

NeMo Gym: Systems design

Brian Yu; Class of 22 (BS), 23 (MS)

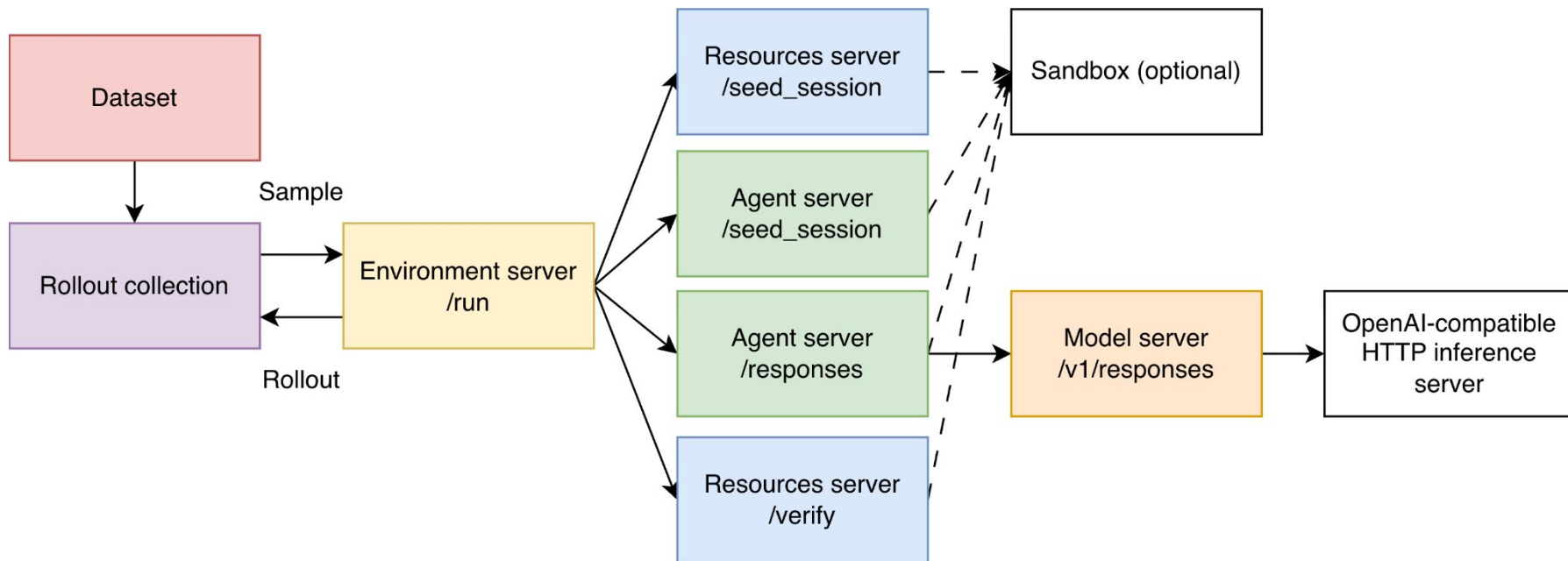
What is Gym and why is it important for Nemotron?

- Functionally, Gym is an HTTP rollout service. You spin Gym up, query it with jsonl rows from your dataset, and it returns rollout results.
- Gym repository is a place to contribute, experiment, and consume environments, for either eval or training.
- Organizationally, Gym connects Nemotron post-training, Frontier evaluation, NeMo FW engineering, Nemotron product, and folks around NV together.
- Gym helps Nemotron post-training scale: concurrency, environment diversity, collaboration.
- Gym recently graduated to production in Jun 2026, and now has two teams staffing it!

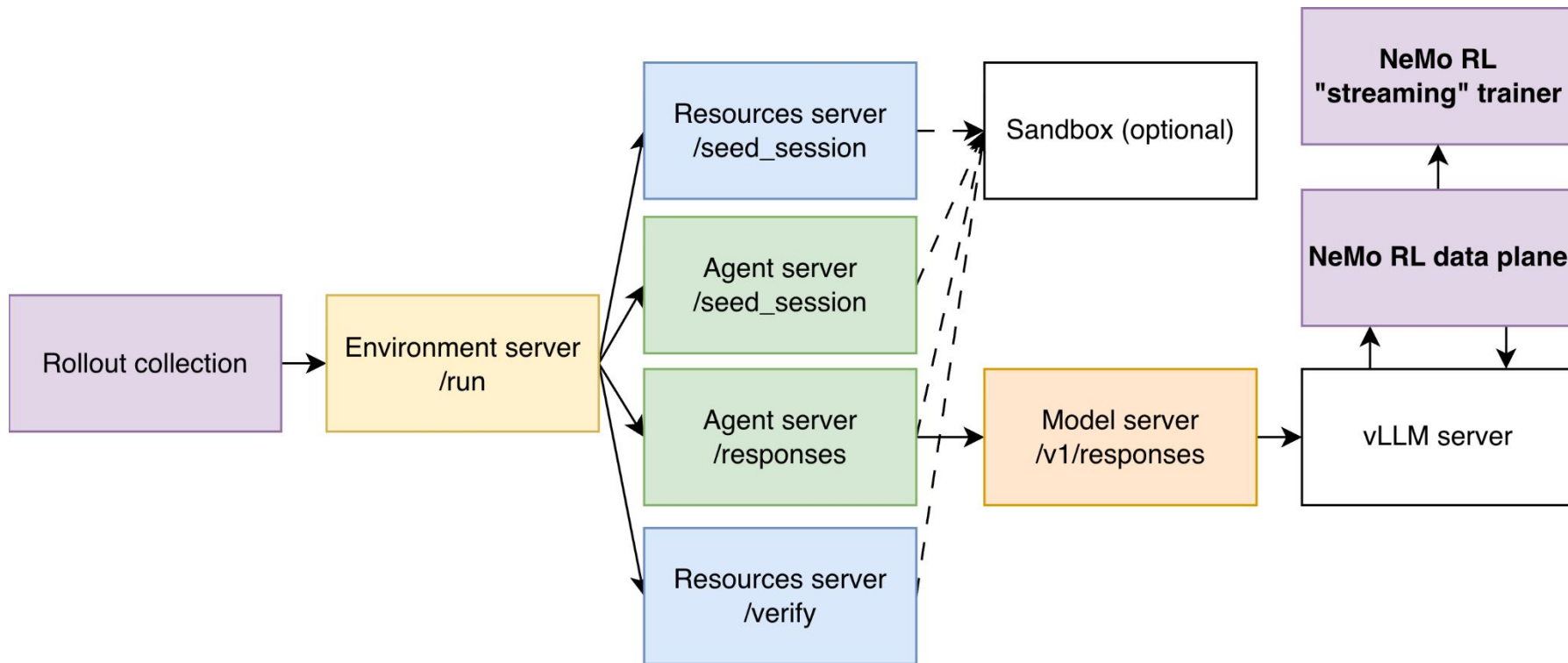
The Zen of Python, by Tim Peters (1999)

Beautiful is better than ugly.
Explicit is better than implicit.
Simple is better than complex.
Complex is better than complicated.
Flat is better than nested.
Sparse is better than dense.
Readability counts.
Special cases aren't special enough to break the rules.
Although practicality beats purity.
Errors should never pass silently.
Unless explicitly silenced.
In the face of ambiguity, refuse the temptation to guess.
There should be one-- and preferably only one --obvious way to do it.
Although that way may not be obvious at first unless you're Dutch.
Now is better than never.
Although never is often better than *right* now.
If the implementation is hard to explain, it's a bad idea.
If the implementation is easy to explain, it may be a good idea.
Namespaces are one honking great idea -- let's do more of those!

Gym data flow



Gym + RL data flow



3 core concepts organization and what we missed

When designing the existing Responses API agents, Responses API models, and resources servers structure, we considered the following key components:

1. Model concept (e.g. calling GPT 4 or Qwen)
2. Tool implementations
3. Verification
4. Systems of models and tool calls
5. Rollout lifecycle and orchestration

These key components are all those that typically change from user to user, use case to use case. As a result, they belong in semantically separate parts of our system.

What we missed: “Task” and “RL or eval Environment” components

Organizing the key components

Key context

1. There's a difference in “core infra” in Gym vs what our users would typically change.
2. Oftentimes tool implementations and verification are coupled together e.g. if the verification checks the state as a result of tool executions.
3. Oftentimes external gyms will couple the rollout orchestration (e.g. including verification) with the system response (e.g. without verification)

Rejected alternatives

1. 5 separate files or folders in the `nemo_gym/*` folder with each of these components held inside.
2. 5 separate top-level folders

Solution: three core components and three top-level folders: model concept, tool implementation + verification, and systems of models and tool calls + rollout orchestration.

Naming the core components

1. Model concept → “model”

- a. Everyone working on LLMs knows roughly what a model is
- b. Defined concretely: when we send context to a “model”, the model will run inference (and nothing else), and return a response in a structured format.

2. Tool implementation + verification → “resources server”

- a. Initially called this “environment” from classical RL, where an “agent interacts with its environment”
- b. But the term “environment” is overloaded
- c. Coining a new term incurs heavy confusion

3. Systems of models and tool calls + rollout orchestration → “agent”

- a. Community defined “agents” as loops/wrappers around model calls that also called tools

Servers: separate folders, separate venvs

Key context

1. When the Gym effort originally began in Aug 2025, we knew that the key was to scale up the diversity of our environments.
2. Dozens or hundreds of people may be working on their own Gym environments at one time.
3. People need to be able to work independently of others.
4. As the number of environments scales, the higher chance of things like dependency conflicts, etc.

Solution: Put each specific component implementation in separate folders in a fixed flat structure, and for each component to have its own environment.

How can components interact with separate deps?

Key insight: this is a very common problem in the wild and the solution is to use servers which talk to each other using a fixed format and channel i.e. HTTP and network sockets which are universally supported on machines today.

Negative consequences

1. Unintuitive for our users at the time (researchers) to learn about these servers. Coding agents have helped
2. It's difficult to trace the path of a request across server boundaries.
3. People now need to write code that is efficient enough for serving, not just efficient enough for their use case.

Positive consequences

1. Microservice architecture is the most generic thing ever - no modelling capacity issues by design
2. Avoid any forward compatibility issues by paying the cost of educating folks up front.
3. Any major problem we run into must have precedence, which makes it easier to do research for key insights e.g. sessions, etc.

Gym's native data format: OpenAI Responses

How should we represent these calls between the servers?

Key context

1. Need explicit support for agentic trajectories i.e. multiple reasoning, tool calls, chats, etc in a single trajectory.
2. At the time, the model builder community was still using OpenAI Chat Completions or Messages as the standardized format. The agent builder community is using bespoke formats that partially involve messages.
 - a. But Chat Completions cannot properly represent agentic trajectories, specifically the agentic return.
 - b. Chat Completions also doesn't contain explicit representation of reasoning, and the community would just insert a random reasoning_content field onto messages.
3. OpenAI had released the Responses API in Mar 2025 to facilitate agentic trajectories.

Solution: Use Responses API for Gym's native format. In hindsight, this format has held up for quite a long time, and only recently have we needed to start revisiting the schema due to e.g. sub agents and agent swarm, where we might migrate or augment via OpenAI Agents SDK format.

The Responses and Chat Completions types are copied explicitly in the Gym core infra

OpenAI Responses consequences

1. We now needed to silently convert between Responses, Gym's native format, and Chat Completions, the format accepted by vLLM and other inference engines.
 - a. Turns out, 10-12 months later this is how every inference engine is moving.
2. Given we keep to OpenAI format, we naturally get to upgrade and benefit to whatever OpenAI schema will evolve to be and support.
3. Why reuse rather than invent? Everyone building models or agents knows what Chat Completions format is, and Responses is a slight modification on top of that while retaining the majority of the flavor.
 - a. We also got very lucky that OpenAI had been doing exactly what the community had wanted to do i.e. they were leading.

Connection to RL

A key requirement for Gym is to easily integrate and connect with RL libraries like NeMo RL.

Key context: Our environment developers are domain experts in their respective domains e.g. safety, legal, etc and not experts in RL. They know how to run RL but that's typically the limit/stretch of their background.

Problem: We need to enable our environment developers to create environments without them having to touch token IDs

Solution: Silently propagate token IDs on requests/responses to the agent and model servers. The token IDs are additionally sent to the vLLM endpoint managed by NeMo RL so that we can perform fixes to guarantee on-policy training.

Better solution: The above doesn't work well for today's black-box harnesses, so we've transitioned to capturing token IDs in Responses API model servers into a data store.

Gym scaling studies

At the time we were designing Gym core, the rough concurrency we targeted was 65k concurrent requests.

Problem: While using AsyncOpenAI client, the maximum concurrency I was only able to achieve was around 128-256 concurrent requests.

- Key insight: Today's CPUs could more than handle this concurrency at even 1GHz. One CPU core should be able to comfortably handle 10k-100k concurrent requests.
- Companies like Google run against significantly higher concurrency than 256 requests, likely on the order of millions to 10s of millions. They must have a compute efficient way of handling all of their web traffic against a certain latency SLA.

AsyncOpenAI client used httpx as its async HTTP backend, and that httpx has a very inefficient $O(n^2)$ where n is the number of requests algorithm for connection pooling. Connection pooling is a critical feature of the HTTP backend since for Gym we typically have significantly less servers than we do requests.

Solution: Swapping to aiohttp HTTP backend with better connection pooling algorithm

Gym scaling studies

Serialization overhead: 65k concurrent requests finished in around 30 mins with 1 FastAPI worker. After using 16 FastAPI workers for the model server, this time dropped to around 20 mins (still not perfect).

Key context: our environment developers don't have the background to properly use this HTTP backends.

Solution: As a result, we wrapped all of these functionalities within a single function request.

A positive consequence of this design is that now if we want to upgrade Gym's networking protocol e.g. to use GRPC, a novel data store, or RDMA zero copy, we can simply override this request function as appropriate and parameterize it in a config.