MLE-Bench Redux

This project aims to understand and expand the MLE-Bench framework in the light of Fleet's customer's demands and ongoing production goals and challenges. This writeup is a supplement to the DEMO_README.md file that has more detail on building and using the repo.

Summary of original MLE-Bench findings and results

MLE-Bench runs containerized agent-environments to complete 50 different kaggle competitions and produce a leaderboard of scores. Tho the results are on now-obsolete models, they do hint at both to potential and problems with MLE agents. My shift in perspective is to push this MLE-bench from a fixed evaluation framework to something more sand-boxy, that can be leveraged for actually improving agent performance. Rather than a leaderboard, this version delivered the produced rollouts and also adds a different categorical dimension of tasks, called technique-tasks.

Conceptual Broad Strokes

For the assignment it was unclear to me what aspect of the problem were intended to be prioritized, so I identified 2 major aspects and developed both in broad strokes- a sketch, so to speak.

Systems Engineering strokes

On the systems engineering side, a teammate and I had briefly discussed the intended dev experience for fleet's customers, as well as their environment-ops, so to speak. Acknowledging this, I made a simple server and client 'sdk' to demonstrate how a single developer may interact with a similar service. Building in general for multiple users and multiple concurrent tasks is outside the scope of the project, though an obvious requirement for a real product. This is a non trivial engineering challenge.

Machine Learning strokes

On the machine learning side, I specifically responded to the 'improving on kaggle competition tasks' phrase in the prompt. Reading the original MLE-Bench paper, it clearly states that the models had trouble debugging and backtracking in many instances. Also if you look at the results, there is a lot of variance in the results with little clear rhyme or reason. What was also an interesting note from the paper was that there was no correlation between the model 'already having seeing the answer' and getting it right! Though it's important to note that these are now-obsolete models and modern ones will definitely perform better. However, even still, this clearly indicates the limits to our understanding of these models.

"More data" and "bigger models" may not be enough.

The couple of competitions I ran with AIDE and GPT-5 seemed to do substantially better, which does suggest that "more data" and "bigger models" go a long way toward solving the issues described in the paper. But even so, it feels a bit silly to rely entirely on whatever the LLM happens to do. From an LLM-research perspective, that's interesting, even canonical ; from an ML-engineering perspective, it ignores decades of accumulated, reliable knowledge about how to actually train models. It's stilly not to encode some of the meta dimensions of this…

“Technique tasks” as building blocks

So that became the motivation for introducing these medium-grained building blocks—not primitives in the literal code sense, but techniques and methodological tools we know to be universally true in machine learning. Things like when and why to do a train/validation split, when cross-validation is appropriate, what to watch out for with temporal adjacency, when normalization or scaling matters, etc. These are the small but crucial decisions that genuinely change model performance. Even basic exploratory data analysis falls into this category: while it feels almost silly to “teach” an AI what data looks like, it’s still worth probing whether the model can pick up structure or relationships from simple EDA that inform how it approaches the task.

The line between environment and agent

Another recurring question throughout the project was where exactly to draw the line between *environment* and *agent* – or more precisely, between what should be produced by the environment versus what should be produced by the agent. At what point is the environment merely a provider of datasets and scoring logic, and when does it become a scaffolding that actively shapes agent behavior?

From the ML side, this boundary turns out to matter a great deal. Adding certain features to the environment (eg technique-tasks, richer artifacts) seemed reasonable because these additions can actually help agents perform better. The goal isn’t just to assign a static score, but to generate artifacts that agents can learn from. Rollouts, in particular, can form a behavioral trace that future agents can use to improve their approach. Because these rollouts all derive from the same dataset, they naturally act as hints, demonstrations, or weak supervision signals for how to structure ML workflows.

More broadly, the aspiration is to push MLE-bench beyond being a leaderboard clone and toward becoming an agent-oriented MLE sandbox. In that framing, “rollouts” and technique-task artifacts are not simply evaluation outputs; they are a behavioral curriculum for future agents. They provide structure, examples, and context for aligning agent behavior with decades of practical ML knowledge. With more time, I’d absolutely try and assess this idea a bit more rigorously, please see “Future Directions, pt. 1”

How it works

Original MLE Bench works with an extensive CLI for running and grading tasks. Broadly, there is a prepare step, task completion step, and grading step. These are each done by separate components in the repo structure. The "bench" prepare step downloads the data from the kaggle competition and does any required data preparation. The agent completes the task and produces a results csv, the "bench" then grades and records the results. Each Competition is a single task in its own directory, with its own context and prepare/grading functions.

The Agent container gets the task directory mounted read-only and completes the task. Basically it makes a new container with the agent image and task directory. A grading server inside the container lets the agent validate submissions during the run. After the container finishes, the runner extracts artifacts and calls the CLI grader to produce the official report.

For Fleet, we've decoupled the interface to be a toy 'sdk' and added a server API that orchestrates runs via the Docker API. We've also tweaked the artifacts produced to include rollouts, code, and other media for developer visibility, and also of course the addition of the technique-tasks (TBD).

The artifacts produced are returned by our bench server to the developer user. Grading always happens - after extraction, the grading process kicks off and produces the official score. All of this is then bundled together and delivered to the developer user via /runs/{id}/artifacts, though in the future can be shown on a web UI.

Anticipated Challenges

1) ***Wrestling with containers***
2) ***Dev-environment Hiccups***

Unanticipated challenges

1) ***Wrestling with containers!!*** The gift that keeps on giving!
2) ***Dev-environment hiccups****:* I initially started in GitHub Codespaces because my new MacBook had just been migrated from an Intel machine and I expected hidden landmines. Codespaces worked very well until I hit an immovable 32GB storage ceiling. Even with a Professional plan and increased budgets, I couldn't expand the limit—despite documentation implying that I should be able to. So I had to move everything to my local machine, which predictably meant several hours of configuration, admin, and environment wrangling, also a lot of brittleness with the x86 containers on the mac arm architecture,
3) ***Late adoption of Claude Opus****:* I made the mistake of beginning the project using only external ChatGPT, and didn't switch to Claude Opus as a coding agent until late on day two. I'd never used a coding agent in Foundry before and did not anticipate just how colossal the difference would be between even a GPT-mini–level model and Opus. Which, amusingly, loops back to the original question raised in the MLE-bench paper: perhaps the earlier benchmark results weren't great simply because they weren't using something like Opus.
4) ***Github Copilot bug*** limits Premium model and does not expand limit even with adjusted budget! https://github.com/orgs/community/discussions/166810. So after a couple hundred requests to Opus, had to to grok code fast

Future Directions

The two major categories that surfaced throughout the project—the *systems* perspective and the *ML technique-task* perspective—remain the natural axes for future work.

1) ***Assess the "technique-task" hypothesis more broadly.*** Add more techniques tasks, and for a larger set of competitions. Get the rollouts for these all, and maybe one of the following: Add the technique-task rollouts to the instructions context for each competition, and see if it makes a difference in the performance of the agent. The implication being, that rewarding agents for using these 'techniques' in addition to whatever other major tasks/competition they are doing, would improve their performance!

2) ***Move toward a multi-user, multi-task orchestration model.*** Right now the benchmark server executes tasks sequentially in a single-process environment. A natural extension is to allow multiple developers (or automated agents) to spin up multiple tasks concurrently—coordinating container execution rather than running everything in-process. This turns the system into a more realistic evaluation service.
3) ***Improve the UX by surfacing artifacts directly.*** Rollouts, plots, logs, and diagnostic images could be presented through a lightweight web UI. For the demo we had prebuilt agents on the server, but a more production-ready structure would have the developer send github repo links instead of tags for which agent, which would trigger verification and a container build of the developer's agent, and run. This would make the system substantially more approachable and would better illustrate the value of behavioral artifacts beyond a single scalar score.

What was accomplished and delivered:

1) A functioning end-to-end evaluation pipeline (prepare → agent → grade) running inside isolated containers.
2) A mock SDK + benchmark service abstraction (the miniature "developer experience").
3) Rollout artifact generation and packaging.
4) The conceptual introduction of "technique-tasks" — mid-level ML behavioral constraints that can scaffold LLM agents.
5) Improvements to the benchmark environment (decoupling UX from CLI, clearer agent–environment boundaries).