. GRPO's advantage is the spread *within* a group, so if that fraction is near 1 then most groups are teaching nothing and no amount of training will fix it. And do not read a reward increase over two steps as learning. At temperature 1.0 that is noise.

> This is not only a small-run habit. Mercor and the SkyRL team describe the same sequence for a 397B run in their RL training guide: validate the harness first by running an eval pass over the full training set *at the concurrency RL will use*, drive the non-model error rate as close to zero as you can, then do an overfitting run before the real one. Their phrasing is blunter than ours: if you cannot overfit a handful of tasks, an end-to-end run has no chance. Worth reading before you spend real money, and reassuring that the checks barely change across three orders of magnitude of scale.

Training

Everything so far has been setup, and that is roughly the right proportion. The training itself is the smallest part of this project by effort and by code.

The trainer is TRL's `GRPOTrainer`, and the piece you have to write is the link between it and the environment. TRL keeps that small. `GRPOTrainer(environment_factory=...)` is a real multi-turn tool-calling loop, so you hand it a factory that returns an environment instance and it runs the episodes for you. No custom rollout function.

Trainer to environment

Figure 9 · Where the group is formed, where the reward is computed, and which side holds the websocket.

A few things about that interface we had to read off the source rather than the docs.

- Every public method of your environment class becomes a tool. Underscore-prefix anything that is not one, and `get_reward` has to be a method rather than a property.

- Tool schemas come from your docstrings via `transformers.utils.get_json_schema`, which wants plain Google style and a type hint on every argument.
- `reset(**row)` receives the whole dataset row, which is how every rollout in a group ends up on the same task.
- Tool results can be multimodal. Returning an image block works and the processor picks it up, which is what makes a visual environment possible at all.
- `max_completion_length` bounds the entire rollout, images and tool results included, not one turn. Set it too small and TRL quietly drops an oversized tool result and leaves the loop, which looks like episodes that end early with no answer and reward 0.

We pointed the trainer at the environment hosted on a Space rather than running it in-process, because that is the same thing the eval talks to. Whatever else is wrong, we are not training against one environment and measuring against another. The capacity check from the previous section is what makes that safe.

Before you start, send every run to one dashboard. We use Trackio, which is a Space plus a bucket, so the runs outlive the Jobs that produced them and sit on a shared axis where they can actually be compared. Reading three runs in three tabs is how you end up believing things that are not true. The whole dashboard is embedded at the end of this article, and every training figure below is built from its database.

The first run

Run 1, from the untrained baseline

Figure 10 · Per-step training reward with a 50-step moving average, and the held-out eval score of each checkpoint on the same axis. Both start from the untrained baseline of 0.4825.

The two series measure different things. The noisy one is the training reward, one task per optimizer step at temperature 1.0, which is why it swings between 0.01 and 0.99 and why only the moving average tells you anything. Smoothed, it goes from 0.20 to 0.64. The other is the eval score, mean-of-4 over the 200 held-out tasks, run separately for eight checkpoints.

The eval line is what we trust, and it says 0.4825 to 0.6445, a gain of +0.162. It also says something less flattering. By step 200 it is at 0.6393, and the remaining 800 steps move it by 0.005. We kept the run going anyway, to watch the behaviour rather than the score, and that turned out to be the right call for reasons that have nothing to do with the number.

Watch the readout while it plays. Turns per episode fall from 6.7 to 1.1, output tokens from 1,062 to 66, and entropy from 0.73 to 0.41. The model is getting *faster* rather than more careful, and the score goes up anyway.

	value
model	`Qwen/Qwen3.5-4B`, LoRA r=16, α=32, dropout 0.05, on `q/k/v/o_proj`
hardware	4×A100 80 GB, DDP under `torchrun`
steps	1000, at 1 task per step
rollouts	`NUM_GENERATIONS=8`, `ACCUM=2`, `per_device_batch=1`, so 8 episodes per step in one
episodes	8,000 total, touching 1,000 tasks, 29% of the split
turns	12
image	448 px
optimiser	LR 3e-5, temperature 1.0, `beta=0` (no KL anchor), `scale_rewards="group"`
wall clock	10.2 hours
cost	about $102 of training, about $25 of evals

You can run all of this on one GPU. The training script defaults to `NPROC=1`, so a single A100 runs the loop end to end, environment, generation, reward and update included. Run 1 used four cards purely for throughput, via DDP, and nothing about the recipe needs them. One card takes proportionally longer and costs proportionally less, which is the right trade while you are still finding out whether your reward works.

```
hf jobs uv run \
  --flavor a100-large --timeout 12h --image huggingface/trl --secrets HF_TOKEN \
  -v hf://buckets/<you>/geoguesser-runs:/outputs \
  -e OUTPUT_ROOT=/outputs -e MODEL=Qwen/Qwen3.5-4B \
  -e ENV_URL=https://<your-space>.hf.space \
  -e MAX_STEPS=250 -e SAVE_STEPS=25 -e MAX_COMPLETION=6144 \
  grpo_geoguesser.py
```

That is one card, one command, checkpoints landing in a bucket. Going to four is `--flavor a100x4` plus `-e NPROC=4`, and the script relaunches itself under `torchrun`, because the Jobs runner wants a plain `.py` entrypoint. `REPRODUCE.md` has the full argument list for each run.

Stop at step 250 if you are reproducing this, by the way. The original went to 1000 and the last 750 steps bought nothing measurable for about $70, which is the subject of the next section.

LoRA is doing a lot of work in making that fit. At r=16 on four projections there are very few trainable parameters, which is why a 4B model and a colocated vLLM share one card without trouble.

> One memory trap, because it killed a later run at step 29. `max_completion_length` sizes a logits tensor at `length × 248,064 vocab × 4 bytes`, and it allocates that whatever the model actually emits. At 12,288 that reserves 11.4 GiB every step while our longest output was 2,114 tokens. Use 6,144. Qwen3.5's vocabulary is what makes a "small" model expensive here, not its parameter count.

What it actually learned

The model stopped exploring.

Behaviour across checkpoints

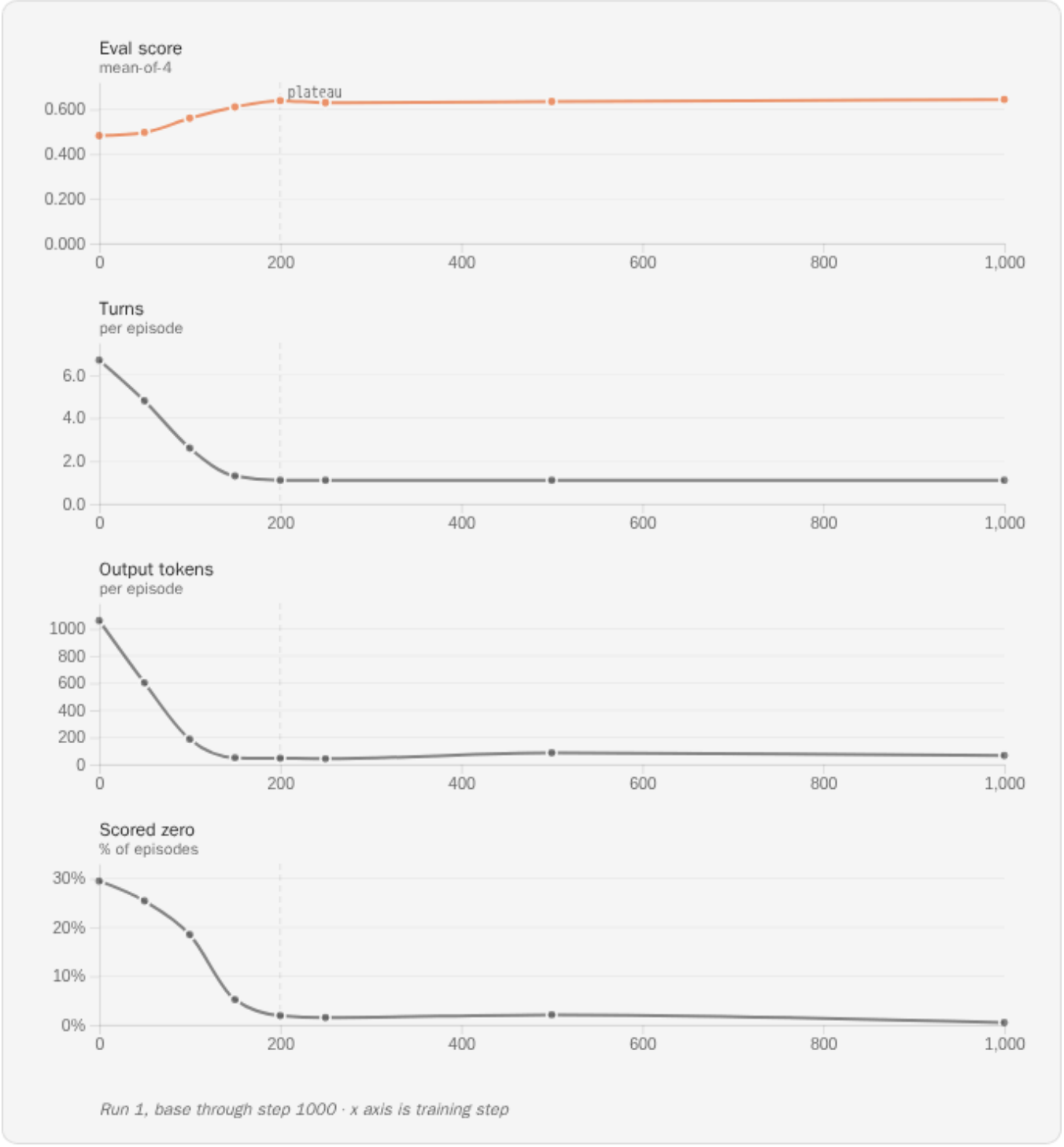


Figure 11 · Turns, output tokens and zero-scoring episodes all fall together while accuracy improves. Score plateaus by step 200.

> Pro tip: serve every checkpoint from one endpoint. Training leaves you with a series of LoRA adapters instead of a series of models, and vLLM will hold a base model plus up to eight adapters at the same time behind `--enable-lora`. Each adapter is addressable as its own model name, so a sweep across eight checkpoints costs one GPU-hour instead of eight. Two things to check. The adapters have to have been trained on the base you are serving, since vLLM will happily load a 2B adapter onto a 4B base and answer anyway, and it is worth asking `/v1/models` what is actually loaded before you record a single score.

checkpoint	eval score	turns	looks	pins	output tokens	scored zero	median error
base	0.4825	6.7	2.2	2.1	1062	29.5%	1226 km
step 50	0.4968	4.8	1.5	1.7	602	25.4%	1308 km
step 100	0.5600	2.6	0.6	0.9	186	18.5%	964 km
step 150	0.6113	1.3	0.0	0.2	50	5.2%	714 km
step 200	0.6393	1.1	0.0	0.1	46	1.9%	675 km
step 500	0.6350	1.1	0.0	0.1	86	2.1%	709 km
step 1000	0.6445	1.1	0.0	0.1	66	0.5%	662 km

Turns fell from 6.7 to 1.1. Looks went to zero. Output tokens dropped from about a thousand to sixty. By step 150 the policy had essentially stopped using the environment at all.

And its accuracy went *up*. Median error nearly halved, 1226 km to 662 km, country identification rose from 21% to 33%, and the share of episodes scoring exactly zero fell from 29.5% to 0.5%.

That last column was wrong in our own write-up for a while. Those are not episodes where the model ran out of turns without answering. Non-submission is essentially absent from this sweep. They are episodes where it *did* answer and landed past the point where the game curve goes flat, which is about 4,500 km for a cheap episode and about 3,300 km at the average cost of these models. That was almost a third of the untrained model's guesses, and one in two hundred of the trained model's.

SO IT IS JUST GUESSING A RANDOM POINT?

This is the first thing everyone asks, and the collapse in output tokens makes it a fair question. The answer is no, and the evidence is in the same table.

A random guess on land averages something like 8,000 km of error. This model's median is 662 km. It names the right country a third of the time, up from a fifth. It puts a fifth of its guesses inside 200 km of the truth. Guesses beyond the cliff, the ones a coin flip would produce, drop from 29.5% of episodes to 0.5%. None of that is what random looks like.

What it is doing is recognising the region from the first frame and committing to a plausible city in it. You can see it in the raw traces. The same coordinates recur across independent rollouts, 41.9028/12.4964 for Italian scenes, 60.1282/24.78 for Scandinavian ones. That is recall of where cities are, applied to a visual prior about what a place looks like. It is one glance and one decision rather than a chain of deductions, but the glance is doing real work.

The uncomfortable part is that deliberation was making a 4B model *worse*, and the training found that out before we did.

> One qualification we owe you on the turn count. Training drives the environment through native tool calls and keeps the whole image history in context; the eval harness sends JSON in text and shows only the current image. Every arm on the board is measured the same way, so the comparison is fair, but the 1.1 turns is measured under a protocol the checkpoint never trained on. The behaviour is real, the exact number is protocol-dependent.

WAS IT REWARD HACKING?

When a policy stops doing the task and scores better, the first assumption should be reward hacking. We spent a day trying to find the exploit. The candidates were the pin loop leaking target information, the task identity being visible in an observation, prompt or metadata leaking the answer, and plain memorisation of the training split.

None of them held up. The pin loop returns only what is at the pinned coordinate. Task identity is hidden. And the memorisation story does not survive arithmetic. A full 1000-step run at one task per step touches 1,000 of 3,452 tasks, one group each, never repeated. The eval split shares no sequence with training and sits at least 1 km away from any training task. On top of that, the gains are measured on 200 held-out tasks it has never seen.

WHERE THE REWARD GOES FLAT

What actually happened is simpler and more interesting. Past that flat point everything scores exactly zero, and that was happening to nearly a third of untrained episodes. So the largest win available to the policy was not being catastrophically wrong, and the fastest route to that was to

stop reasoning itself into a different continent and commit to its first instinct instead. The gain is that plus a sharper prior about where street scenes look like they are.

Turns against score, every model on the board

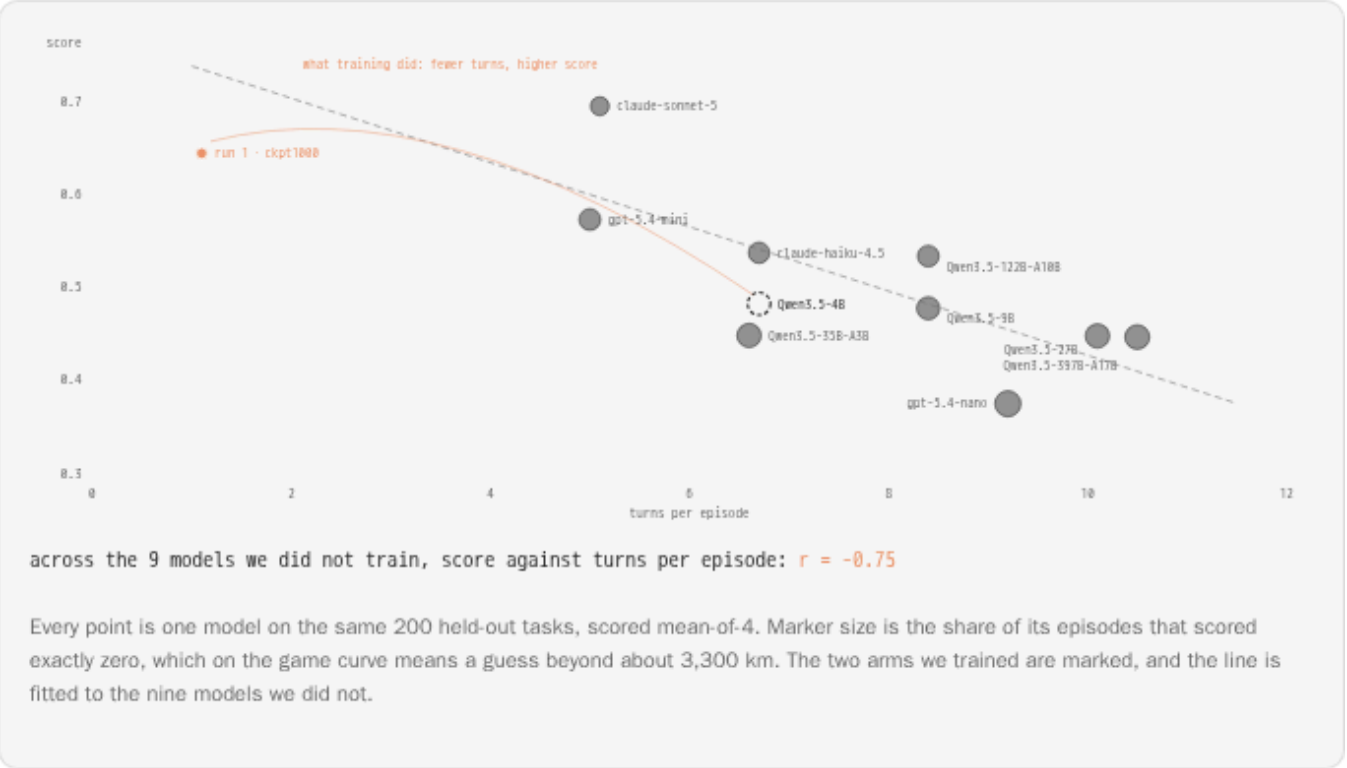


Figure 12 · Across the nine models we did not train, more turns go with a lower score, r = −0.75. Marker size is the share of episodes that scored zero.

The effect generalises past our own checkpoints, which is what points at the reward rather than the training. Across the nine off-the-shelf models we never touched, score correlates with turns per episode at r = −0.75. The ones that deliberate longest do worst. It also correlates with the zero-scoring share at r = −0.96, but that one is partly the same measurement twice, since both are functions of the same distances. Models that commit early score higher, whether we trained them or not.

> We set out to build a model that reads road signs and we trained one that recalls a plausible city and commits. It is more accurate and it is not cheating. It is also not the visual agent we were aiming for, and the reason is the reward.
>
> The direction is counter-intuitive. Run 1 carried the *largest* action cost of the three configurations and collapsed hardest. A higher cost buys a faster commit, not more careful looking. Making evidence-gathering worthwhile needs a *lower* action cost, plus a curve with no cliff at all, so that a careful episode landing 4,000 km out still scores better than a careless one landing 12,000 km out. Under the game curve those two are identical, and a policy cannot learn from a distinction the reward does not make.

WHERE IT LANDED

All arms on the same 200 held-out tasks, pass@4, 800 episodes each, scored through the same curve.

The finished board

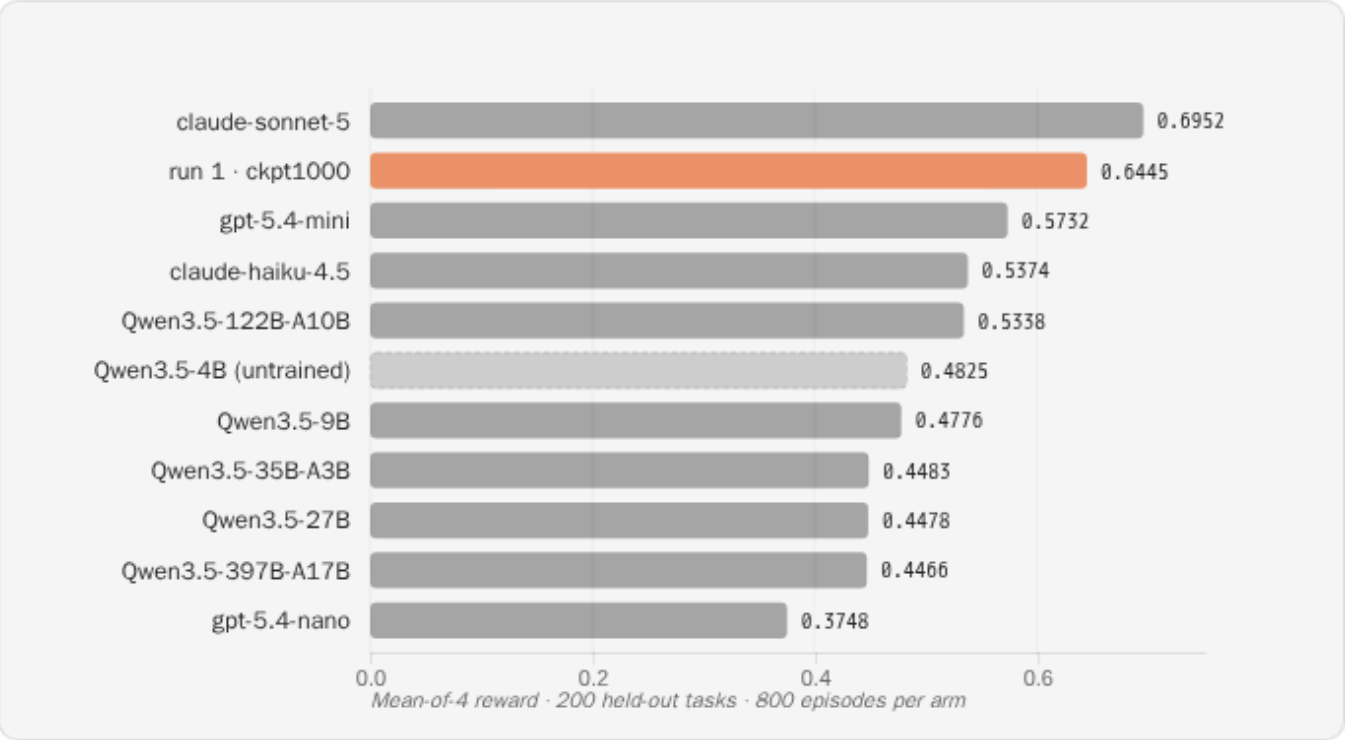


Figure 13 · Mean-of-4 on 200 held-out tasks, 800 episodes per arm. The trained 4B is second of eleven.

model	mean-of-4	median error
claude-sonnet-5	0.6952	324 km
run 1, step 1000 (Qwen3.5-4B + LoRA)	0.6445	662 km
gpt-5.4-mini	0.5732	753 km
claude-haiku-4.5	0.5374	939 km
Qwen3.5-122B-A10B	0.5338	767 km
Qwen3.5-4B, untrained	*0.4825*	*1226 km*
Qwen3.5-9B	0.4776	1203 km
Qwen3.5-35B-A3B	0.4483	1485 km
Qwen3.5-27B	0.4478	1289 km
Qwen3.5-397B-A17B	0.4466	1420 km
gpt-5.4-nano	0.3748	2541 km

Second of eleven. It beats `gpt-5.4-mini`, `haiku-4.5`, and every Qwen3.5 up to 397B, a model roughly a hundred times its size. It loses to `claude-sonnet-5` by 0.0508.

The improvement over its own base is +0.1620, paired per task, 95% CI ±0.0137, better on 169 of 200 tasks. Ten hours and about $100.

> The same table carries one caveat. The run plateaued by step 200. Step 200 scores 0.6393 against step 1000's 0.6445, which is inside the noise. The remaining 800 steps span a 0.013 band across nineteen checkpoints and bought nothing but about $70 of A100. If you reproduce this, stop at 250.

The second and third runs

Run 1 worked, and one working run is a story, not a result. So we ran two more, changing as little as possible each time, to find out *why* it worked.

Run 1's training dynamics looked bad. Entropy fell from 0.73 to 0.41, the spread inside each group went to almost nothing, and grad norm, which sat at 0.18 early on, peaked above 11.

Twice during that run the sensible recommendation was to kill it. Both times that would have been wrong, and working out why taught us more than anything else in the project.

Three runs, one knob at a time

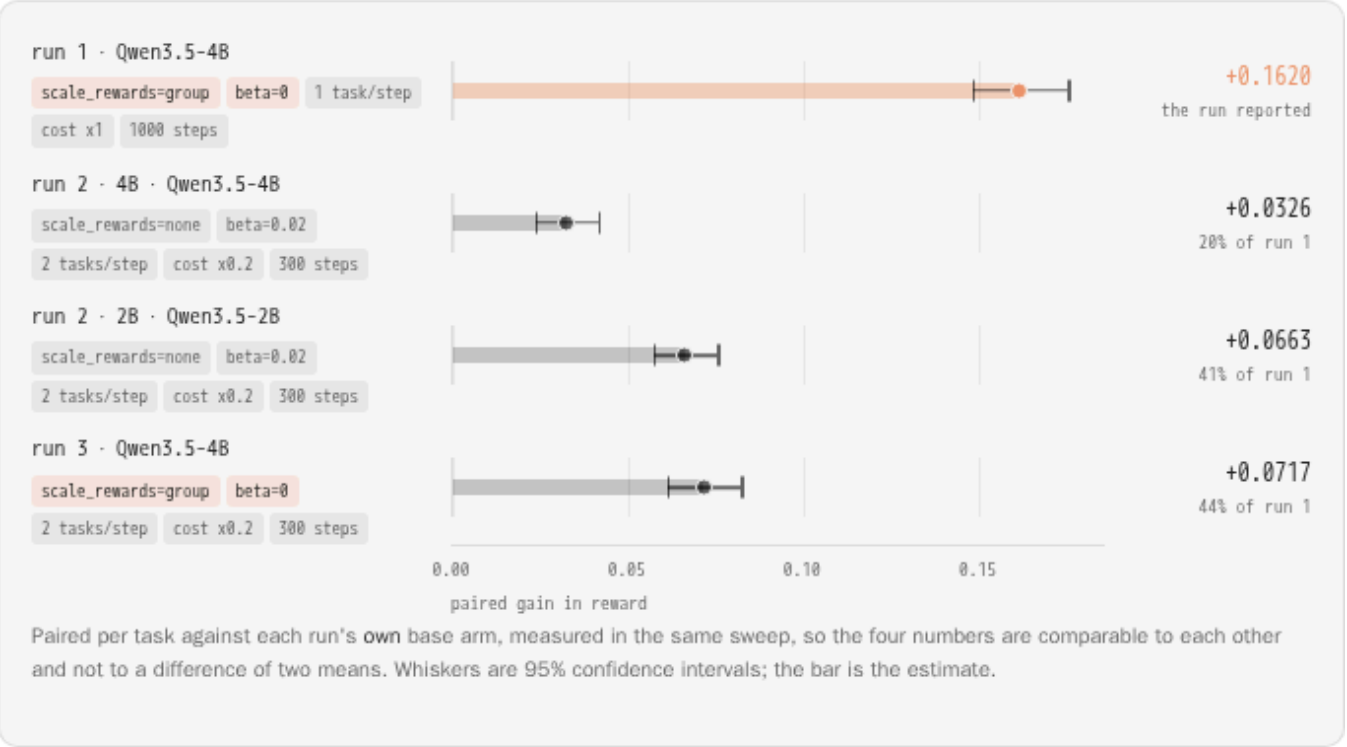


Figure 14 · Paired gain against each run's own base, with 95% confidence intervals. Run 2 damped the instability and lost most of the gain.

Run 2 was designed to suppress exactly those dynamics. We turned off advantage amplification (`scale_rewards="none"`, i.e. Dr.GRPO), added a KL anchor (`beta=0.02`), and doubled gradient accumulation to two tasks per step. It trained cleanly and gained +0.0326, about a fifth of run 1. we also trained a 2B alongside it on the same config, which gained +0.0663, so the recipe does transfer down.

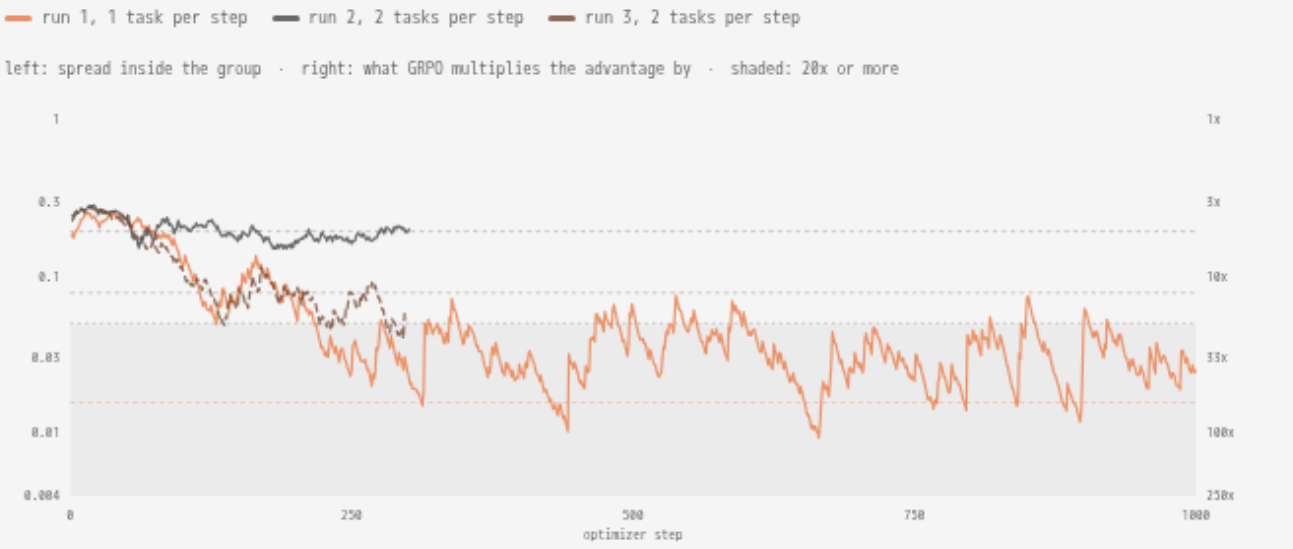
Run 3 reverted exactly those two knobs, `scale_rewards` back to `group` and `beta` back to 0, keeping everything else from run 2. It gained +0.0717, recovering roughly 44%.

	run 1	run 2 · 4B	run 2 · 2B	run 3
`scale_rewards`	`group`	`none`	`none`	`group`
`beta`	0	0.02	0.02	0
tasks per step	1	2	2	2
action cost scale	1.0	0.2	0.2	0.2
paired gain	+0.1620	+0.0326	+0.0663	+0.0717
95% CI	±0.0137	±0.0090	±0.0091	±0.0105

THE INSTABILITY WAS THE MECHANISM

`scale_rewards="group"` divides each advantage by its own group’s standard deviation. As the policy becomes confident on a task, that spread shrinks and the divisor grows, so the same reward difference pushes harder and harder. Whether that runs away depends on gradient accumulation, which is the coupling nobody warns you about.

Why one task per step matters



run 1	1 task per step
median spread	0.016
so a typical push of	64x
peak grad norm	11.25
worst dead groups	100%
eval gain	+0.1620

run 2	2 tasks per step
median spread	0.193
so a typical push of	5x
peak grad norm	0.16
worst dead groups	0%
eval gain	+0.0326

run 3	2 tasks per step
median spread	0.078
so a typical push of	13x
peak grad norm	6.77
worst dead groups	50%
eval gain	+0.0717

GRPO divides each advantage by its own group's standard deviation, at `grpo_trainer.py:2809`. With one task per optimizer step, that spread is just how much eight rollouts of the same task disagree, and it collapses as the policy makes up its mind. With two tasks per step it also contains the difference between the tasks, which never collapses. Same line of code, two entirely different training dynamics.

Figure 15 · At one task per step the group standard deviation is the within-task spread, and that collapses. At two it pools between-task variance, which never does.

The numbers are stark. Run 1 saw one task per step, so the group's standard deviation *was* the within-task spread, and its median across the run was 0.016, which means a typical step had its advantage multiplied by about 60. At its worst, every rollout in the group scored alike and the multiplier hit its ceiling. Run 2 saw two tasks per step, so the spread also contained the difference *between* the tasks, which never collapses: median 0.19, a multiplier around 5, peak grad norm 0.16, and not one dead group in the whole run.

Run 2 was stable and gained a fifth as much. The amplification was where most of the learning signal came from.

> Run 3 still falls well short of run 1, and two differences remain. One task per step instead of two, and a five times larger action cost. Run 1 changed both at once, so this narrows the cause without isolating it. The single experiment the data points at is changing only the action cost on top of run 3's config, and that is the next run.

WHAT IT COST TO TRUST THE NUMBERS

Six measurement bugs, each of which produced *plausible* results rather than errors.

bug	what it looked like	
Wrong base served	2B checkpoints scoring 0.469–0.479	vLLM accepted 2B LoRAs on a 4B bas
Dead tunnel	ckpt75 regressing to 0.4475	13% of requests got an HTML 404 rec
Two reward scales	run 2 deltas incomparable to run 1's	the stored reward is the environment's
No base arm	deltas read across sweeps	the same frozen base scored 0.465–0
Comparing across `k`	run 1 tying Sonnet 5	baselines read at pass@1 gave Sonne
Concurrent sweeps	`k = 1.8` with `PASSES=1`	two sweeps writing one directory. Arith

> Per-checkpoint gains here are about +0.008, smaller than the ±0.011 noise at 200 tasks. Single checkpoints cannot be ranked, only trends across many. Resolving a 25-step gain needs about 800 tasks per arm, not more checkpoints.

Takeaways

If you skim one section, make it this one. The sequence to follow is in the introduction. These are the things that actually cost money, and that we would not have guessed up front.

Probe your data source before you trust it. An hour spent on 40 OSV-5M sequences told us none of them were spherical, which killed a week of work we had not yet started. Read the licence on day one instead of on day thirty. Street View has the best coverage in the world and terms that rule this out entirely.

Run pass@4, and put a frozen base arm in every sweep. The same untrained model scored between 0.465 and 0.500 across our sweeps, a drift wider than most of the differences we wanted to claim. At pass@1 we briefly believed we had tied Sonnet 5. Then, once you are training, watch `frac_reward_zero_std` instead of reward. If most groups have no spread inside them, no amount of training will help, and the loss curve will not tell you. And check what your reward *cannot* distinguish before you tune anything else. Mine went flat at somewhere between 3,300 and 4,500 km depending on what an episode spent, which covered nearly a third of the untrained model's episodes, so a third of our data carried no gradient at all. Across the nine models we never touched, score also fell with turns per episode at $r = -0.75$, which was the first hint that deliberation was hurting rather than helping.

Size `max_completion_length` from a measured p95, because it allocates a logits tensor for the full budget no matter what the model emits. When the dynamics look alarming, evaluate the checkpoint before killing the run. Run 1's entropy collapse and grad-norm spike produced two recommendations to stop it, and it was the best run we did. Suppressing exactly those dynamics in run 2 cost four fifths of the gain.

Distrust a result that arrives without an error. All six of our measurement bugs returned plausible numbers, and exactly one of them ever threw an exception. The 2B adapters served on a 4B base produced a perfectly ordinary looking table of nonsense.

That last one generalises past this project. In RL the feedback loop is long and expensive, so most of the skill is in not fooling yourself between the runs.

Reproduction

Everything is published, including the raw episode records behind every number above, so nothing here is a screenshot of something you cannot check. The whole set is gathered in the GeoGuesser Env collection.

Play it. `HuggingEnvs/geoguesser-env` is the environment as a Space, playable in a browser with the controls exposed. It is the same thing the trainer and the eval talk to.

The data. `HuggingEnvs/geoguesser-tasks` holds both splits: 200 held-out eval tasks across 73 countries, and 3,452 training tasks. It is metadata only, which is what makes it small enough to version. `HuggingEnvs/geoguesser-panos` is the 22 GB of Mapillary imagery, in a Storage Bucket because it is a mutable cache instead of a source; `hf sync` pulls it locally.

The models. `geoguesser-qwen3.5-4b-grpo` is run 1, the adapter behind every number in this article. `geoguesser-qwen3.5-4b-grpo-v3` is run 3, the ablation that isolated why run 1 worked. Both are LoRA adapters on `Qwen/Qwen3.5-4B`, so they are a few megabytes rather than a few gigabytes.

The runs. `HuggingEnvs/geoguesser-trackio` has every run in one project on a shared axis, and it is embedded below in full, so you can read the curves yourself instead of taking our word for the ones we chose to draw.

The dashboard, in full

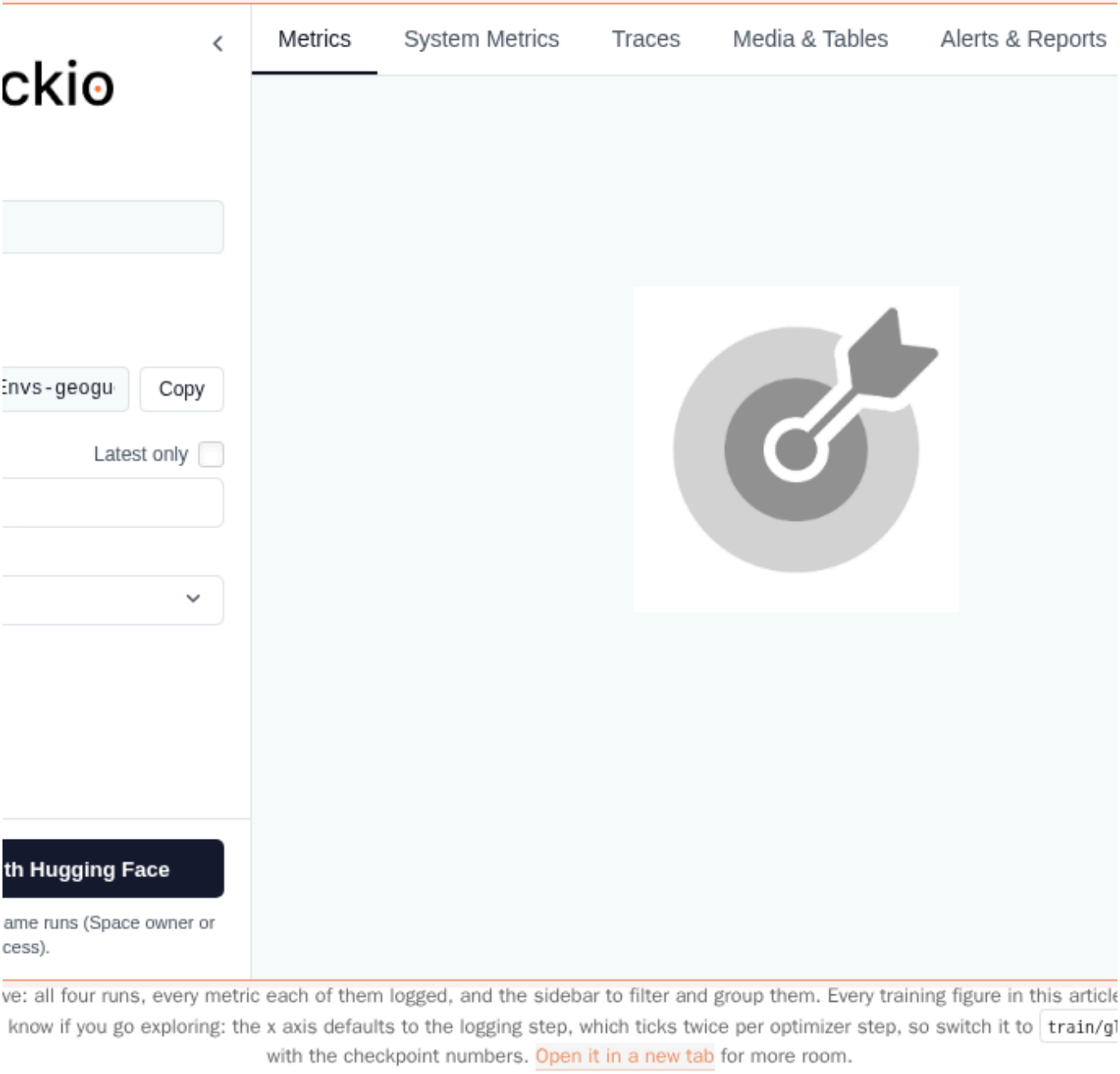


Figure 16 · All four runs and every metric they logged, live. This is the database behind every training figure in this article.

The code. `03-geoguesser` is the environment, the dataset pipeline, the eval harness and the training script, with a README per directory. `REPRODUCE.md` has the exact commands for each configuration, including the traps that cost us a run each, and every one of them works on a single GPU: `NPROC=1` is the default, and four cards were only ever about wall clock. `results/summaries/` holds the aggregated tables this article cites, and every one of them regenerates from the raw episodes with a single command, so a number that no longer matches its data is a bug in one of them.

> A reproduction that disagrees with the tables above is a bug report we want.

Citation

For attribution in academic contexts, please cite this work as

```
Adithya S Kolavi (2026). "How to turn a game into an RL environment: the technical intuition".
```

BibTeX citation

```
@misc{kolavi2026_how_to_turn_a_game_into_an_rl_environment_the_technical_intuition,
  title={How to turn a game into an RL environment: the technical intuition},
  author={Adithya S Kolavi},
  year={2026},

}
```