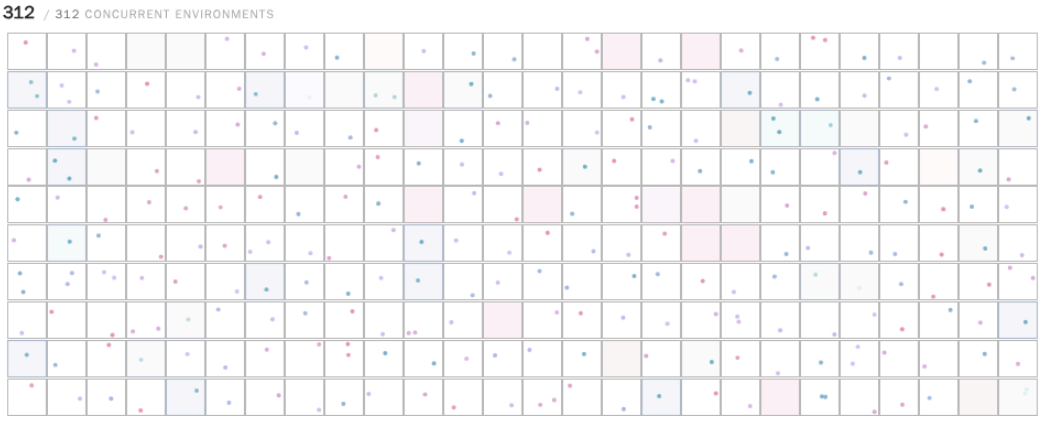
The ultimate guide to RL environments: building and scaling them in the LLM era



The anatomy of RL environment frameworks for LLM training: how they're built, how rewards are wired, and how they scale to thousands of concurrent sessions.

AUTHORS

Adithya S Kolavi, Lewis Tunstall, Leandro von Werra, Quentin Gallouédec, Amine Dirhoussi, Ben Burtenshaw, Sergio Paniego

AFFILIATION

Hugging Face

PUBLISHED

May 5, 2026

CODE

RL_Envs_101 ★

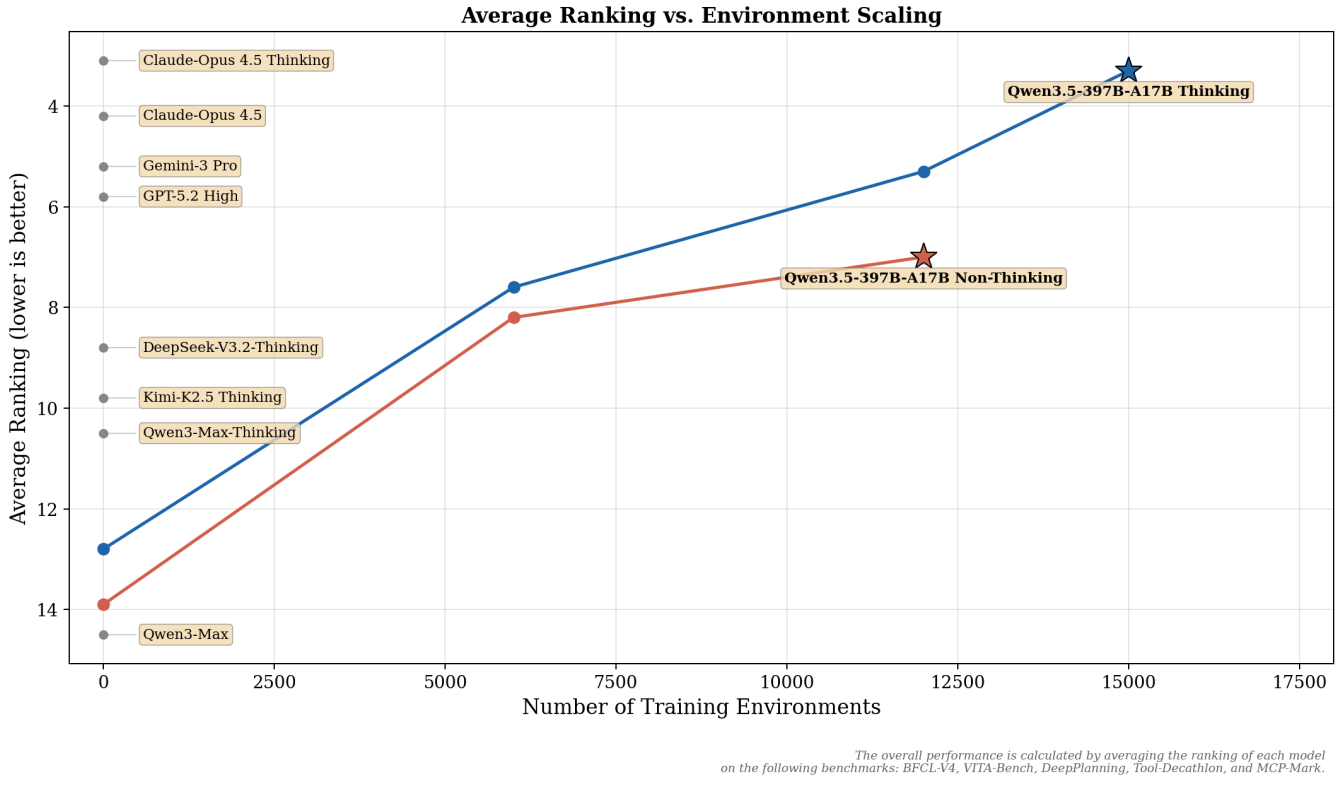
Table of Contents

Introduction

> Last updated in May 2026. This space is evolving quickly, and framework APIs, features, and ecosystem maturity may have changed since this was written.
> The research, experiments, and notes were all done by hand. Claude was used afterward to format the article, build the visualizations, and reformat the handwritten and human-reviewed content.

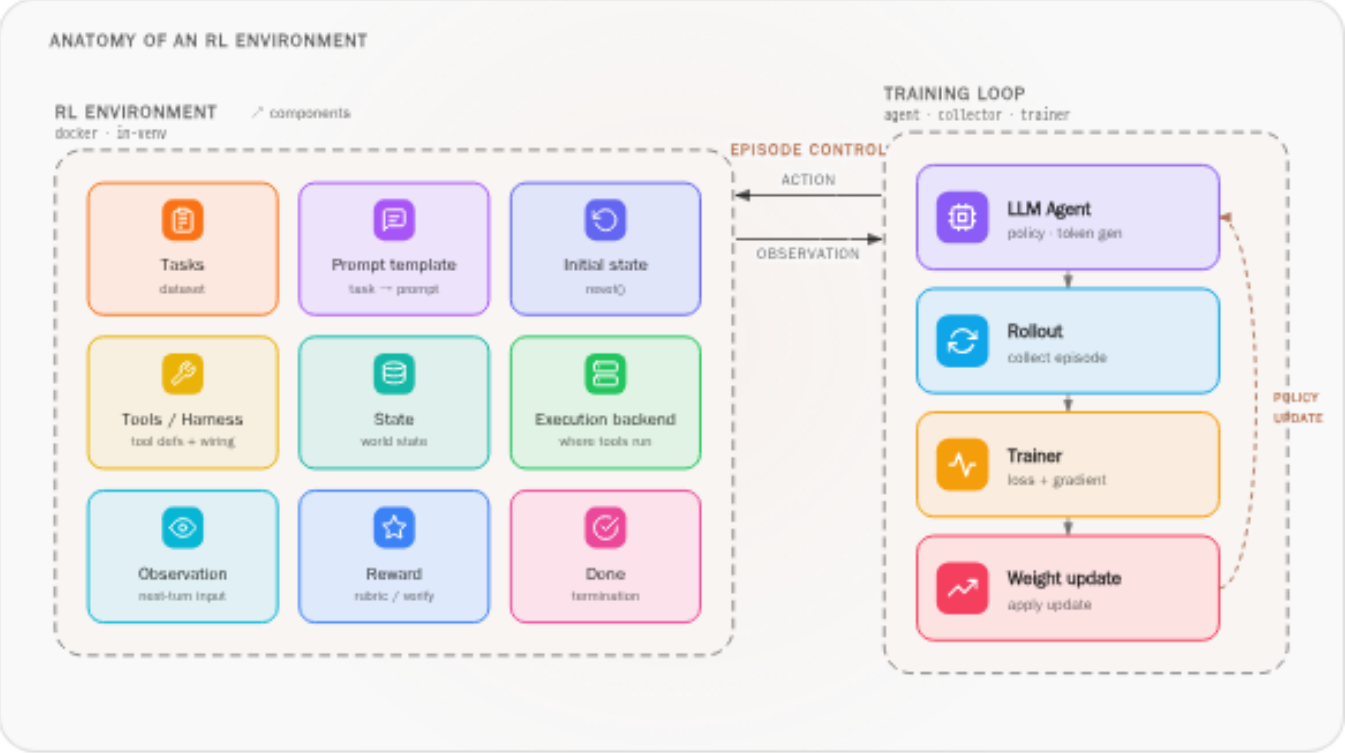
RL has become the main driver of capability gains for agentic LLMs and reasoning models, the place where supervised fine-tuning hits a ceiling and RL keeps lifting performance past it. A core piece of that progress is the RL environment, the place a model practises, gets graded, and learns from interaction over long horizons. To match capability targets, environment counts have scaled dramatically: Qwen3 trained across roughly 20 general-domain tasks, Qwen3-Coder pushed that to 20,000 parallel environments on Alibaba Cloud, MiniMax's Forge framework trains M2.5 across hundreds of thousands of real-world environments, and Qwen3.5 reports training across million-agent environments with progressively complex task distributions.

The Qwen team is explicit about why this matters. In the Qwen3.5 release notes, they attribute most of the post-training gain over Qwen3 to *"extensive scaling of virtually all RL tasks and environments we could conceive"*, deliberately raising environment difficulty and generalisability rather than optimising for narrow benchmarks.



The bottleneck is no longer “can we set up an environment”, it’s “how do we run 100,000 of them, keep them honest, and feed them into a training loop”. Frameworks are emerging to standardise that, and environment hubs are showing up alongside them where pre-built environments can be plugged into a run. The largest catalog today sits on Hugging Face Spaces, with 4k+ MCP-compatible environments shipped by the community, with PrimeIntellect’s Environments Hub and openreward.ai adding several thousand more. The anatomy of an RL environment, what it’s actually made of, has stopped being obvious and started being important.

We built the same environments across multiple RL environment frameworks. Each has a different design for what an environment should look like, what it’s composed of, and how it fits into the rest of training. We wanted to understand what components make up an RL environment in the LLM era, how they’re built, how different frameworks tackle the same problems, how rewards are wired into the loop, how easy it is to scale, and how the environment fits into the overall RL training run.



This is the anatomy we settled on. The environment side has a lot of moving parts; the agent side wraps a tight training loop around the action–observation cycle. We walk through each component in detail, then go through how environments are built, how rewards are wired, and how things scale, paired with visualisations and comparison tables so you can see exactly what each framework offers and get a feel for the RL environment landscape as a whole.

All the code (the six framework implementations, build walkthroughs, training scripts) lives in the companion repo, RL_Envs_101, so anything in here is reproducible end-to-end and easy to fork as a starting point for your own environment.

TL;DR: each framework has its own way of doing the same thing. The differences are mostly about how the environment wires into the rest of training, not what it can ultimately do. You won't miss anything fundamental by picking one over another, the same environment can be replicated across all of them. What does change is convenience, which one is most pleasant to live with day to day depends on what's already in your stack.

Why this comparison

There is no standard protocol for how LLMs interact with RL environments yet. Each framework picks its own answer for the same handful of questions, and the answers shape how you write code, how you deploy, and what you have to debug when training breaks. The four that mattered most while we were building the same env six ways:

- What is an "environment"? Some frameworks treat it as just a reward function, others include tools, state management, and the full multi-turn loop, others again bundle a whole training pipeline.
- Where does it run? Some run as HTTP servers (Docker, HF Spaces) so the env scales independently from training, others run in-process inside the training venv so there's no network hop but no isolation either.
- How much trainer comes with it? A few frameworks ship their own trainer (Prime RL, NeMo RL, SkyRL); others require adapters to plug into external training loops like TRL.
- When does the reward fire? Per-tool-call, per-step rubric, post-episode verify, or an external scoring function; each makes different assumptions about how dense the signal is and who owns the scoring code.

The rest of this article walks through these and a handful of related questions, framework by framework, with side-by-side code, benchmark numbers, and a decision tree at the end if you just want a recommendation.

Framework inventory

> *Why have environment frameworks at all?*
>
> *Mostly for standardisation. If there's an agreed protocol for how an LLM trainer talks to an environment, like MCP for tools, any training loop can plug into any environment, researchers across different domains can follow the same shape, and someone else's environments become reusable for your training run instead of one-off scripts. The frameworks below are different attempts at that standardisation.*

We surveyed the space and picked these six to build the same environment across and compare head-to-head. There are other RL-environment-adjacent projects out there that didn't fit this comparison (different abstraction layer, training-only, pure verifier libraries); they're listed below with the reasons.

Frameworks we implemented and compared

FRAMEWORKS WE IMPLEMENTED AND COMPARED — TABLE | CARDS

#	FRAMEWORK	CREATOR	TYPE	PACKAGE
1	OpenEnv	Meta PyTorch	HTTP server (MCP)	`openenv-core`
2	ORS	General Reasoning	HTTP server (REST+SSE)	`ors-sdk` (PyPI)
3	NeMo Gym	NVIDIA	HTTP server (REST)	`nemo_gym` (Git)
4	Verifiers	PrimeIntellect	In-process Python	`verifiers` (Py
5	SkyRL Gym	NovaSky-AI · Berkeley	In-process (Gym)	`skyrl-gym` (Py
6	GEM	Axon-RL	In-process (Gymnasium)	`gem-llm` (PyPI)

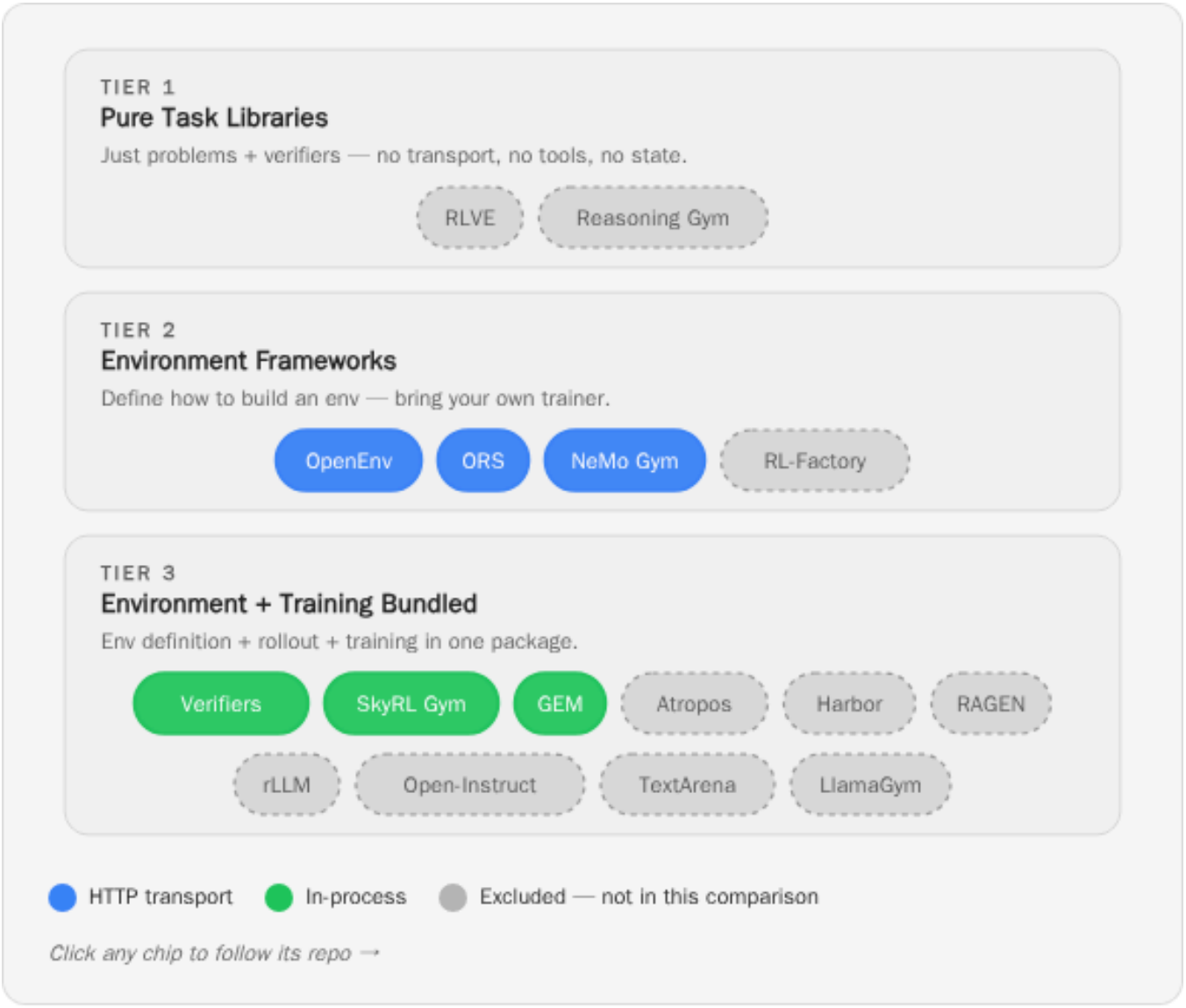
Other frameworks in the landscape

These are notable RL environment frameworks we evaluated but did not implement. They're excluded because they serve a different purpose or operate at a different level of abstraction.

Framework	Creator	
Atropos	Nous Research	Different paradigm, environments own inference and POST scored
Harbor	Laude Institute	Eval and RL rollout-generation framework, the official harness for
RLVE	Zhiyuan Zeng	Pure verifier library (445 tasks), `generate() → verify()` with
Reasoning Gym	Open Thought	Procedural task generators + verifiers, same tier as RLVE. Stateless
RAGEN	Zihan Wang	Full stack (env + StarPO + veRL), tightly coupled to its own trainin
rLLM	Agentica	Decorator pattern, wraps existing agent code, intercepts LLM call
RL-Factory	Simple-Efficient	MCP config-based, any MCP server becomes an environment. Int
Open-Instruct	Allen AI	Full training framework with env hooks, environments are reward
TextArena	Leon Guertler	Game-specific multi-agent environments, narrow domain, not a ge
LlamaGym	KhoomeiK	Gymnasium wrapper for LLMs, early prototype, not actively maint

How these relate

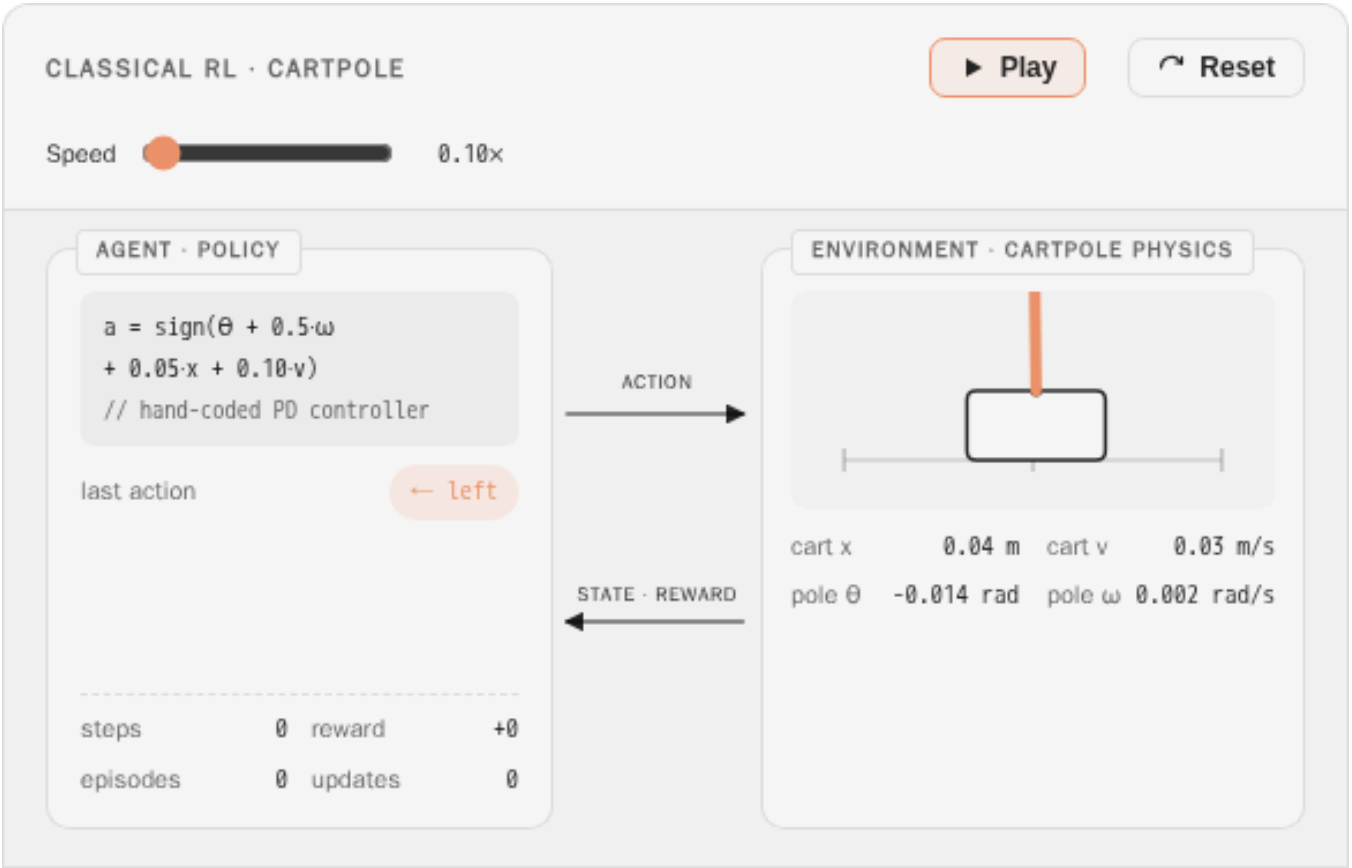
The 16 frameworks we surveyed split cleanly into three tiers.



We focused on Tier 2 + Tier 3 frameworks that support multi-turn tool-calling environments.

What is an RL environment in the LLM age?

In classical RL (Atari, robotics), an environment is small and self-contained. The textbook example is CartPole: a cart on a track with a pole balanced on top. The agent observes a 4-number state (cart position and velocity, pole angle and angular velocity), picks one of two actions (push left or right), and the environment ticks one step of physics forward. The reward is +1 every step the pole stays upright. The whole loop is a tight feedback cycle, and "the environment" is the physics simulator plus the reward rule.



One full RL loop: the agent reads the state, the policy picks a discrete action (push left or right), the environment ticks one physics step and returns the next state plus a reward of +1 for every step the pole stays upright. When an episode ends the policy would be updated from the collected return — shown here as the agent box flashing. Physics and bounds match the OpenAI Gymnasium CartPole-v1 spec.

In the LLM era the picture is more complex, and there's no single canonical implementation. To set the analogy, here's one common shape: the "agent" is a language model, the "environment" is a sandbox that runs shell commands or executes code, and the "action space" is whatever set of tools the framework exposes. Each rollout is a multi-turn conversation where the model writes, runs a tool, reads the output, decides what to try next, and eventually submits an answer. The environment scores the completed rollout and returns a reward, and a training step pulls in a group of these rollouts per prompt to learn from. Many other shapes exist (single-turn reasoning, agentic web tasks, code-repo agents, multi-agent setups), but they're all variations on the same skeleton. What changes is the tools, the observations, and the reward rule.

LLM RL · MULTI-ROLLOUT

Pause Reset

Speed 1.00×

AGENT · LLM

MODEL

```
samples N rollouts per task
// trainer updates from
rewards
```

LAST TOOL CALL

–

STATS

rollouts shown 0/2 tool calls

group avg – group size

TOOL CALL ↓ ↑ OBS · REWARD

ENVIRONMENT · SANDBOXED SHELL

TASK Find the `.py` file with the most lines in the current directory.

ROLLOUT 1

ROLLOUT 2

+ N more rollouts sampled in parallel

Each rollout is a multi-turn trace: the model writes a thought, calls a tool, reads the output, decides what to do next, and submits an answer. The environment scores the completed trace and returns a reward, and the training loop collects a group of these traces per prompt to learn from. This is one shape of environment, multi-turn tool use; LLM-based RL environments vary widely (single-turn reasoning, agentic web tasks, code-repo agents, multi-agent setups), and the same traces can feed very different training methods. GRPO is an example of online RL, where the policy updates from the group right away, but the same traces can equally

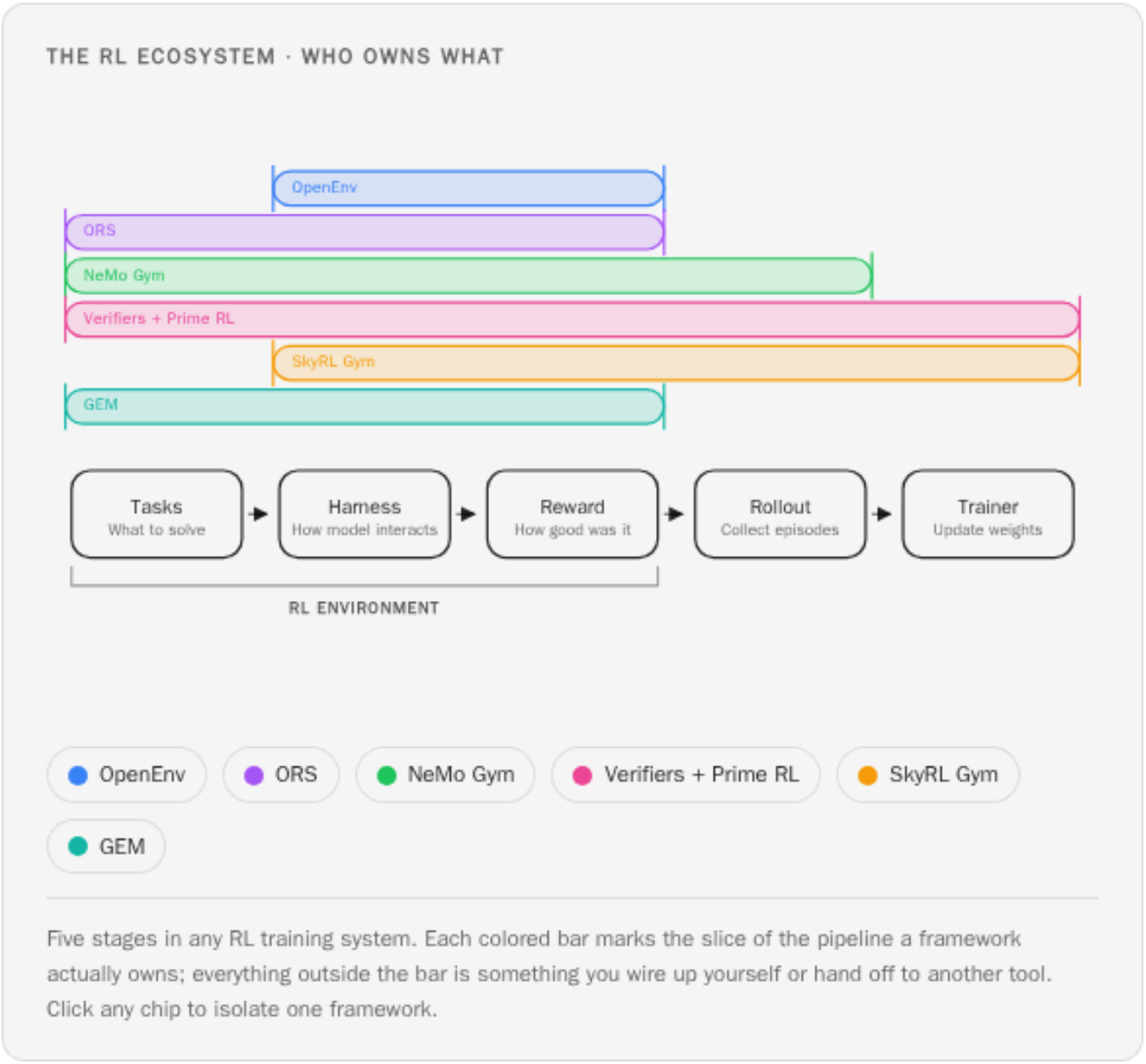
example of online RL, where the policy updates from the group right away, but the same traces can equally be stored and used later for distillation, offline RL, or imitation learning.

That whole loop, from picking a task through to updating the policy, is what an RL training system has to handle end to end. The catch is that no two frameworks split that work the same way. Some hand you only a thin protocol over the environment; others bundle a full trainer on top. Before we taxonomize the individual pieces, it's worth pulling back to a bird's-eye view and seeing who covers what.

How an RL training system fits together

Every RL training system for LLMs sits on the same five-stage spine: there are tasks to solve, a harness that lets the model interact, a reward signal that scores the behaviour, a rollout collector that gathers full episodes, and a trainer that turns those episodes into a policy update. What changes from one framework to the next is which slice of that spine they ship to you in the box.

The diagram below maps each framework's bracket onto those five stages.



Take OpenEnv, for example: it gives you the harness (the tools the model can call) and the reward, but you have to write the trainer yourself. Verifiers, on its own, is really just a way to describe the task, the tools, the harness, and the reward; the actual training is handled by Prime RL, which Verifiers pairs with. That's why a flat "X vs Y" comparison is hard. Each framework was built to slot into a different ecosystem, and they don't all live at the same level of the stack. What we can do, and what the rest of this article does, is define what an RL environment actually is, and then compare every environment framework across those same dimensions so you can see exactly what each one offers. For a complementary cut at the same question from the angle of async vs sync training architectures, see Hugging Face's async RL training landscape.

That bird's-eye view tells us where each framework sits in the wider stack. The next thing to pin down is what's actually inside the slice marked "environment".

What makes an RL environment for LLMs (the components)

So far we've looked at the loop in both its classical and LLM form, and at how the frameworks split that loop across the same five stages. Now we open the box itself. After spending enough time inside all six frameworks, the same set of pieces kept showing up: the parts you actually have to think about when you build or pick an environment for an LLM agent. They're listed below, each with the question it answers and how it usually shows up in code.

Grid | **Table**

COMPONENT & WHAT IT ANSWERS	EXAMPLE
Tasks / Dataset *What problems should the model solve?*	Coding tasks, math problems, game instances
Initial State Management *How is per-episode state set up at the start of a rollout?*	`reset()` populates state, `setup()` lifecycle, `seed_session` endpoint, `setup_state` hook
Prompt Template *How is the task presented to the model?*	System prompt + user instruction + tool descriptions
Tool Definition *What can the model do in the world?*	`execute_code()`, `search()`, `submit_answer()`
Observation Format *What does the model see back after an action?*	Raw text, structured `ToolOutput` with blocks, Gymnasium 5-tuple
Execution Backend *Where do actions actually run?*	Cloud sandbox, Docker container, Python `eval()`, game simulator, or none (pure Python)
State Management *How is state tracked across turns?*	Session cookies, sandbox persistence, conversation history
Reward / Rubric	String matching, LLM-as-judge, unit tests, `verify()`

	How do we score the model's behavior?	endpoint
Done / Termination	*How does the episode end?*	`finished` per tool call, `terminated` / `truncated`, max turns, `verify()` decides
Episode Control	*Who drives the multi-turn loop and decides when to stop?*	Trainer (TRL), environment (Atropos), or framework (Verifiers)
Transport / Protocol	*How does the model talk to the environment?*	HTTP REST, WebSockets, JSON-RPC (MCP), SSE, in-process Python calls

These are the dimensions we'll use to compare frameworks across the rest of the article. The next section maps each framework against this list to show what they ship for you and what they leave for you to build.

What each framework ships (raw, out of the box)

This is the key table. Each framework provides a different subset of components. Three states show up in the cells: ✅ full means the framework ships a first-class API for that component; ⚙️ partial means it works but you lean on a convention or trainer hook to get there; 🔧 BYO means it's left to you to bring.

What this means in practice

Most components bundled. Verifiers bundles dataset + tools + rubric + rollout harness + training. You define your problem and it handles everything, with the trade-off that you work within the Verifiers API surface.

Fewest components bundled. OpenEnv provides a protocol (MCP), session management, and a built-in Rubric system for rewards. You bring your own tasks and execution backend, more setup, more control over each piece.

In between. ORS and NeMo Gym provide a deployment protocol + reward mechanism, but you bring your own execution backend and tasks. GEM provides built-in environments + Gymnasium API but you bring the trainer.

The catch with reading any one framework's docs is that everyone uses different vocabulary for the same idea. The reward function is a `Rubric` in OpenEnv and Verifiers, a `verify()` endpoint in NeMo Gym, the `reward` field on a `ToolOutput` in ORS, and just whatever you return from `step()` in SkyRL Gym and GEM. The end-of-episode flag is `done` in some, `terminated` vs `truncated` in others, and `finished` on every tool call in ORS. A single training prompt is a "task" in one place, a "split row" in another, a "JSONL line" in a third, and a `Dataset` row in a fourth. Same concept, six names. Before we go deeper, here's a translation table.

How each framework names the same things

The same idea has a different name in every framework. Reading two of these docs back-to-back can feel like reading two different APIs for the same concept, because that's basically what it is. The table below pins each concept (the rows we just walked through) to whatever each framework calls it, so when you jump between docs you can carry your mental model with you.

Cards | Table

CONCEPT	OPENENV	ORS	NEMO GYM
Task collection	—	`Split` · `list_tasks()`	JSONL dataset
Initial state setup	`reset()`	`setup()`	`seed_session`
Single task	—	`task_spec` (JSON)	JSONL line
Tool definition	`@mcp.tool`	`@tool` → `ToolOutput`	`app.post("/name")`
Observation back	`str` return	`TextBlock` in `ToolOutput`	`Response(output=str)`
Reward signal	`Rubric(action, obs)`	`ToolOutput.reward`	`verify()` → reward
Episode state	MCP session	HTTP session (`X-Session-ID`)	Cookie session
Done signal	`Observation.done`	`ToolOutput.finished`	—
Task prompt	—	`get_prompt()`	In JSONL `input`

A few things stand out once it's all in one view: every framework has *something* under "Initial state setup" and "Tool definition" because every multi-turn env needs both, but "Task collection" is only filled in for frameworks that bundle their own task source. "Episode state" is where the HTTP frameworks all use sessions and the in-process ones lean on plain Python. With the components named and the vocabulary mapped, the next chapter takes each of these dimensions in turn and compares the frameworks in detail.

Dimensions of comparison

So far we've listed the components and seen which framework covers which. Now we go through them one at a time. Each section picks one component, asks the practical question it raises, and lines up all six frameworks side by side so you can see how each one handles it.

Dimension 1: Building an RL environment

What code do I actually write?

The first thing any environment author runs into is the API surface. What do I subclass, how do I declare a tool, how much boilerplate sits around the actual logic? The same toy environment is written six different ways below; flip between tabs to see how each framework wants you to write it. The runnable end-to-end versions of every snippet (build scripts, server scaffolding, training entrypoints) live in the companion repo, RL_Envs_101, so you can clone it and have all six up locally.

OpenEnv · ORS · NeMo Gym · Verifiers · SkyRL Gym · GEM

SAME ENV, SIX APIS

OpenEnv · MCP tools registered inside an MCPEnvironment — PYTHON — COPY

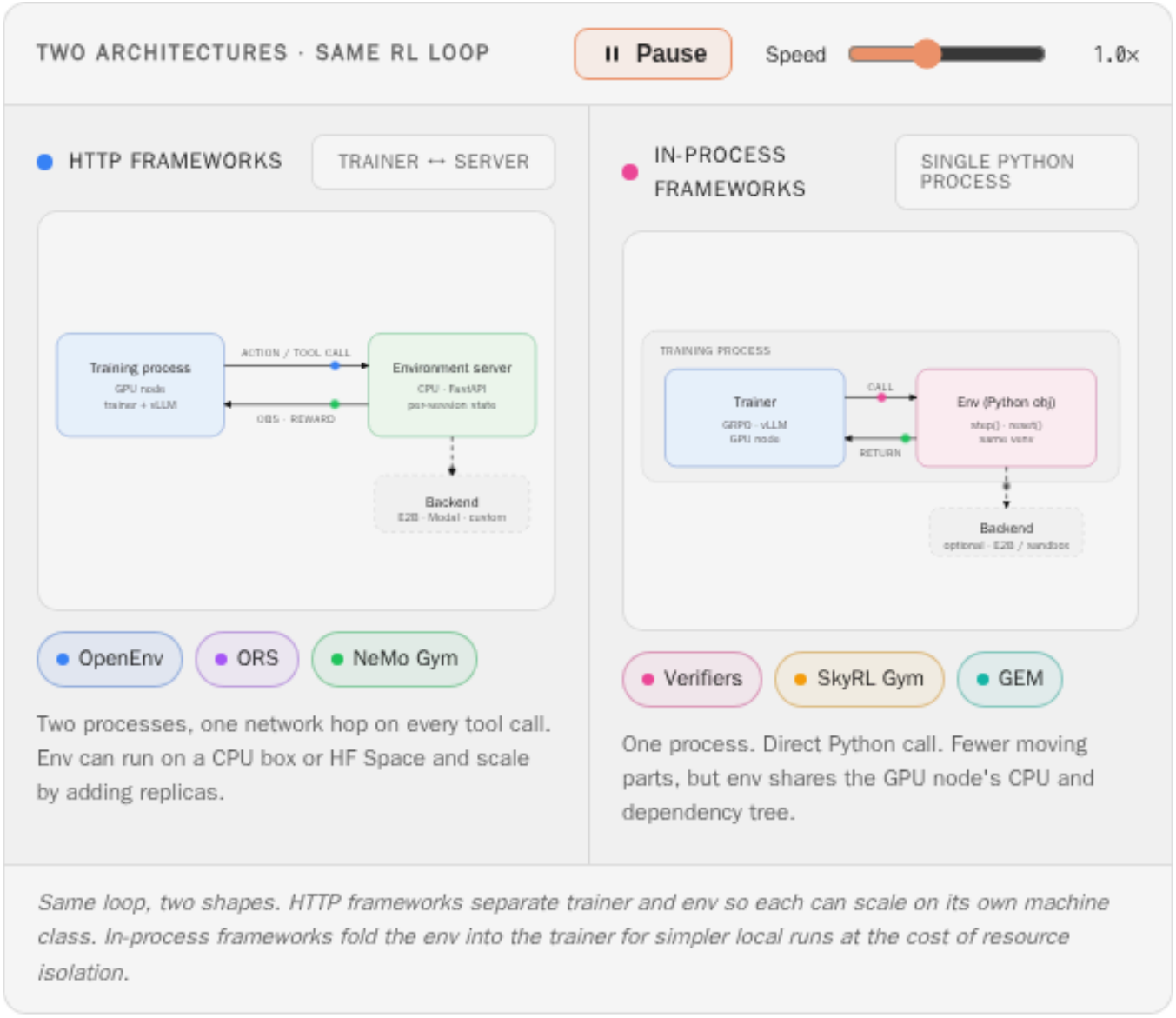
```python
class MyEnv(MCPEnvironment):
    def __init__(self):
        mcp = FastMCP("my_env")

        @mcp.tool
        def do_action(input: str) -> str:
            """Execute an action. Returns observation."""
            return execute(input)

        super().__init__(mcp)

    def reset(self, **kwargs) -> Observation:
        self._state = fresh_state()
        return Observation(done=False, reward=None, metadata={"status": "ready"})
```

	OpenEnv	ORS	NeMo G
Base class	`MCPEnvironment`	`ors.Environment`	`SimpleResourc`
Tool syntax	`@mcp.tool`	`@tool` + Pydantic input	`app.post("/na`
Return type	`str`	`ToolOutput(blocks, reward, finished)`	`Response(outp`
Entry point	`create_app(Env)`	`Server([Env]).run()`	`Env.run_webs`

Dimension 2: Communication & deployment

Where does it run and how do they talk?

The most fundamental architectural split: does your environment run as a separate HTTP server or inside the training process? Everything else, the protocol, the deployment target, what you have to install on the trainer side, falls out of this single choice.



The two patterns differ on three things that show up in practice:

- Where it runs. An HTTP framework lives on its own machine, often a cheap CPU box or a Hugging Face Space. The in-process kind shares the training GPU node.
- What you install on the trainer side. HTTP only needs an SDK or `requests`. In-process pulls the full framework package into the training venv.
- How it scales. HTTP scales by adding server replicas behind a load balancer. In-process scales by adding more identical training workers, each with its own copy of the environment.

The table below lines all of that up at a glance.

COMMUNICATION & DEPLOYMENT · 6 FRAMEWORKS — ✓ ships — — no — N/A not applicable

CAPABILITY	OPENENV	ORS	NEMO GYM
SHAPE			
Type	HTTP server	HTTP server	HTTP server
Protocol	JSON-RPC 2.0 (MCP) · WebSocket	REST + SSE	REST + cookies
DEPLOYMENT			
Docker	✓ Yes	✓ Yes	✓ Yes
HF Spaces	OpenEnv ↗	ORS ↗	NeMo ↗
OpenReward	—	✓ Yes	—
PrimeIntellect Hub	—	—	—
Scales independently	✓ Yes	✓ Yes	✓ Yes
TRAINER-SIDE			
Deps on trainer	`openenv-core`	`requests` only	`requests` only
Auth	None	`X-API-Key`	Cookies

Note: Verifiers integrates with PrimeIntellect's training stack (Prime RL) and their managed environment hub, which hosts community-contributed environments for distributed RL training.

One thing worth flagging: in practice, most environments don't run the heavy work themselves, they delegate it to a sandbox provider like E2B, Modal, or a custom container backend. That changes the scaling story for both shapes. An HTTP server becomes mostly a thin router holding sessions and forwarding tool calls into the sandbox, so adding replicas is cheap. The in-process flavor ends up doing roughly the same thing, the env class is a thin client over the sandbox, so the per-worker footprint is small even when you spin up hundreds of workers. The deployment shape still matters (HTTP isolates dependencies, in-process shares them), but the heavy compute lives behind whichever sandbox you picked.

Dimension 3: Tool & action model

How does the model interact with the environment?

All six frameworks ultimately expose the same thing to the model, a list of callable tools with names, descriptions, and typed parameters. What differs is where the tool definition lives and how the trainer discovers it. HTTP frameworks define tools on the server and ship a discovery endpoint the client hits at runtime. In-process frameworks define tools as Python functions and register them when the env constructs. After discovery, both shapes look identical: send a tool call, get back an observation.

TOOL DISCOVERY & INVOCATION · 6 FRAMEWORKS

Pause

Speed 1.0×

OpenEnv

HTTP

DEF	`@mcp.tool` on FastMCP
DISCOVER	`tools/list`
SCHEMA	MCP tool spec (JSON Schema)

ORS

HTTP

DEF	`@tool` + Pydantic
DISCOVER	`GET /tools`
SCHEMA	ORS ToolSpec

NeMo Gym

HTTP

DEF	`app.post()` endpoints
DISCOVER	`GET /openapi.json`
SCHEMA	OpenAPI JSON Schema

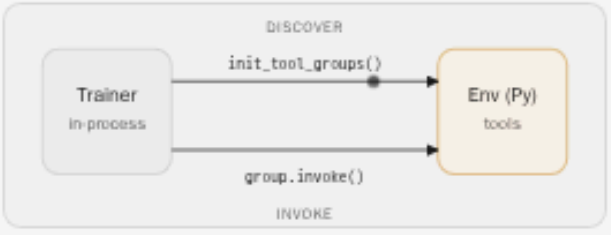
Verifiers

IN-PROCESS

DEF	Plain Python fns
DISCOVER	`tools=[fn]`
SCHEMA	Type hints (auto)

SkyRL Gym

IN-PROCESS



DEF	`ToolGroup` subclass
DISCOVER	`init_tool_groups()`
SCHEMA	Type hints (auto)

GEM

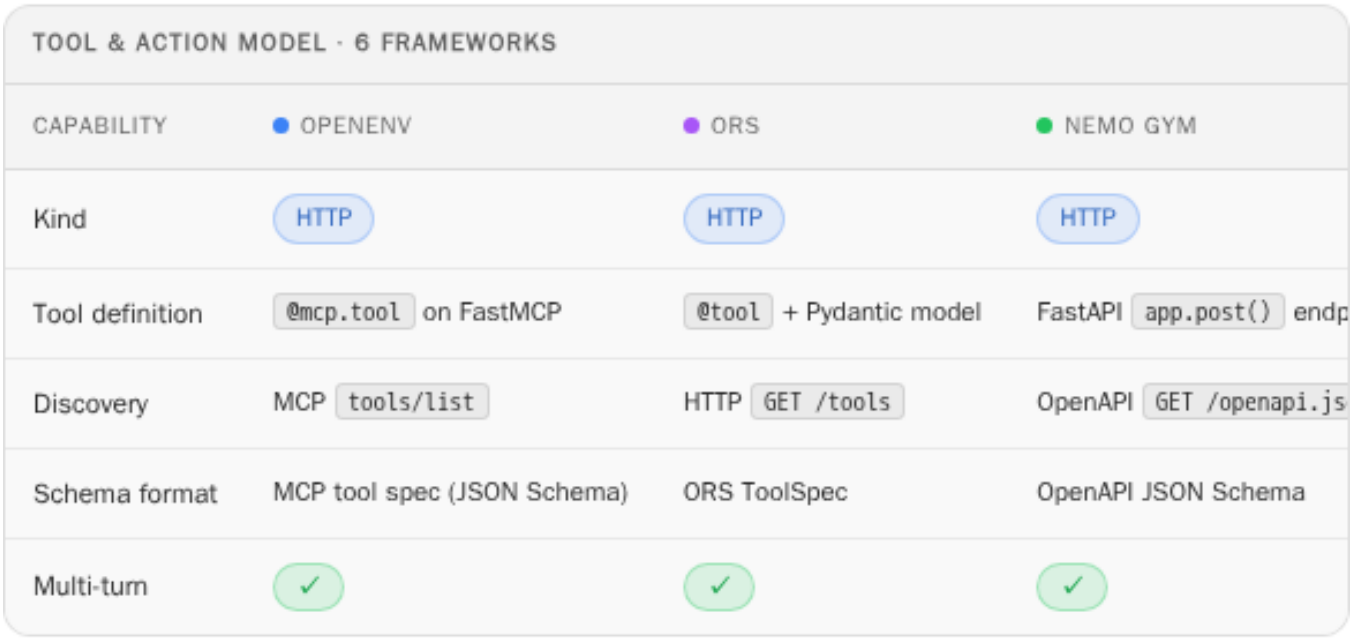
IN-PROCESS

DEF	`ToolEnvWrapper`
DISCOVER	`wrapper compose`
SCHEMA	Wrapper-defined

Same loop, two discovery shapes. HTTP frameworks fetch the tool list at runtime over the wire (dashed line). In-process frameworks register tools when the env constructs (solid Python call). After that, both look the same to the model: name + typed args in, observation out.

A few practical things fall out of these differences:

- Schema portability. HTTP frameworks ship JSON Schema (MCP, ORS ToolSpec, OpenAPI), which any OpenAI-compatible model knows how to call, so the same tool spec hops between LLMs cleanly. In-process frameworks auto-derive schemas from Python type hints, faster to write but harder to lift into another framework or a model that expects a specific schema dialect.
- Authoring cost per tool. A `@tool` decorator on a Python function is one line. A FastAPI endpoint plus its Pydantic body model is closer to a small file. Multiply by your tool count when you're picking.
- Multi-turn looping is universal across all six. Once tools are wired up, the LLM-side experience is identical: name + typed args in, observation out. The differences live upstream, in how you got that wiring there.



Dimension 4: Reward architecture

Who decides how good the model's behavior was, and when?

This is where frameworks differ most philosophically. Four distinct patterns exist. The animation below stacks them on the same trajectory so the timing differences read at a glance.

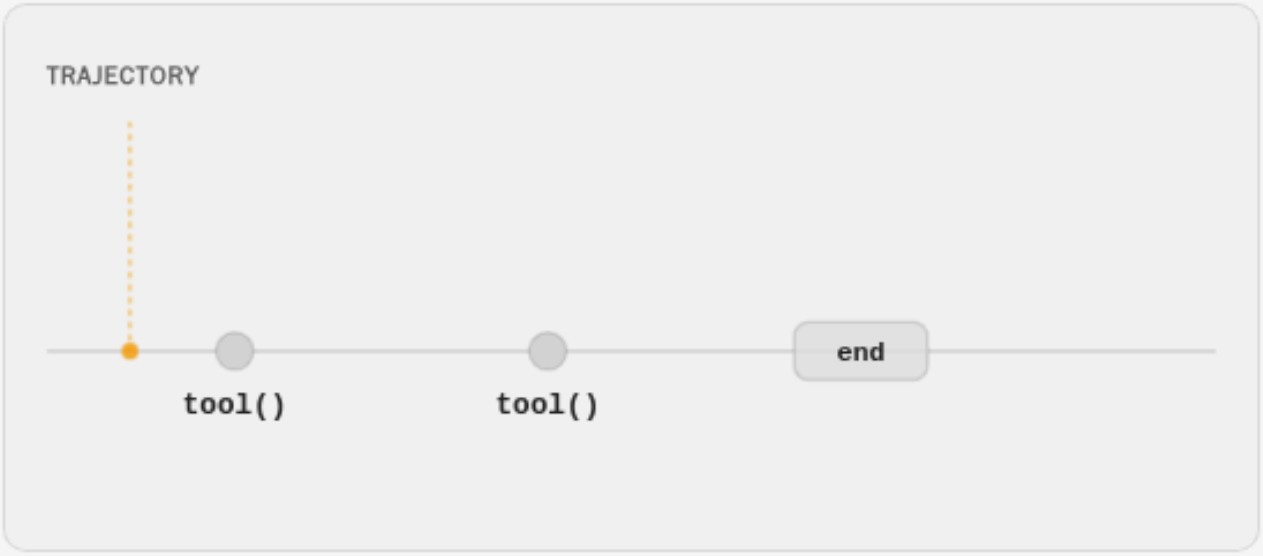
WHEN DOES THE REWARD FIRE?

Pause

Speed 1.0×

External reward

AFTER ROLLOUT

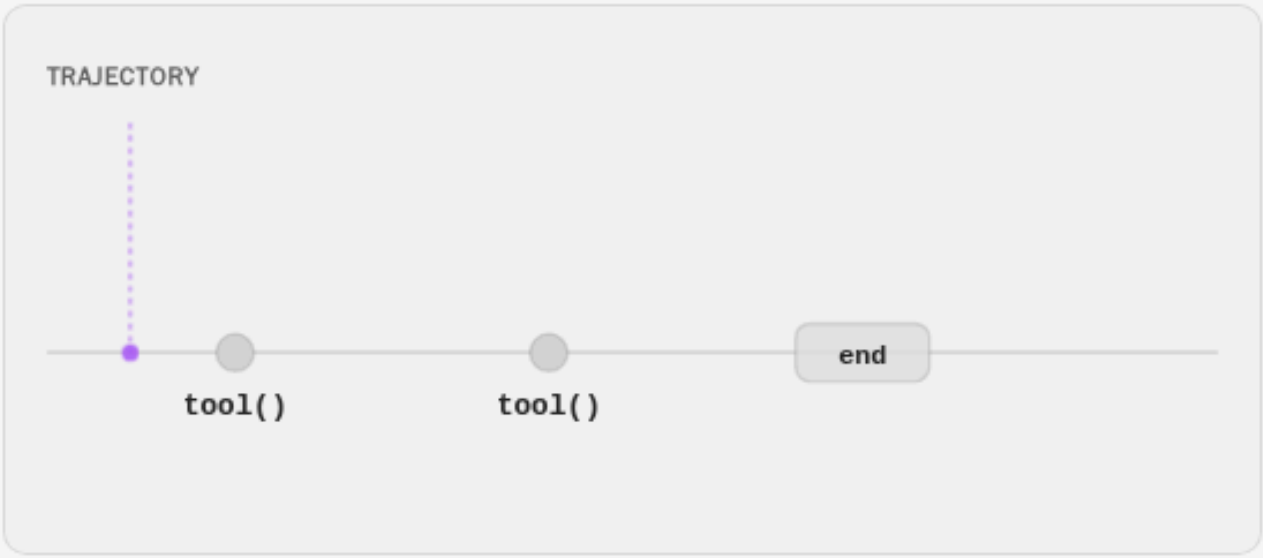


SkyRL Gym · GEM

Trainer wires up its own reward function. The env returns text from `step()`; you score the trajectory afterwards.

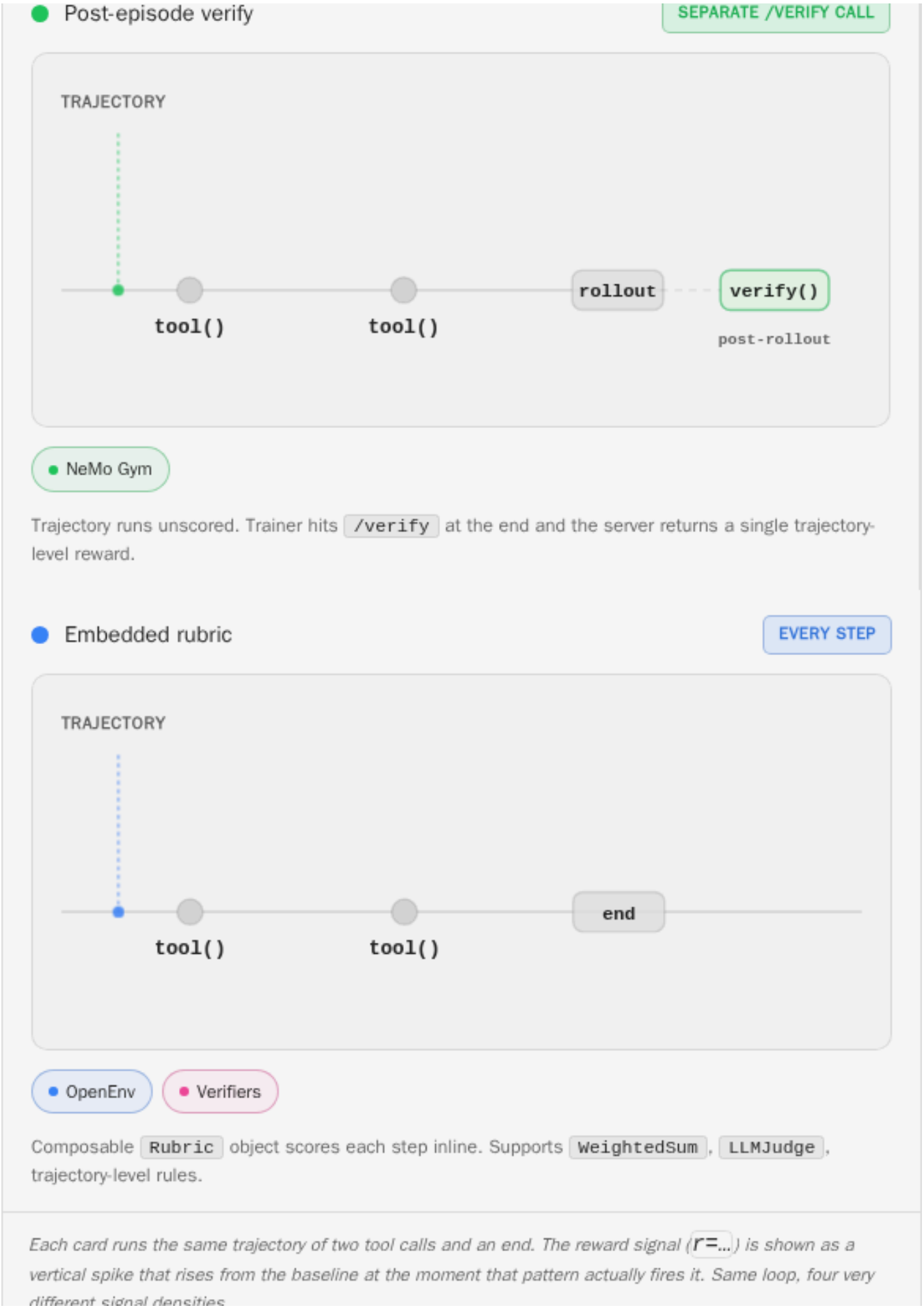
Server-embedded

EVERY TOOL CALL



ORS

Server attaches a reward to every `ToolOutput`. Trainer reads `env.reward` as it goes, no separate scoring step.



Pattern 1: External reward. The training script decides. The environment just returns tool outputs (text). A separate reward function compares the result to expected output. The

environment returns a reward you wrote, but provides no scoring system or rubric primitives, you write the reward function yourself.

> *Used by: SkyRL Gym, GEM*

Pattern 2: Server-embedded reward. Every tool response includes a reward. The server evaluates as it goes. The trainer just reads `env.reward`.

> *Used by: ORS*

Pattern 3: Post-episode verification. A separate `/verify` endpoint is called after the episode. The server does a holistic evaluation of the full trajectory.

> *Used by: NeMo Gym*

Pattern 4: Environment-embedded rubric. The environment has a composable `Rubric` object (modeled after `nn.Module`) that computes rewards from actions and observations. The rubric is called automatically during `step()` and sets `observation.reward`. Supports composition (`WeightedSum`, `Sequential`, `Gate`), LLM-as-judge (`LLMJudge`), and trajectory-level scoring.

> *Used by: OpenEnv, Verifiers*

The table below pins each framework to its pattern, with column tinting so frameworks that share a pattern (OpenEnv + Verifiers; SkyRL + GEM) cluster visually without re-ordering the columns.

REWARD ARCHITECTURE · 6 FRAMEWORKS — Embedded rubric | Server | Verify | External

CAPABILITY	OPENENV	ORS
Pattern	Embedded rubric	Server-embedded
Who computes	Env (via `Rubric`)	Env server
When	Per step (`_apply_rubric`)	Every tool call
Reward in tool response?	✓ `Observation.reward`	✓ `ToolOutput.rewar`
LLM-as-judge	✓ `LLMJudge` rubric	In server `@tool`
Composability	High · `WeightedSum`, `Sequential`, `Gate`	Medium · server decides

A NOTE ON WHAT'S ACTUALLY INSIDE A REWARD FUNCTION

The four patterns above are about *when* the reward is computed and *who owns the code*. What goes *inside* the reward function is a separate question, and three flavors keep showing up across these frameworks: procedural / verifiable rewards, LLM-as-judge rewards, and dense (vs sparse) rewards.

Procedural (verifiable) rewards are simple deterministic checks: did the model's final answer match the expected output, did the unit test pass, did the JSON parse, did the math evaluate to the right number. This is the "reinforcement learning with verifiable rewards" (RLVR) pattern that DeepSeek-R1 and others scaled up: a model only earns reward when its output passes a programmatic check, typically a 0/1 binary signal. Scalable, hard to game, and well-suited to math, code, and logic tasks. The catch: it only works where you have a clear ground-truth oracle.

LLM-as-judge rewards (sometimes called RLAIF) cover the cases procedural rewards can't: creative writing, open-ended reasoning, summarization quality, agent coherence, anything where "correct" is subjective. A separate language model reads the trajectory and scores it, often guided by a rubric so the judgment is structured (e.g. "rate factual accuracy 1–5, rate clarity 1–5, weight them"). This is what `LLMJudge` and `JudgeRubric` are doing in the `Rubric` systems above. The catch is reward hacking: policies trained against an LLM judge can learn to produce outputs the judge over-rewards but a stricter gold-standard judge would penalize. Recent work pushes back on this with rubric-based decomposition and "thinking judges" (J1-style) that reason before scoring.

Dense vs sparse is a separate axis cutting across both of the above. A *sparse* reward fires only at the end of a trajectory: pass/fail, final score. A *dense* reward fires per step, per token, or per intermediate sub-goal, giving the trainer a much richer credit-assignment signal. The four patterns map to this fairly directly: external and post-episode-verify are sparse by default, server-embedded and embedded-rubric can be either depending on how you write them. Embedded `Rubric` is the most natural place to make rewards dense, you compose multiple per-step scoring functions (`WeightedSum`, `Sequential`) and the rubric ships a non-trivial reward at every action. Dense rewards train faster and more stably; sparse rewards are simpler to define and harder to game.

In practice you almost always end up with some mix: a procedural component for the parts you can verify (the unit test passed, the JSON is valid, the answer matches), plus an LLM-judge component for the parts you can't (was the explanation good, did the agent stay on task), composed with weights into a single scalar. The frameworks above just differ on where you write that composition and when it fires.

Dimension 5: Episode control

Who drives the multi-turn loop, and what tells the episode to stop?

A multi-turn rollout is just a tight cycle: model generates an action, tool gets called, observation goes back to the model, repeat. Two things have to be decided up front: who runs that cycle (the trainer pulling tokens out of the model and pushing them into the environment, or the environment owning generation itself), and how the cycle knows it's done (a flag on the response, a dedicated terminator tool call, a max-turn cap, or a verifier deciding after the fact). Each framework picks its own answers, and those choices shape how you batch rollouts, how parallelism works, and what kind of training data falls out the other end.

> *Note: In all the frameworks we tested, the trainer drives the multi-turn loop, it generates, parses tool calls, executes them via the adapter, and re-generates. The environment is passive (it responds to tool calls but doesn't initiate anything). This means the trainer controls episode length, tool-call parsing, and retry logic. Some frameworks we didn't test (Atropos, Verifiers native mode) flip this, in Atropos, the environment owns its own vLLM instance and generates rollouts autonomously; in Verifiers' `env.evaluate()` mode, the framework manages the loop internally. This distinction matters because it changes how you think about batching, parallelism, and resource allocation.*

The other half of episode control is termination, what signal tells the loop to stop. Each framework picks its own answer: a `done` field on the observation, a `finished` flag on every tool call, a separate `/verify` step after the rollout, a Pythonic `@vf.stop` decorator, a single `done` bool, or `terminated` vs `truncated` from a Gym-style 5-tuple. The table lines them all up.

TERMINATION & LOOP DRIVER · 6 FRAMEWORKS

FRAMEWORK	TERMINATION SIGNAL	GRANULARITY
OpenEnv	`Observation.done` · `final_answer` tool	per step + trainer max-len
ORS	`ToolOutput.finished`	per call
NeMo Gym	`verify()` after rollout	post-episode
Verifiers	`@vf.stop` · `is_completed()`	per step
SkyRL Gym	`step().done`	per step
GEM	`terminated` · `truncated`	terminated / truncated

What this looks like in practice depends on what's exposing the tools. Take a coding-agent environment with a Claude-Code-style tool set: file ops (`read`, `write`, `edit`, `multi_edit`, `ls`, `glob`, `grep`), shell (`bash(command, timeout=30)`), planning (`todo_write(todos)`), and a terminator `submit_solution()`. Two examples show how the trainer-driven vs environment-driven split plays out in that setup:

> *Example 1 — Tool-based control (trainer-driven). The environment just exposes the tools and the loop ends when the model calls `submit_solution()`, a `final_answer`-style terminator. Every framework in the comparison supports this out of the box and it ports almost identically across all six: trainer drives generation, environment validates each tool call, episode closes when the terminator fires (or `max_completion_length` cuts it off). For most agent-tool environments, this is the path of least resistance.*

> *Example 2 — Harness-controlled rollout (env-driven). Sometimes you don't want the trainer to drive at all. You want a real coding harness, OpenCode, openclaw, Aider, claude code style, running inside a sandbox with the model talking to that harness directly. The harness owns the conversation: managing the file tree, compressing context, retrying on tool failures, knowing when the task is done. The environment just hosts the harness and hands the trajectory back when it finishes. Atropos and `vf.evaluate()` lean this way. It's powerful for benchmark-grade fidelity, but the catch is collecting training-ready data, the harness has to emit the right per-turn record (action, observation, reward, logprobs) for whatever loss your trainer uses, and that's harness-specific work each time.*

The practical read: tool-based control is a portable convention across all six frameworks, which is where most real environments live. Harness-controlled rollouts buy you benchmark-grade fidelity but turn data collection into its own problem.

Dimension 6: Tasks & datasets

Where do the prompts come from, and what comes with them?

Every rollout starts with a task. The model takes that task as input, acts on the environment across the span of the episode, and the environment scores the result against whatever the task said success looked like. The task is what tells the model *what to do this episode*, the prompt, the input data it operates on, and (for scoring) the expected answer or test that decides whether it succeeded. This is the most varied dimension after reward, the six frameworks land on six different answers for where that task comes from. Some bundle a dataset (Verifiers ships HF `Dataset` integration, GEM has a registry of 24+ built-in environments). Some put the task store on the server (ORS exposes `list_tasks(split)`). Some preprocess JSONL through a CLI (NeMo Gym's `ng_prepare_data`). And two leave it to you (OpenEnv, SkyRL Gym). The cards below trace each path from source to environment.

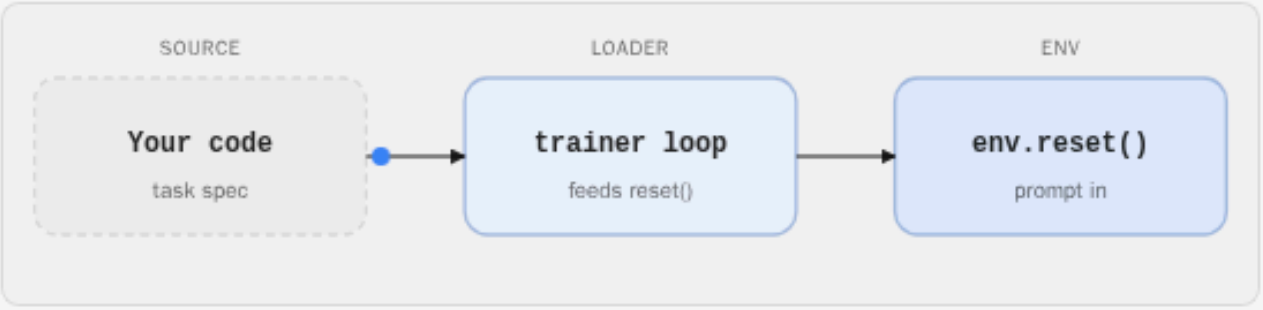
WHERE DO PROMPTS COME FROM? · 6 FRAMEWORKS

Pause

Speed 1.0×

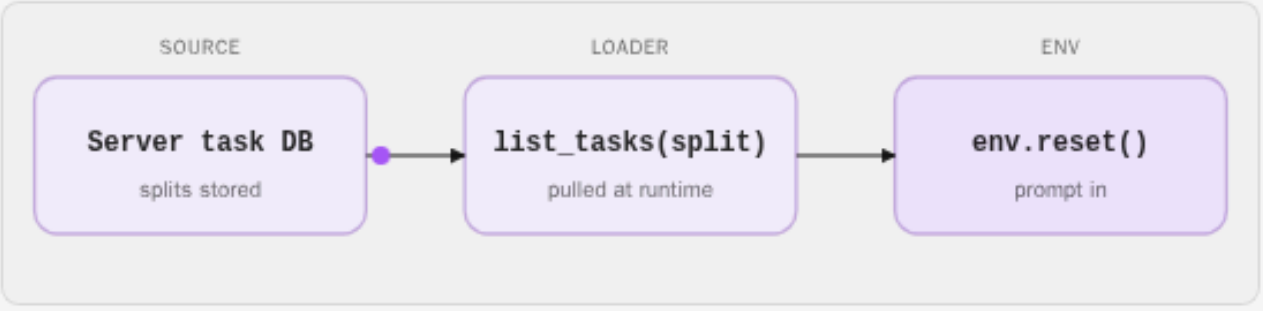
OpenEnv BYO

PROMPT BYO
SPLITS —



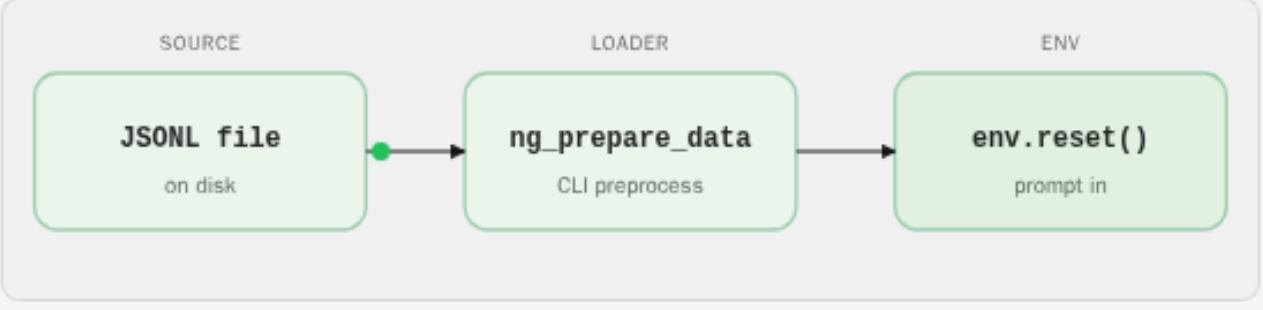
ORS SERVER

PROMPT get_prompt() from server
SPLITS Yes (named)



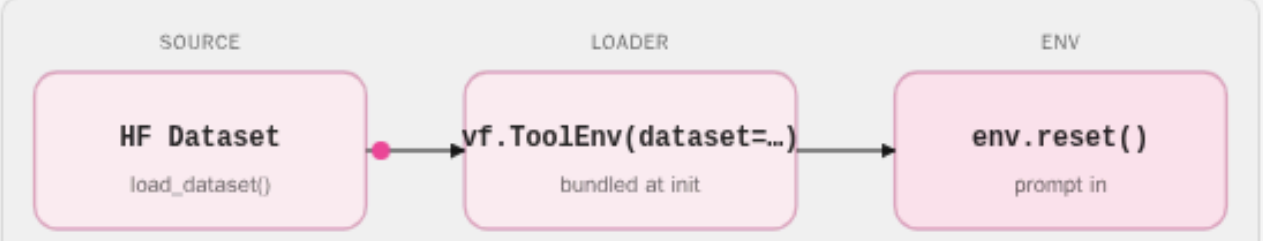
NeMo Gym JSONL

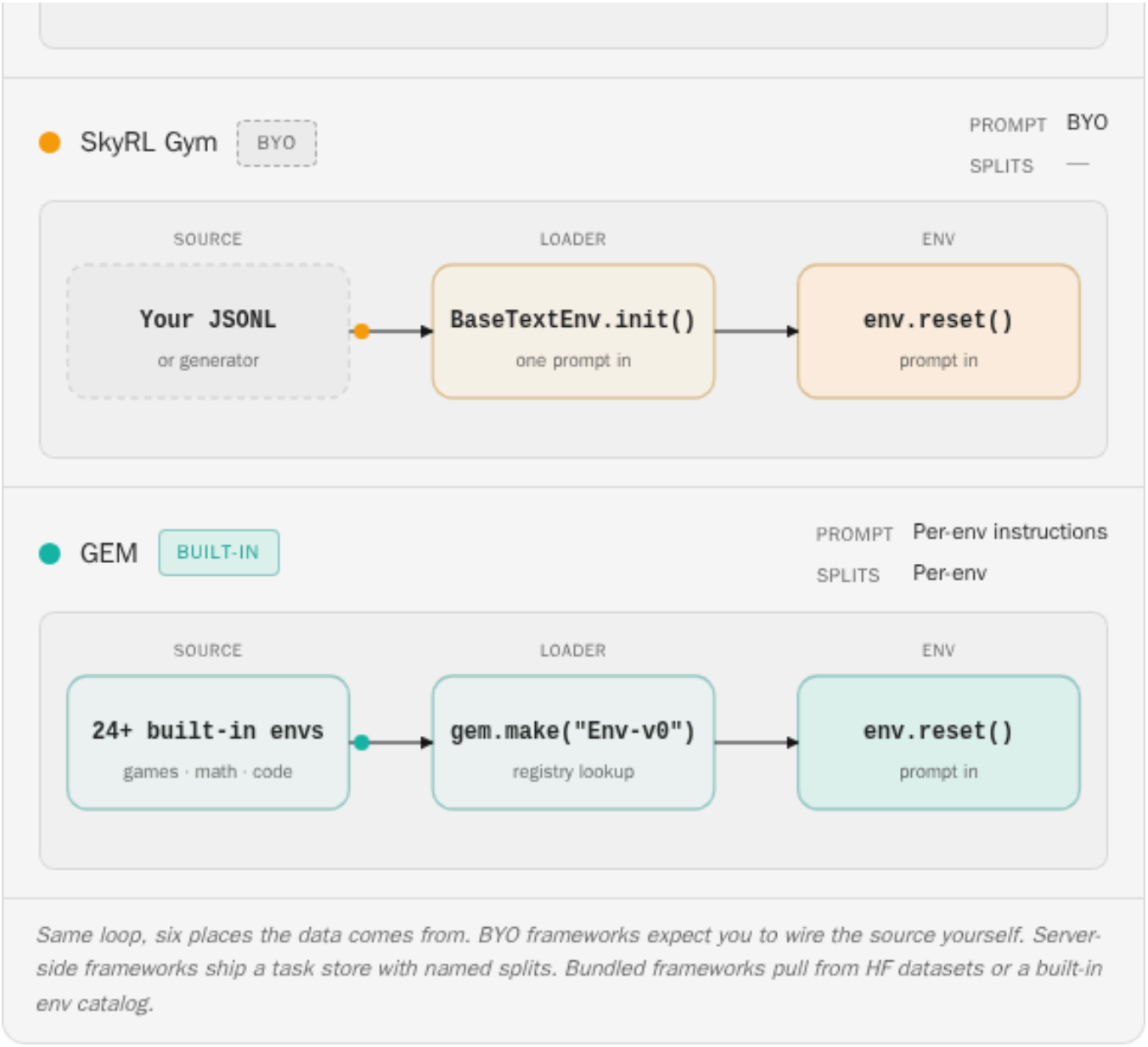
PROMPT In JSONL · responses_create_params
SPLITS Yes



Verifiers HF DATASET

PROMPT system_prompt param
SPLITS Yes (HF splits)





Same loop, six places the data comes from. BYO frameworks expect you to wire the source yourself. Server-side frameworks ship a task store with named splits. Bundled frameworks pull from HF datasets or a built-in env catalog.

The differences matter when you go from one task to a curated dataset with splits and versions. Frameworks that bundle a dataset abstraction also give you splits and versioning for free. BYO frameworks need you to wire that yourself.

TASKS & DATASETS · 6 FRAMEWORKS

CAPABILITY	OPENENV	ORS	NEMO GYM
Source	BYO	Server-side task DB	File-based J
Loader	Trainer loop · feeds `reset()`	`list_tasks(split)` at runtime	`ng_prepare_`
Prompt template	BYO	`get_prompt()` from server	In JSONL · `res`
Splits	—	Yes · `list_splits()`	Yes · per file
Versioning	Up to you	Server-versioned	Filename / com

A TASK IS MORE THAN A ROW

We say “tasks” rather than “dataset rows” on purpose. A row is one input plus one expected output. A task, especially in agent and coding environments, is a bundle of artifacts that has to land in the sandbox before the rollout can start. OpenReward’s SETA, 1,376 terminal-agent tasks on the Open Reward Standard, is the cleanest published example: every task ships a `task.yaml` (prompt and metadata), the data files the agent operates on, a `tests/test_outputs.py` harness the environment runs after submission, and a `weights.json` that turns the test outcomes into a scalar reward. SWE-Bench rows go further: each row carries `repo`, `base_commit`, `problem_statement`, and `test_patch`, the environment clones the repo at that commit, applies the test patch, and grades the agent’s diff with `FAIL_TO_PASS` / `PASS_TO_PASS`.

These bundles usually live behind the dataset row, in S3, an HF dataset repo, or a tarball. The environment pulls them on episode start, drops them into the sandbox, scores, then tears down. The dataset abstractions in Verifiers, GEM, ORS, and NeMo Gym track the prompt and the answer column cleanly; the file tree and test harness ride alongside as that artifact bundle.

COUPLING: WHO OWNS THE TASK SPEC

Frameworks split on how strict the task spec is, and that strictness is what lets a task hop between training runs without rewiring.

- Coupled. Verifiers expects an HF `Dataset` with a `prompt` column and optional `answer` or `info` columns; GEM ships built-in environments with their own loaders; ORS and NeMo Gym pin the schema on the server side. The Environments Hub and OpenReward go further and standardise the whole package, the layout, the scoring contract, even the wheel-based packaging, so any task that fits the spec runs in any environment that follows it.
- BYO. OpenEnv and SkyRL Gym leave the dataset up to you. Prompts come in from any source, the environment doesn’t look at the schema, but every new source costs a little integration.

> *Note: who owns the data transformation. Coupling means the environment dictates the spec and you transform your raw data to fit. Concretely:*
>
> - *SWE-Bench JSONL with `{repo, base_commit, problem_statement, test_patch}` into a Verifiers `Dataset` with a `prompt` column and `info` carrying the patch metadata. The mapping is yours, written once.*
> - *A Kaggle Q&A dump into OpenReward-style task folders, one per row, each with `task.yaml`, `data/`, and `tests/`. One script fans the JSONL out into the layout.*
> - *HumanEval or GSM8K into an environment that already ships a loader for it. No transformation needed, the framework already knows the shape.*
>
> *The payoff for writing the mapping once is that the same environment runs unchanged across every task that fits the spec, and someone else's environments become reusable for your training run instead of one-off scripts.*

If you only care about one task, the spec is overhead. If you're training across many tasks or pulling in benchmark suites, the spec is what makes that work without per-task glue.

Dimension 7: Ecosystem & maturity

Who's behind it and how production-ready is it?

The frameworks land in very different spots on the maturity curve, who built it, where the community lives, what's already shipped in production. The matrix below pins those facts side by side, with the production-usage column tinted by kind (deployed in production, shipped as a managed platform, or research-stage).

ECOSYSTEM & MATURITY · 6 FRAMEWORKS

CAPABILITY	OPENENV	ORS	NEMO GYM	VERIFIE
Creator	Meta PyTorch	General Reasoning (GR Inc)	NVIDIA	PrimeIntell
PyPI	`openenv-core`	`ors-sdk`	Git only	`verifie`
Python	≥3.11	≥3.10	≥3.12	≥3.11
Built-in envs	Example in repo	Example in repo	50+ (NVIDIA)	Example ir
Community hub	4k+ on HF Spaces	330+ on openreward.ai	—	~1k on En
Production usage	Meta	OpenReward	NVIDIA · Nemotron	PrimeInte

Dimension 8: Ease of adding new environments

How hard is it to go from "I have an idea for an environment" to "it's running in training"?

This matters both for humans writing environments by hand and for AI agents (like Claude) scaffolding them. The friction points are: how much boilerplate, how many files, how many concepts to learn, and how fast you can test.

By hand: The chart below plots each framework on the two friction axes (files to create × concepts to learn), with bubble size scaling to time-to-first-test and a solid outline marking frameworks that ship a CLI scaffolder. Bottom-left is the easy quadrant; top-right is the heavy-boilerplate quadrant.

Bottom-left = easy and lean (1-2 files, 3 concepts, ~5 min to test). Top-right = heavy boilerplate (4-6 files, 5 concepts, ~30 min). Bubble size grows with time-to-first-test. Solid outline means a CLI scaffolder ships (`openenv init`, `orwd init`, `prime env init`); dashed means start from scratch.

EFFORT TO AUTHOR · 6 FRAMEWORKS

CAPABILITY	OPENENV	ORS
Files to create	4-6 env, app, models, Dockerfile, pyproject	3-4 se
Concepts to learn	`MCPEnvironment`, `create_app`, `Observation`, `Rubric`, FastMCP	Envi
CLI scaffold	✓ `openenv init`	✓
Time to first test	~30 min Docker / server	~20
Test without training	✓ curl	✓

By AI (Claude Code, Cursor, etc.):

In-process frameworks require fewer files for AI to scaffold, one Python file, no Docker, no server, and pytest feedback is immediate. HTTP frameworks require more files, Docker

configuration, and deployment steps. We tested this: Claude scaffolded a complete Wordle environment across all 6 frameworks in a single session. The in-process ones (Verifiers, SkyRL, GEM) completed in minutes; the HTTP ones (ORS, NeMo Gym, OpenEnv) required additional iterations for Dockerfiles and deployment configs.

Dimension 9: Local & cluster setup

How easy is it to run environments on your own infrastructure?

Local development (laptop/desktop):

The shape of the laptop install differs in two ways: how much you have to install before anything runs, and what kind of state you have to babysit (a server vs. a Python process). The matrix below covers both.

	OpenEnv	ORS	
Needs Docker	✅ (for production)	✅ (for production)	✅ (f
Needs external service	Depends on backend (e.g., E2B)	Depends on backend	Depe
Deps install	`pip install openenv-core fastmcp`	`pip install ors-sdk`	`pip`
Heavy deps	FastMCP, FastAPI, Gradio	FastAPI, uvicorn	Ray,
Python version	≥3.11	≥3.10	≥3.1
Dep conflicts risk	Low	Low	High

Slurm cluster:

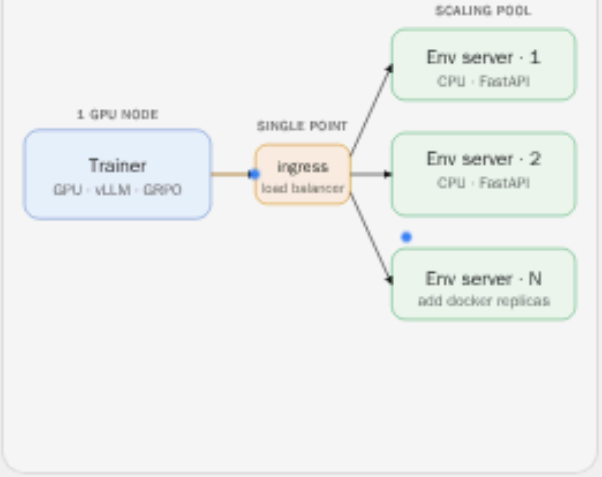
Once you leave your laptop, the question is who lives where. HTTP frameworks let you put the env on its own (often CPU-only) node and point any number of GPU training nodes at it, with a load balancer fronting a pool of env replicas you can grow. In-process frameworks put trainer and env in the same venv on the same GPU node, which is simpler to operate but means scaling the env is whatever orchestration story your training framework already has.

CLUSTER TOPOLOGY · TWO DEPLOYMENT SHAPES

Pause

HTTP / DOCKER DEPLOY

ENV SCALES HORIZONTALLY

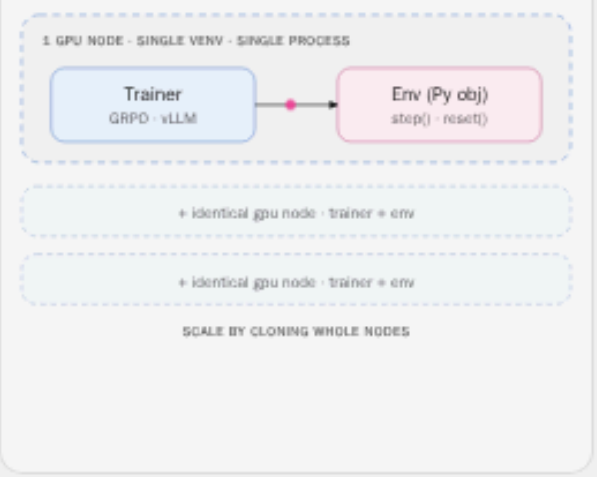


OpenEnv · ORS · NeMo Gym

One trainer, one ingress, many env servers. To add capacity you add Docker replicas behind the load balancer; the trainer sees a single endpoint.

IN-PROCESS DEPLOY

SCALING = ORCHESTRATION



Verifiers · SkyRL Gym · GEM

One process, no network, the call is a Python invocation. There is no shared "env service" to grow, so scaling falls to whatever orchestration the trainer already runs.

Same RL loop, two scaling stories. HTTP frameworks scale by cloning the env behind a single endpoint, training stays put. In-process frameworks rely on the orchestration layer you already have, more workers, or a remote sandbox.

Orchestration patterns · GEM uses `AsyncVectorEnv` per worker. Verifiers and SkyRL Gym use per-worker instances. Any of them can delegate compute to a remote sandbox (E2B, Modal) for true horizontal scale.

	HTTP Frameworks	
Env deployment	Deploy to HF Spaces or separate node; training nodes just need `requests`	Insta
GPU usage	Env server uses no GPUs (can be CPU-only)	Env s
Dep isolation	Clean, server has its own Docker/venv	Must
Multi-node	Server on one node, training on others	Singl

Dimension 10: Scaling & deployment

How do environments scale from development to production, and what are the concurrency limits?

RL training generates multiple rollouts per prompt, ideally in parallel, which means interacting with many environments simultaneously. In GRPO specifically, that's `num_generations` (typically 4-16) environments per prompt across the batch: with 64 prompts and `num_generations=8`, you have 512 concurrent environment instances per step. This section covers how the two deployment models handle that.

TWO SCALING MODELS

In-process frameworks (Verifiers, SkyRL Gym, GEM) create Python objects, startup takes `<1ms`, so 512 or even 10,000 instances are straightforward. Parallelism is handled by the training framework: each worker gets its own environment instance.

The real trade-off is orchestrational, not architectural. If the environment does its own compute (heavy parsing, simulation, in-process logic), it shares CPU with training and scaling means scaling training nodes, expensive GPU work for CPU tasks. But if the environment delegates the actual work to a sandbox or remote service (E2B, Modal, custom HTTP backend), the in-process class is effectively a thin client and the env scales independently of training. The catch is that you own that orchestration: choosing the sandbox provider, managing replicas, handling failover, load-balancing requests. HTTP frameworks bake this into the framework. In-process frameworks leave it to you.

Beyond orchestration, two things stay constant:

- Dependency conflicts: environment dependencies must coexist with training dependencies (TRL, vLLM, DeepSpeed, etc.) in the same Python environment. A framework that pins a conflicting version of `numpy` or `torch` can break training.
- Fault isolation depends on where the work happens: a crash in pure in-process env code can take down the training worker; sandboxed execution isolates failures naturally.

HTTP frameworks (OpenEnv, ORS, NeMo Gym) run as separate servers handling concurrent sessions. The environment and training scale independently, you can add cheap CPU servers for environments while keeping GPUs focused on training. The openenv-scaling benchmark tested how these scale across five infrastructure configurations. Since the server architecture is similar across HTTP frameworks (FastAPI + uvicorn + per-session state), these results are broadly representative.

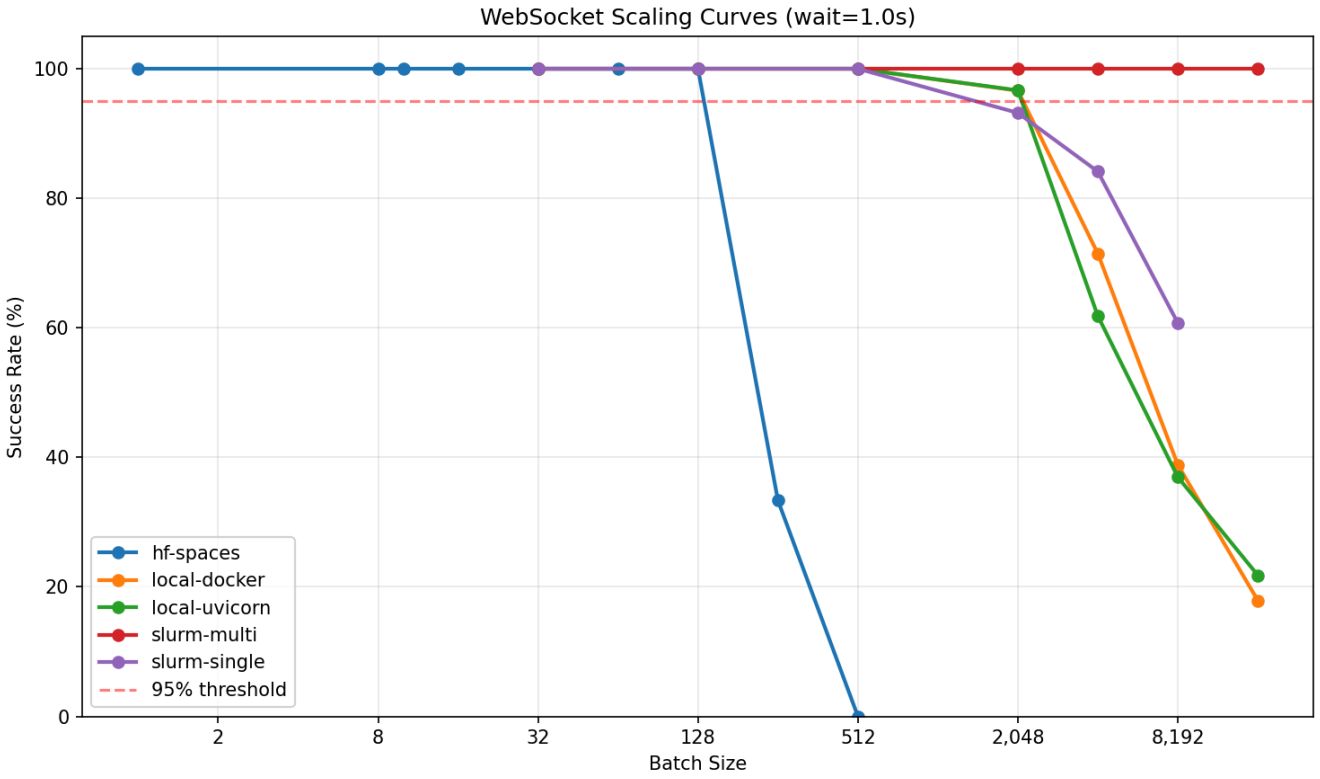
BENCHMARK RESULTS: HOW CONTAINERIZED ENVIRONMENT SERVICES SCALE

The openenv-scaling benchmark tested an environment deployed as a FastAPI server in a Docker container, across five infrastructure configurations. OpenEnv, ORS, and NeMo Gym all follow the same shape, a FastAPI app holding per-session state, packaged in the same image used for HF Spaces, so these numbers are broadly representative of any environment deployed as a containerized service. The benchmark itself runs OpenEnv's WebSocket mode; the per-protocol differences (WS / SSE / REST) matter less than the container-and-load-balancer story.

Maximum concurrent environments at ≥95% success rate (`wait=1.0s`):

Infrastructure	Cores	Max Concurrent	Batch/Core
Multi-node SLURM (2 nodes + Envoy load balancer)	96	16,384	170.7
Local uvicorn (8 workers)	8	2,048	256
Local Docker (same image as HF Spaces)	8	2,048	256
SLURM single-node (48 workers)	48	512	10.7
HuggingFace Spaces (free tier, cpu-basic)	2	128	64

> *Source: openenv-scaling benchmark. Full results, figures, and raw data in the repo.*



> *Note on what's being measured: the benchmark deploys the environment as a long-running service: a FastAPI app behind uvicorn, packaged in the same Docker image used for HF Spaces. OpenEnv, ORS, and NeMo Gym all follow this pattern: persistent-session services running in containers, differing only in wire protocol (WS, SSE, REST+cookies). What scales is the service-in-a-container shape, not any specific protocol. Verifiers, SkyRL Gym, and GEM take the opposite shape, environment-as-process inside the trainer, which is why they don't appear in this comparison. The infrastructure scaling patterns (local → single-node → multi-node) apply broadly across the three service-shaped frameworks.*

LATENCY AT MAX LOAD

Where time is spent at maximum concurrency (`wait=1.0s`). The chart below stacks the three p50 phases per infrastructure, with a ▼ marker for the p99 total. Toggle to the table for the raw numbers.

LATENCY AT MAX LOAD · 5 INFRASTRUCTURES

connect p50 reset p50 step p50 total p99

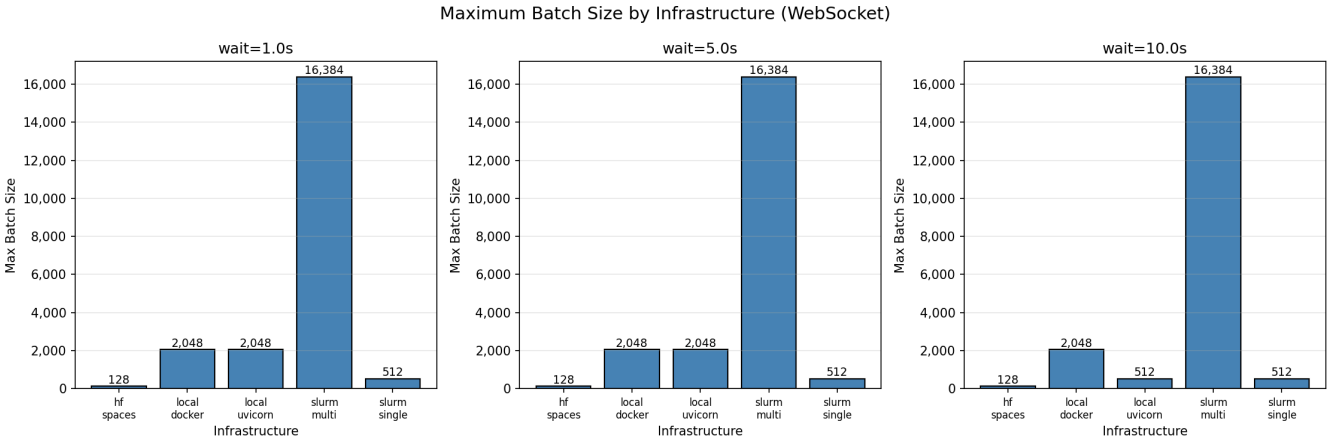
TABLE CHART

Sort by Total p99 (low → high)

INFRASTRUCTURE	CONNECT P50	RESET P50	STEP P50	TOTAL P99
SLURM single 48 workers	0.26s	0.04s	1.00s	**1.33s**
Local uvicorn 8 workers	0.58s	0.08s	1.05s	**1.95s**
HF Spaces free tier · 2 cpu	0.79s	0.10s	1.10s	**2.48s**
Local Docker same image as HF	1.38s	0.19s	1.05s	**2.90s**
SLURM multi 16K sessions	17.50s	2.25s	2.42s	**26.30s**

Stacked p50 segments (connect / reset / step) plus the p99 marker (▼). The SLURM-multi row is mostly connect-queueing across the Envoy proxy, every other row is dominated by step latency, the actual environment work.

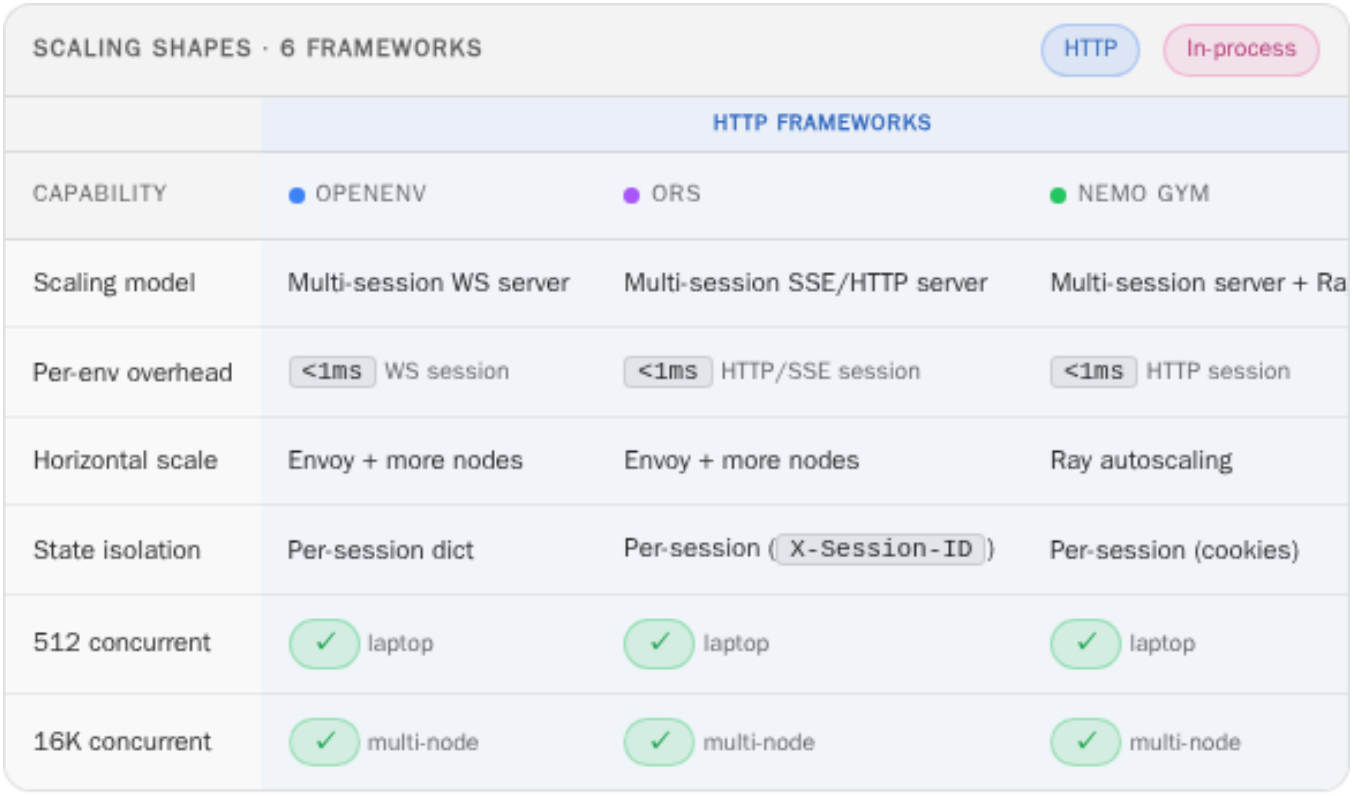
The multi-node p99 reflects connection queuing at 16,384 concurrent sessions across the Envoy proxy, absolute throughput (518 RPS) is the highest of any configuration.



KEY OBSERVATIONS

1. Docker adds no meaningful overhead: Local Docker and uvicorn reach the same 2,048 max batch.
2. Load balancing configuration matters: Before fixing Envoy, multi-node achieved only 128 max batch. After: 16,384 (128x improvement).
3. HF Spaces caps at ~128 concurrent sessions: sufficient for development and demos, and convenient since it's also the largest community catalog of pre-built environments to start from.
4. The server is rarely the bottleneck: even a laptop handles 2,048 sessions. The execution backend (sandbox creation, tool execution) dominates per-step latency regardless of framework.
5. Horizontal scaling is a load-balancer config problem, not a protocol problem: the 128 → 16,384 jump came from fixing Envoy's settings, not from changing the wire format. Sticky sessions (which WebSocket forces) make this harder to load-balance; for designs targeting thousands of envs, a stateless-per-request shape with a session ID has fewer footguns.

FRAMEWORK COMPARISON



Global comparison matrix

The previous chapter walked the ten dimensions one at a time, each with its own visualisation and matrix. This is the same data folded into a single sortable table, useful when you want to scan across frameworks without flipping between sections, or pull out a row for your own notes.

Four groups, Identity, API surface, Deployment, and Tooling, sit on top of each other so the table reads as four short tables instead of one long one. Cells are tinted by category. The filter chips along the top hide everything that doesn't match a constraint (e.g. *HTTP only*, *No CLI scaffold*), so you can narrow to the two or three frameworks that actually fit your setup. The copy icon on each row hands you that row as Markdown if you're tracking your own picks.

GLOBAL COMPARISON MATRIX · 6 FRAMEWORKS — **6** visible

FILTER: **HTTP only** | **In-process only** | **Has own trainer** | **No CLI scaffold** | Reset

CAPABILITY	OPENENV	ORS	NEMO GY
IDENTITY			
Creator	Meta PyTorch	General Reasoning	NVIDIA
Type	HTTP · MCP	HTTP · REST+SSE	HTTP · RES
Package	`openenv-core`	`ors-sdk`	`nemo_gym`
Python	≥3.11	≥3.10	≥3.12
API SURFACE			
Tool syntax	`@mcp.tool`	`@tool` + Pydantic	`app.post`
Observation	`str`	`ToolOutput(blocks)`	`Response`
Reward	`Rubric` system	Per-tool-call	`/verify`
Done signal	`Observation.done`	`finished` per call	Post-ep ver
DEPLOYMENT			
Deployment	Docker · HF Spaces	Docker · HF Spaces · OpenReward	Docker · HF
Docker needed	For production	For production	For produc
Scales independently	Yes	Yes	Yes
Heavy deps	FastMCP · Gradio	Minimal	`Ray` · Om
TOOLING			
CLI scaffold	`openenv init`	`orwd init`	—

Built-in envs	Community on HF Spaces	330+ on openreward.ai	50+ (NVIDI
Has own trainer	No	No	Yes

A few things worth noticing as you scan:

- No framework dominates every row. Each one wins a column or two and loses elsewhere. The picks come down to two or three dimensions you actually care about for your training run, usually deployment shape, episode control, and how strict the task spec is.
- The loudest split is still Deployment. HTTP server (OpenEnv, ORS, NeMo Gym) versus in-process (Verifiers, SkyRL Gym, GEM) cuts cleanly through almost every other row. Pick that axis first and the rest of the table narrows fast.
- Tooling maturity is uneven. CLI scaffolding, dataset registries, and observability are strong in some frameworks and absent in others, even ones that look identical on API surface. Worth checking before you commit.

The next chapter zooms back in, one framework at a time, with a profile card per framework that pulls together its rows from this matrix plus the patterns we noticed while building on it.

Framework profiles

Six profiles. What each framework is, what ships out of the box, and what it's good for. Structured facts live in the Framework inventory cards above; per-axis specifics live in Dimensions of comparison.

OpenEnv (Meta PyTorch)

OpenEnv is an MCP-based contract for building RL environments. It ships the protocol, session and transport layer (WebSocket), and a composable `Rubric` reward system (`LLMJudge`, `WeightedSum`, trajectory rubrics). Tasks, datasets, and execution backends are left to you. Good for thin, swappable env interfaces and tooling that needs to stay MCP-compatible.

ORS: Open Reward Standard (General Reasoning)

ORS is a standard API for tool, task, and reward shapes: `@tool` decorators, `ToolOutput` responses with inline per-step rewards, and server-side task management via `list_tasks(split)`. The openreward.ai platform hosts 330+ environments built on top of the spec. Good for plugging into a large pre-built environment catalog.

NeMo Gym (NVIDIA)

NeMo Gym ships FastAPI tool servers and a separate `/verify` endpoint for post-episode reward, with 50+ built-in environments and tight integration with NVIDIA's NeMo / Megatron training stack. Good for trajectory-level scoring and teams already on the NVIDIA stack.

Verifiers (PrimeIntellect)

Verifiers bundles the most components out of the box: datasets, tools, rubrics, rollout harness, trainer, and a `prime env init` CLI scaffold. The Environments Hub is the community registry for sharing and pulling in envs as wheels. Good for going from zero to training fast with a full toolchain.

SkyRL Gym (NovaSky / Berkeley)

SkyRL Gym is a Gym-style API with `BaseTextEnv` and `ToolGroup` classes, minimal dependencies, and the same library used to train SkyRL-Agent for SWE-Bench. Good for full control over the rollout loop with a familiar Gym mental model.

GEM (Axon-RL)

GEM follows the Gymnasium API most closely: `reset()` returns an observation, `step()` returns a 5-tuple `(obs, reward, terminated, truncated, info)`, and `AsyncVectorEnv` provides vectorised environments. Ships 24+ built-in games, math, and code environments. Good for carrying a Gymnasium / Atari mental model over to text agents.

Observations

Six frameworks, same environment built across all of them. A few things popped out.

What stood out

The “environment” boundary is a design choice, not a standard. Some frameworks (OpenEnv, SkyRL Gym) hand you a thin tool interface and let you own dataset, reward, and trainer wiring. Others (Verifiers, GEM) bundle all four. Neither is wrong, it’s a control vs time-to-running trade.

HTTP vs in-process is the loudest fork. Sandboxed execution (code, shell, browser) wants HTTP, you scale env compute independently from training. Pure Python (games, math, text reasoning) wants in-process, zero RPC overhead, no infra to babysit. Pick this axis first, the rest of the table narrows fast.

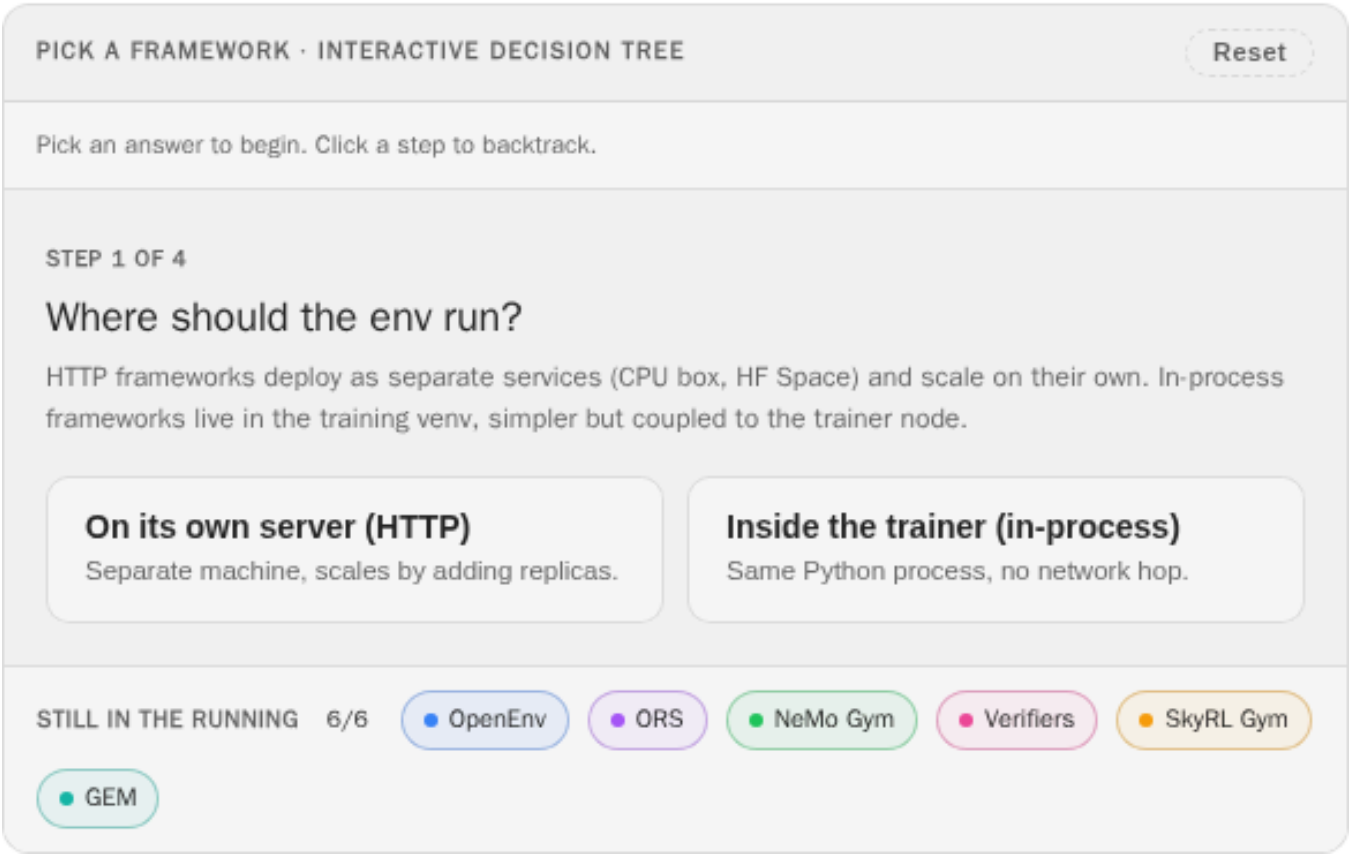
Dataset coupling cuts both ways. Bundled (Verifiers, ORS) means env and dataset come as one unit, can’t swap one without the other. Decoupled (OpenEnv, SkyRL Gym) makes you wire datasets yourself, but any task fits any env.

Reward timing matters more than reward content. ORS scores per tool call, NeMo Gym scores post-episode via `/verify`, OpenEnv and Verifiers use composable Rubrics. Per-call is clean for per-step feedback, awkward when reward only makes sense at the end. Post-episode is clean for trajectory-level scoring, no in-episode signal.

Execution backend dominates per-step latency. Sandbox creation and tool execution own most of the wall clock regardless of framework. Numbers in Dimension 10.

Picking one

If you’ve made it this far and just want a recommendation, walk the tree. Four yes/no questions narrow the field, breadcrumb up top lets you backtrack.



Framework-specific gotchas

Stuff that bit us while integrating each one.

Framework	Note
OpenEnv	MCP protocol is still evolving, API changes between versions may need adapter updates
ORS	aiohttp `base_url` handling in SDK client requires raw HTTP workaround
NeMo Gym	Requires Python 3.12; strict Pydantic validation on `/verify` returns 422 on unexpecte
Verifiers	Dataset required at init time, env and dataset coupled
SkyRL Gym	`step()` returns dict, not dataclass, type inconsistency to handle
GEM	`gem-llm` may not be installed, conditional imports needed

Where this leaves us

Snapshot is May 2026. The space is young, all six frameworks launched in 2025 and are moving fast. Six frameworks doing the same thing with different APIs is what early-stage

exploration looks like. Expect consolidation around fewer protocols (MCP, ORS) over the next year.

Each framework is the same thing wearing different clothes. The same environment ports across all six. What changes between them is how the env wires into the rest of training, not what it can do. You won't miss anything fundamental by picking one, what changes is convenience, which one is most pleasant depends on what's already in your stack.

Citation

For attribution in academic contexts, please cite this work as

```
Adithya S Kolavi, Lewis Tunstall, Leandro von Werra, Quentin Gallouédec, Amine Dirhoussi, Ben Burtenshaw, Sergio Paniego (2026). "The ultimate guide to RL environments: building and scaling them in the LLM era".
```

BibTeX citation

```
@misc{kolavi2026_the_ultimate_guide_to_rl_environments_building_and_scaling_them_in_the_llm_era,
  title={The ultimate guide to RL environments: building and scaling them in the LLM era},
  author={Adithya S Kolavi and Lewis Tunstall and Leandro von Werra and Quentin Gallouédec and Amine Dirhoussi and Ben Burtenshaw and Sergio Paniego},
  year={2026},

}
```