CS329Z Assignment 2: Building an Evaluation Task

Version 26.1.0

CS329Z Staff

Fall 2026

1 Overview

In this assignment, you will build an evaluation task for a Language Model (LM) agent from the ground up. By the end, you'll have built a complete, carefully validated evaluation task. You'll decide what capability you want to measure, design a task that measures it, and analyze model performance on your task.

1.1 Brief Recap

Recall from lecture, the core principles and components of benchmarks for LM-based agents.

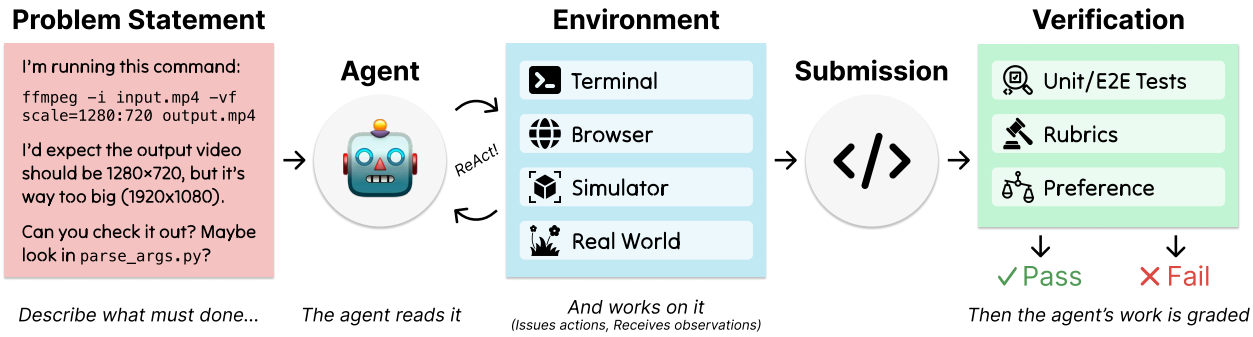


Figure 1: The core components of an agent evaluation: an agent receives a problem statement, acts in an environment, and its work is graded through verification.

A benchmark refers to a set of tasks that test an agent's capabilities or behavior. Per task...

1. A *problem statement* is given to the agent in the initial prompt.
2. The agent then sets out to solve the task in an *environment* (e.g., fixing a codebase via a terminal, accomplishing a web task via a browser). The agent goes through multiple turns of issuing actions and receiving observations against the environment.
3. To measure performance, we then grade the agent's submission (usually the environment's state when an agent finishes the task) against one or more forms of *verification*. There are multiple forms of verification, namely code unit tests or rubrics.

Together, these components define what the agent must do, what it can interact with, and how its performance is measured.

1.2 Why do benchmarks matter?

A key driver of progress in natural language processing (NLP) and artificial intelligence (AI) over the past decade has been the research community's adoption of shared benchmarks.

Benchmarks underpin a virtuous cycle. A well-posed benchmark establishes a common yardstick for measuring progress. Subsequent efforts then improve current methods' performance on the benchmark (often referred to as "hillclimbing"), and more importantly, identify a new set of limitations and gaps that in turn motivate the next benchmark.

1.3 What You Will Do

You will gain hands-on experience across the entire benchmark formulation, construction, and evaluation lifecycle in four parts:

- **Basics** (5%): You will take a guided deep dive into the Harbor Framework: how a task is represented, how an agent attempt is executed, how verification is isolated, and what artifacts an evaluation run produces.
- **Run a task** (10%): You will run models on task instances from several existing benchmarks (e.g., SWE-bench Verified, Terminal-Bench 2.1). Along the way, you will see how a task is represented in code, and how that representation shifts with the choice of formulation.
- **Make a task** (70%): You will author one task and evaluate `mini-SWE-agent` with Gemini 3.8 Flash on five independent attempts. You'll design the task's problem statement, environment, and verification, then analyze what makes the task valid, realistic, reproducible, and challenging.
- **Peer review** (15%): After task submissions close, you will download three classmates' tasks from a read-only course gallery, review each one using a provided template, and submit the completed reviews to a separate Gradescope assignment.

The assignment is worth 100 points.

1.4 Submission

There are two Gradescope submissions:

- **Task submission (due Monday, Nov. 23)**: your repository, uploaded with Gradescope's GitHub integration. It holds your task (with its five model-run artifacts and `report.pdf`) and your Section 2–3 write-ups; we grade everything from that snapshot.
- **Peer review (due Wednesday, Dec. 2)**: `peer_review.pdf`.

> ⚠ **On AI Assistance**
>
> You may use Claude Code, Codex, Copilot, or any other AI tool while working on this assignment. We will check understanding with oral quizzes, in which we ask you questions about your design and code. You should be able to explain, and you are responsible for, everything in your submission.

2 Basics

In this section, we'll download the assignment starter code and walk through the technical fundamentals of building a benchmark.

2.1 Setup

We use `uv` to manage Python environments and dependencies. Clone the assignment repository, then run any command with `uv run`, which resolves the environment on first use:

```
git clone https://github.com/cs329z/assignment2-harness.git
cd assignment2-harness
```

Push your work to your own *private* GitHub repository (not a public fork), so classmates cannot see your task before the deadline; you will submit that repository.

Next, you will have to use API credits to run models on tasks. Copy the template and fill in either `GEMINI_API_KEY` or `OPENROUTER_API_KEY`:

```
cp .env.example .env
```

> 💡 **Task Scope**
>
> As you know by now, "agents" can refer to a spectrum of systems. For this assignment, we'll focus primarily on benchmarks for *coding* agents, due to their relative maturity and lineage in the AI agents evaluation space.
>
> We'll be using the `mini-SWE-agent` scaffold. `mini-SWE-agent` is a simple, lightweight coding agent that exposes a single `bash` tool to the model.
>
> Harbor installs `mini-SWE-agent` in each task container without pinning its dependencies. Our commands load it through the provided `pinned_agent.py`, which installs exact versions; you do not need to modify it.

2.2 Building Blocks

Let's get familiar with fundamental infrastructure that supports how we implement and run contemporary LM benchmarks:

- **Docker**: Docker runs software in isolated, reproducible environments called containers.
- **Harbor Framework**: A widely used package supplying the infrastructure for defining coding agent tasks and evaluating LMs against them.

2.2.1 Docker ensures reproducibility

Imagine you write and evaluate LMs on a task locally on your MacBook. The tests pass, scripts run, and all seems well. But when you send the task assets to your friend, they can't reproduce your

findings. Why? Perhaps it's because you're using a different operating system. Or the versioning for Python and some dependencies are slightly off. Or you didn't install a necessary `.so` library.

Docker addresses this problem by running each attempt inside a **container**: a controlled workspace with a specified operating system, dependencies, and starting files.

Problem 1 (`docker-setup`): Set up Docker

Follow the installation instructions for your operating system.

- Mac installation
- Linux installation

Once it's installed, take some time to make sure you understand Docker and terms such as *Dockerfile*, *image*, and *container*.

Deliverable: Pull the `ubuntu:22.04` image, start an interactive container, and confirm its operating system from inside the container. Record your commands and output in `writeup/2_basics/responses.md`.

2.2.2 Harbor defines evaluation primitives

Reproducible containers solve only one part of running an evaluation. A benchmark also needs consistent conventions for packaging tasks, invoking agents, verifying work, and recording results.

Harbor provides these conventions and orchestrates the evaluation lifecycle. It gives the problem statement, environment, and verification components in Figure 1 a standard on-disk representation.

Once a task is defined in this structure, Harbor provides the additional wiring to launch an agent attempt, run the corresponding verifier, and save the resulting reward and artifacts.

In this assignment, Harbor orchestrates the evaluation, Docker provides the task environment, and mini-SWE-agent is the agent scaffold being evaluated.

Problem 2 (`harbor-setup`): Set up Harbor

From the assignment repository, install the pinned dependencies and inspect Harbor's command-line interface:

```
uv sync
uv run harbor --version
uv run harbor --help
docker version
```

Do not run an evaluation yet. From the help output, identify the commands used to start an evaluation job and browse completed results.

Deliverable: Confirm that `harbor --version` runs and that the Docker client can reach the Docker server. Record your output and answer in `writeup/2_basics/responses.md`.

3 Run Example Tasks

In this section, you'll be running and studying three tasks that effectively trace the lineage of coding evaluations from 2021 to present. As you run each task, take time to study the task, understand its components, and inspect how an agent performs on the task.

3.1 Task 1: HumanEval

Take a look at the `tasks/humaneval-0/` folder, which contains a local port of a HumanEval task. Recall from Figure 1 the key components of a task. See that these key conceptual components map to specific assets in this folder!

```
tasks/humaneval-0/
├── task.toml                 # Metadata and execution settings
├── instruction.md            # Problem statement shown to the agent
├── environment/
│   ├── Dockerfile            # Builds the agent's starting environment
│   └── humaneval_0.py        # Initial task asset
├── tests/
│   ├── test.sh               # Verifier entry point
│   └── test_outputs.py       # Behavioral tests
└── solution/
    └── solve.sh              # Reference solution used by the oracle
```

Problem 3 (`run-example-task`): Run the example task

First, inspect each component of `tasks/humaneval-0`. Then run its reference solution through the complete Harbor evaluation pipeline:

```
uv run harbor run \
--path tasks/humaneval-0 \
--agent oracle \
--n-attempts 1 \
--n-concurrent 1 \
--job-name humaneval-oracle \
--jobs-dir writeup/3_run_example_tasks/artifacts \
--yes
```

Run `uv run harbor view writeup/3_run_example_tasks/artifacts` to inspect the outcomes. Locate the verifier logs and confirm that the reference solution received full reward.

Deliverable: Check that `writeup/3_run_example_tasks/artifacts` contains the outcomes of running evaluation for this problem.

In `writeup/3_run_example_tasks/responses.md`, answer each of the following questions

in 3 to 5 sentences:

- Which files are available to the agent while it works, and which files become available only during verification? Why does this separation matter?
- Run `uv run harbor run --help`. Name two agent scaffolds other than `oracle` and two environment backends other than Docker. What distinct role does each category play?
- Trace the verification process after the agent finishes: which file does Harbor execute, where do the private tests come from, and how is the final reward communicated back to Harbor?

You just ran your first coding agent task! This is a task instance from HumanEval, introduced alongside Codex by Mark Chen and colleagues at OpenAI, helped establish the first widely adopted evaluation designed specifically around an LM's ability to write code [1]. The benchmark became the starting point for a lineage of coding evaluations that later expanded toward harder, more realistic software-engineering work.

```python
def encode_cyclic(s: str):
    """
    returns encoded string by cycling groups of three characters.
    """
    # split string to groups. Each of length 3.
    groups = [s[(3 * i):min((3 * i + 3), len(s))] for i in range((len(s) + 2) // 3)]
    # cycle elements in each group. Unless group has fewer elements than 3.
    groups = [(group[1:] + group[0]) if len(group) == 3 else group for group in groups]
    return "".join(groups)

def decode_cyclic(s: str):
    """
    takes as input string encoded with encode_cyclic function. Returns decoded string.
    """
    # split string to groups. Each of length 3.
    groups = [s[(3 * i):min((3 * i + 3), len(s))] for i in range((len(s) + 2) // 3)]
    # cycle elements in each group.
    groups = [(group[-1] + group[:-1]) if len(group) == 3 else group for group in groups]
    return "".join(groups)
```

Figure 2: An example HumanEval problem. The prompt provided to the model is shown with a white background, and a successful model-generated completion is shown in a yellow background.

Let's now run it with a model and see how it performs. For model runs in the rest of this section, we'll use Gemini 3.8 Flash as our model of choice. Depending on which key you're using, please use the corresponding model name:

- Gemini API: `gemini/gemini-3.8-flash`
- OpenRouter: `openrouter/google/gemini-3.8-flash`

Problem 4 (`run-example-agent`): Run a real agent

Now run the task with `mini-SWE-agent`.

```
uv run harbor run \
--path tasks/humaneval-0 \
--agent-import-path pinned_agent:MiniSweAgent \
--model <MODEL> \
--env-file .env \
--n-attempts 1 \
--n-concurrent 1 \
--job-name humaneval-agent \
--jobs-dir writeup/3_run_example_tasks/artifacts \
--yes
```

Inspect the trajectory, verifier logs, and reward with:

```
uv run harbor view writeup/3_run_example_tasks/artifacts
```

Deliverable: Check that `writeup/3_run_example_tasks/artifacts` contains the `humaneval-agent` job.

In `writeup/3_run_example_tasks/responses.md`, answer each question in 3–5 sentences:

- Summarize the agent's approach. Which files and commands did it use?
- Identify one decision in the trajectory that helped or hurt the final result. Cite the relevant step.
- Did the verifier reward accurately reflect the quality of the agent's solution? Why or why not?

3.2 Task 2: SWE-bench Verified

As you may have noticed, HumanEval tasks are both quite easy for current models to accomplish and not very representative of the work we now ask coding agents to perform.

Therefore, let's get ourselves more familiar with tasks representing benchmarks that state-of-the-art models have and continue to evaluate on.

The next task instance we'll take a look at is from the SWE-bench Verified task set.

SWE-bench, introduced by Carlos Jimenez and John Yang along with colleagues at Princeton, challenges LMs to resolve real-world GitHub issues [2]. Where HumanEval focused on self-contained functions, SWE-bench ushered in repository-scale software engineering tasks with much higher fidelity to real-world work. Its widespread adoption helped make real-world issue resolution a central measure of progress for coding agents.

To follow along, download the task from Harbor:

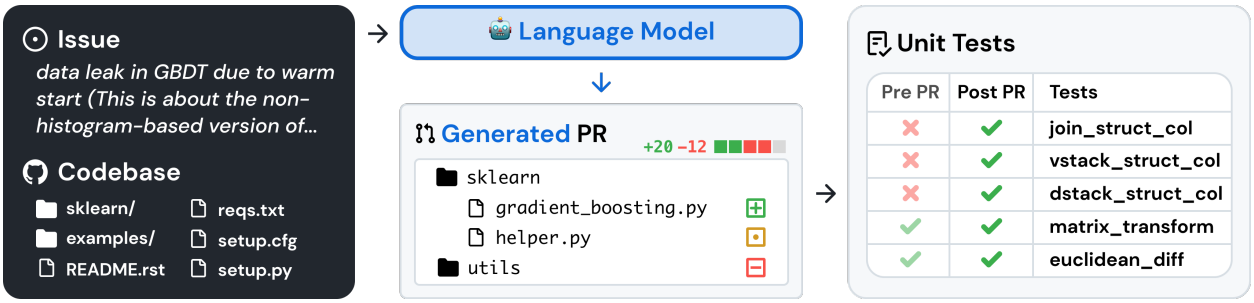


Figure 3: SWE-bench sources task instances from real-world Python repositories by connecting GitHub issues to merged pull request solutions that resolve related tests. Provided with the issue text and a codebase snapshot, models generate a patch that is evaluated against real tests.

```
uv run harbor tasks download swe-bench/django__django-11099@2 --output-dir tasks
```

This creates `tasks/django__django-11099`. Before running an agent, read the issue in `instruction.md` and briefly inspect the repository and verifier setup.

⚠ Apple Silicon Macs

This image is x86_64 only. On Apple Silicon, first run `export DOCKER_DEFAULT_PLATFORM=linux/amd64`; it then runs under emulation, so the first build is slow.

Problem 5 (`run-swebench-agent`): Run an agent on SWE-bench Verified

Run the task with `mini-SWE-agent` and Gemini 3.8 Flash:

```
uv run harbor run \
--path tasks/django__django-11099 \
--agent-import-path pinned_agent:MiniSweAgent \
--model <MODEL> \
--env-file .env \
--n-attempts 1 \
--n-concurrent 1 \
--job-name swebench-django-11099 \
--jobs-dir writeup/3_run_example_tasks/artifacts \
--yes
```

Inspect the resulting trajectory, verifier logs, and reward with:

```
uv run harbor view writeup/3_run_example_tasks/artifacts
```

Deliverable: Check that `writeup/3_run_example_tasks/artifacts` contains the `swebench-django-11099` job.

In `writeup/3_run_example_tasks/responses.md`, answer each question in 3–5 sentences:

- Summarize the agent's approach. How did it locate the relevant code, and what change did it make?
- Identify one point where the agent used repository context that would not exist in a self-contained HumanEval task.
- Compare this trajectory with the HumanEval trajectory. What made this task more difficult to understand, modify, or verify?
- Did the final reward accurately reflect the quality of the agent's solution? Support your answer with evidence from the verifier logs.

3.3 Task 3: Terminal-Bench 2.1

HumanEval and SWE-bench center on writing and modifying code, but many useful computer tasks extend beyond software repositories.

Terminal-Bench, a multi-institution effort led by Mike Merrill, Alex Shaw, Andy Konwinski, and Ludwig Schmidt, broadened coding-agent evaluation into long-horizon, end-to-end work performed through a command-line interface [3]. Unlike SWE-bench which was tethered to repositories, Terminal-bench encompassed *any* tasks that could be accomplished in the terminal, including system administration, security, machine learning, data processing, and scientific computing.

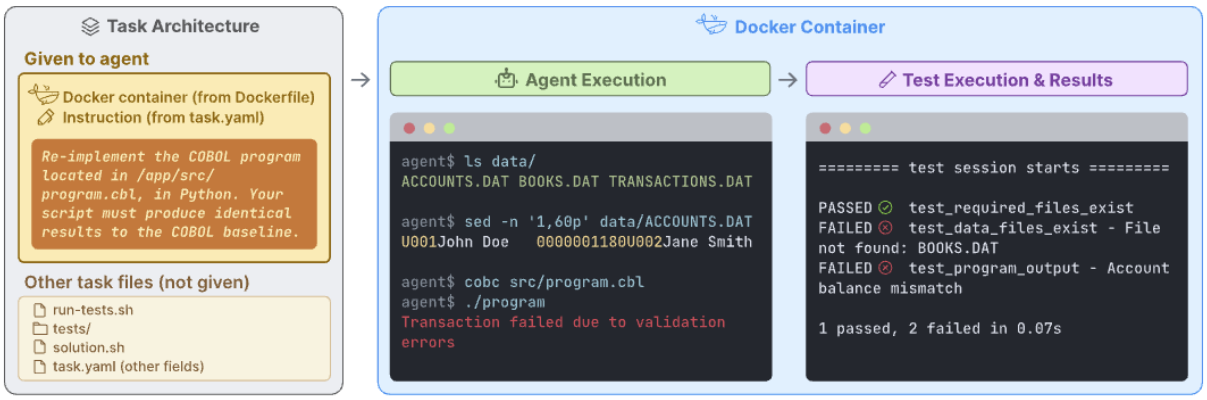


Figure 4: Terminal-Bench tasks place an agent inside a container and evaluate the resulting environment with hidden tests.

For this problem, you will run the `model-extraction-relu-logits` task from Terminal-Bench 2.1. Harbor classifies this task as hard and estimates that it would take an expert approximately eight hours to complete. When you run the model, we're not expecting the model to necessarily obtain a passing solution; your goal is to study how the agent approaches a difficult task and how much progress its trajectory captures.

First, download the task from Harbor:

```
uv run harbor tasks download \
terminal-bench/model-extraction-relu-logits@3 \
--output-dir tasks
```

This creates `tasks/model-extraction-relu-logits`. Read its `instruction.md` and briefly inspect its environment and verifier before running the agent.

Problem 6 (`run-terminal-bench-agent`): Run an agent on Terminal-Bench

Run the task with `mini-SWE-agent` and Gemini 3.8 Flash:

```
uv run harbor run \
--path tasks/model-extraction-relu-logits \
--agent-import-path pinned_agent:MiniSweAgent \
--model <MODEL> \
--env-file .env \
--n-attempts 1 \
--n-concurrent 1 \
--job-name terminal-bench-model-extraction \
--jobs-dir writeup/3_run_example_tasks/artifacts \
--yes
```

Inspect the resulting trajectory, verifier logs, and reward with:

```
uv run harbor view writeup/3_run_example_tasks/artifacts
```

Deliverable: Check that `writeup/3_run_example_tasks/artifacts` contains the `terminal-bench-model-extraction` job.

In `writeup/3_run_example_tasks/responses.md`, answer each question in 3–5 sentences:

- Summarize the agent's approach. What information did it gather, and what solution strategy did it pursue?
- Identify one point where the agent made meaningful progress and one point where its approach broke down or became unproductive.
- What evidence does the verifier provide beyond the scalar reward? Does it capture partial progress on this task?
- How did the open-ended terminal environment affect the agent's ability to plan, test its assumptions, and recover from mistakes?

3.4 Analyzing Agent Trajectories

By this point, you should have agent trajectories corresponding to all three tasks.

Next, you'll be presented with several questions that will exercise your ability to visualize and

analyze trends in trajectories, a core skill when it comes to communicating findings about model performance on benchmarks effectively.

Problem 7 (`analyze-trajectories`): Analyze agent trajectories

Use the saved trajectories from the HumanEval, SWE-bench Verified, and Terminal-Bench runs to create visualizations of agent behavior. You may choose the visual form and define reasonable measurements, provided that each figure makes a clear, evidence-based comparison.

Save each figure as a PNG in:

```
writeup/3_run_example_tasks/
```

Use descriptive filenames such as `bash-command-usage.png` or `trajectory-length.png`. Your figures should be understandable without opening the underlying Harbor artifacts.

Deliverable: Create three visualizations and briefly interpret (2-3 sentences) each one in `writeup/3_run_example_tasks/responses.md`.

For each figure, state what was measured, how it was computed, and what you conclude.

Choose three of the following questions:

- `effort-distribution.png`: How did the agent distribute its effort across activities such as inspecting files, editing code, running tests, and debugging?
- `bash-command-usage.png`: `mini-SWE-agent` performs its work through `execute_bash`. What command-line programs and command patterns did it use across the three trajectories, and how did their use vary by task?
- `exploration-to-implementation.png`: At what points did each agent appear to transition from exploring the task to implementing a solution? Define how your visualization identifies that transition.
- `repeated-effort.png`: Where did the agent repeat commands, revisit earlier hypotheses, or otherwise spend effort without making clear progress?
- `context-consulted.png`: How did the amount and type of repository or environment context consulted by the agent differ across tasks?
- `failure-recovery.png`: What patterns of failure and recovery appear in the trajectories? For example, when a command or hypothesis failed, what did the agent do next?
- `custom-analysis.png`: Choose another measurable property of the trajectories that you believe reveals something important about agent behavior. Explain why you selected it.

Many analyses can work; we will evaluate whether yours is clearly defined, faithful to the artifacts, and meaningfully interpreted.

4 Create Your Own Task

You will now design one original Harbor task from the ground up. The task may target any capability that can be exercised and verified inside its environment; we care more about the strength of a single, carefully developed evaluation than about covering a particular domain. For ideas, see the example tasks in Appendix A.

4.1 Task Requirements

Place your task at `tasks/<task-id>/` and follow the Harbor task structure introduced in Section 3. Your submission must satisfy the following requirements:

- The task has a clear problem statement and tests a capability that you can motivate.
- The task is meaningfully multi-turn: the agent must explore, act, and respond to what it observes rather than produce an answer in a single step.
- The environment builds from a clean checkout and contains all initial assets for the agent.
- The oracle receives full reward through the same verifier used for model attempts.
- The verifier meaningfully checks outcomes. File existence alone is not sufficient verification.
- The verifier produces a reward in $[0, 1]$. Binary, continuous, or discrete rewards are acceptable.
- The task resembles a plausible use of an LM agent rather than a puzzle made difficult through ambiguity, hidden assumptions, or fragile infrastructure.

We value tasks that are difficult because they require meaningful reasoning, tool use, or iteration—not because their instructions are ambiguous or their environments are fragile. Model failures count as evidence of difficulty only when the task is well-specified, solvable, and failed for reasons connected to the capability being evaluated.

4.2 Develop and Validate Your Task

Use the oracle throughout development to validate the complete evaluation pipeline:

```
uv run harbor run \
--path tasks/<task-id> \
--agent oracle \
--n-attempts 1 \
--n-concurrent 1 \
--job-name <task-id>-oracle \
--jobs-dir tasks/<task-id>/artifacts \
--yes
```

You should also test incorrect and incomplete solutions against your verifier. A strong verifier should reject plausible shortcuts while accepting solutions that genuinely satisfy the task, including correct approaches that differ from your oracle.

4.3 Run Five Model Attempts

Once the task is finalized, run five independent attempts with `mini-SWE-agent` and Gemini 3.8 Flash. Run all five attempts together and submit every artifact, not only favorable runs.

```
uv run harbor run \
--path tasks/<task-id> \
--agent-import-path pinned_agent:MiniSweAgent \
--model <MODEL> \
--env-file .env \
--n-attempts 5 \
--n-concurrent 1 \
--job-name <task-id>-gemini \
--jobs-dir tasks/<task-id>/artifacts \
--yes
```

We will run five fresh attempts using the same scaffold, model, and evaluation settings when grading. Our runs determine the empirical difficulty score, while your runs provide the evidence for your report.

4.4 Report and Deliverables

Write a one-page report at `tasks/<task-id>/report.pdf`. The report should identify the capability being tested, the environment and initial assets, the verification strategy, the outcomes of all five attempts, and the most informative success or failure. Start from the template in `write-up/4_create_tasks/report.tex`.

Deliverable: Your submission must contain:

- A complete Harbor task at `tasks/<task-id>/`
- A successful oracle artifact and all 5 model-attempt artifacts in `tasks/<task-id>/artifacts/`
- Complete one-page report at `tasks/<task-id>/report.pdf`.

Before submitting, run `uv run python check_submission.py` to check for missing files.

4.5 Grading

The task is worth 70 points. Each criterion is described in more detail below the table.

Criterion	What we will evaluate	Points
Reproducibility	Whether the environment builds, the task runs reliably from a clean checkout and the oracle receives full reward	10
Capability	Whether the task tests a clear, well-motivated capability with a readily apparent real-world application, and requires meaningful multi-turn work	15
Task–verifier fidelity	Whether the verifier reliably accepts correct outcomes and rejects incomplete or incorrect ones	15

Criterion	What we will evaluate	Points
Difficulty	Whether the task is frequently failed by frontier models and failures reflect meaningful capability limitations rather than task defects	20
Report	Whether the five outcomes are accurately reported and thoughtfully interpreted	10
Total		70

Reproducibility **10 points**

We will build your task from the submitted commit and run the oracle through the same verifier used for model attempts. Full credit requires that the environment builds, the oracle receives a reward of 1, and the oracle artifact and all five model-attempt artifacts are present.

Capability **15 points**

The task targets a specific capability that is useful in real world settings. Success should require that capability, not a trivial strategy or a single command. The task should also be meaningfully multi-turn: the agent must explore, act, observe results, and adjust.

Task–verifier fidelity **15 points**

The problem statement and the verifier should describe the same task. We will evaluate whether:

- Every behavior the verifier checks is stated or clearly implied in the problem statement, and every requirement in the statement is checked;
- The verifier clearly penalizes incorrect and incomplete solutions;
- The starting state of the environment, or an agent that does nothing, receives a reward of 0;
- The verifier accepts correct solutions that take a different approach from your oracle; and
- The verifier checks the outcome more thoroughly than just checking that a file exists or searching source files for keywords.

Rewards must lie in $[0, 1]$. They may be binary, continuous, or discrete.

Difficulty **20 points**

- **5 points**: The agent must succeed on the task in at most two of your five attempts.
- **15 points**: The agent's failures reflect a genuine shortcoming in the capability your task targets. The task should not be hard because of ambiguity, hidden assumptions, or contrived constraints.

Report **10 points**

We will check that all five outcomes are reported accurately and interpreted thoughtfully.

5 Peer Review

Designing evaluations also requires learning to distinguish genuine capability measurements from tasks that are merely difficult, brittle, or underspecified. After the task-creation deadline, you will review several tasks written by your classmates and apply the same criteria used to assess your own work. Writing your review with intent is also a sign of respect to your classmates.

5.1 Review Process

After submissions close, we will freeze each repository at its submitted commit and post the class's tasks to a read-only course gallery for download. Each download will include the task files, report, and submitted model trajectories. We will then assign you three tasks other than your own to review.

For each assigned task:

1. Download it from the course gallery and read the task description and report.
2. Inspect the task's instructions, environment, verifier, and oracle solution.
3. Examine the five submitted trajectories and verifier results.
4. Fill out one review in the template at `writeup/5_peer_review/`.

You may also run an assigned task, but doing so is not required. The gallery is only for downloading tasks; you do not submit anything there. Instead, compile your three completed reviews into one `peer_review.pdf` and submit it to the separate Peer Review assignment on Gradescope.

5.2 What to Evaluate

Each review should include:

- A concise summary of the capability the task measures;
- Ratings for each criterion in Section 4's grading rubric: reproducibility, capability, task–verifier fidelity, difficulty, and report;
- Evidence supporting your assessment, citing at least one specific task file, verifier check, or trajectory event;
- The strongest aspect of the task; and
- One concrete, actionable improvement.

Your ratings do not need to match ours. We will grade whether your review demonstrates an accurate understanding of the task and supports its conclusions with specific evidence.

Deliverable: Submit `peer_review.pdf` to the Peer Review assignment on Gradescope. The PDF must contain one completed review for each of your three assigned tasks.

5.3 Grading

Peer review is worth 15 points, with each of the three reviews worth 5 points: **1 point** for an accurate summary of the task and capability, **2 points** for a specific, evidence-based assessment, **1 point** for a concrete and useful proposed improvement, and **1 point** for completeness and clarity.

A Task Inspiration

Beyond the tasks in in Section 3, the Harbor Hub hosts thousands of tasks. The instances below span a wide range of capabilities, environments, and verification strategies. Click an instance to view its instruction, environment, and verifier, or a benchmark to learn more about it.

Instance	Benchmark	Description
`burntsushi--ripgrep.3b7fd44`	ProgramBench	Rebuild `ripgrep` from scratch given only its binary and documentation. [paper] [site]
`httpx-streaming-json-iteration`	DeepSWE	Add streaming JSON iteration to `httpx`, with content-type and charset handling. [site]
`task-69391d8d1ce51c407be1e531`	SWE Atlas (RF)	Refactor duplicated buffer-pool code in a Go repository into shared packages. [paper] [blog]
`task-6905333b74f22949d97ba998`	SWE Atlas (QnA)	Answer questions about a web app's API calls, Redux actions, and responsive CSS. [paper] [blog]
`polyglot_go_forth`	Aider Polyglot	Implement an evaluator for a subset of the Forth language in Go. [blog]
`job-shop-scheduling`	AlgoTune	Write a job-shop scheduling solver that runs faster than the reference. [paper] [site]
`gso-speedup-pandas-merge`	GSO	Speed up `pandas` merge/join operations without changing results. [paper]
`cybergym-arvo-368`	CyberGym	Craft an input that crashes a vulnerable program but not its patched version. [paper] [blog]
`corrupted-filesystem-recovery`	OpenThoughts TBLite	Repair a corrupted filesystem image and recover a hidden flag. [blog]
`tau3-bench__tau3-airline-25`	τ^3-bench	Resolve an airline customer's request over chat while following booking policy. [paper] [blog]
`gaia2-timed-1`	Gaia2	Reply to incoming emails on a fixed schedule and create calendar events. [paper] [blog]
`law433-mk-01-ec4c48f7`	APEX-Agents	Argue both sides of a contract dispute using only a residency agreement and handbook. [paper] [blog]
`corporate-ma-draft-ppa-memo`	Harvey LAB	Review an acquisition's closing balance sheet and draft a purchase price adjustment memo. [blog]
`court-form-filling`	SkillsBench	Fill out a California small claims court PDF form from a case description. [paper] [site]
`freecad-0280cda4fc`	CAD-Bench	Script a parametric FreeCAD part from a description, then apply a design edit. [site]
`gaia-find-chess-winning-move`	GAIA	Find Black's winning move from an image of a chess board. [paper]
`arcagi2-grid-transform-88e3`	ARC-AGI-2	Infer a grid transformation rule from examples and apply it. [paper] [blog]
`qcircuitbench-design-ternary-simon-circuit-n3`	QCircuitBench	Design a quantum circuit that solves a ternary Simon's problem. [paper]
`gravitational-wave-detection`	SkillsBench	Find a black hole merger signal in noisy detector data via matched filtering. [paper] [site]
`bix-diff-expr-mirna`	BixBench	Analyze patient miRNA data to measure differential expression. [paper] [blog]
`neo-orbit-determination`	TB-Science	Pick out a near-Earth asteroid's detections from raw astrometry and fit its orbit. [blog]
`dna-storage-codec`	TB-Science	Design an encoder and decoder for an error-prone DNA data storage channel. [blog]

Acknowledgements

This assignment was designed and built by John Yang for Stanford's CS329Z Engineering AI Agents course with help from Claude Opus 5.5. We owe a debt of gratitude to the various people who shared thoughts and feedback on this assignment. In particular we would like to highlight Michael Ryan, Diyi Yang, Shreyas Sharma, and Anantharaman Iyer.

If you would like to use this assignment for your course you are welcome to do so with brief attribution. It is released under the MIT license.

References

[1] Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, Alex Ray, Raul Puri, Gretchen Krueger, Michael Petrov, Heidy Khlaaf, Girish Sastry, Pamela Mishkin, Brooke Chan, Scott Gray, Nick Ryder, Mikhail Pavlov, Alethea Power, Lukasz Kaiser, Mohammad Bavarian, Clemens Winter, Philippe Tillet, Felipe Petroski Such, Dave Cummings, Matthias Plappert, Fotios Chantzis, Elizabeth Barnes, Ariel Herbert-Voss, William Hebgen Guss, Alex Nichol, Alex Paino, Nikolas Tezak, Jie Tang, Igor Babuschkin, Suchir Balaji, Shantanu Jain, William Saunders, Christopher Hesse, Andrew N. Carr, Jan Leike, Josh Achiam, Vedant Misra, Evan Morikawa, Alec Radford, Matthew Knight, Miles Brundage, Mira Murati, Katie Mayer, Peter Welinder, Bob McGrew, Dario Amodei, Sam McCandlish, Ilya Sutskever, and Wojciech Zaremba. Evaluating large language models trained on code, 2021.

[2] Carlos E. Jimenez, John Yang, Alexander Wettig, Shunyu Yao, Kexin Pei, Ofir Press, and Karthik Narasimhan. Swe-bench: Can language models resolve real-world github issues?, 2024.

[3] Mike A. Merrill, Alexander G. Shaw, Nicholas Carlini, Boxuan Li, Harsh Raj, Ivan Bercovich, Lin Shi, Jeong Yeon Shin, Thomas Walshe, E. Kelly Buchanan, Junhong Shen, Guanghao Ye, Haowei Lin, Jason Poulos, Maoyu Wang, Marianna Nezhurina, Jenia Jitsev, Di Lu, Orfeas Menis Mastromichalakis, Zhiwei Xu, Zizhao Chen, Yue Liu, Robert Zhang, Leon Liangyu Chen, Anurag Kashyap, Jan-Lucas Uslu, Jeffrey Li, Jianbo Wu, Minghao Yan, Song Bian, Vedang Sharma, Ke Sun, Steven Dillmann, Akshay Anand, Andrew Lanpouthakoun, Bardia Koopah, Changran Hu, Etash Guha, Gabriel H. S. Dreiman, Jiacheng Zhu, Karl Krauth, Li Zhong, Niklas Muennighoff, Robert Amanfu, Shangyin Tan, Shreyas Pimpalgaonkar, Tushar Aggarwal, Xiangning Lin, Xin Lan, Xuandong Zhao, Yiqing Liang, Yuanli Wang, Zilong Wang, Changzhi Zhou, David Heineman, Hange Liu, Harsh Trivedi, John Yang, Junhong Lin, Manish Shetty, Michael Yang, Nabil Omi, Negin Raoof, Shanda Li, Terry Yue Zhuo, Wuwei Lin, Yiwei Dai, Yuxin Wang, Wenhao Chai, Shang Zhou, Dariush Wahdany, Ziyu She, Jiaming Hu, Zhikang Dong, Yuxuan Zhu, Sasha Cui, Ahson Saiyed, Arinbjörn Kolbeinsson, Jesse Hu, Christopher Michael Rytting, Ryan Marten, Yixin Wang, Alex Dimakis, Andy Konwinski, and Ludwig Schmidt. Terminal-bench: Benchmarking agents on hard, realistic tasks in command line interfaces, 2026.