Snorkel AI

Terminus - Expert Contributor Onboarding

December 2025

Project Overview

Terminus is Snorkel's effort to build a high-quality dataset in the style of Terminal-Bench.

- Terminal-Bench is a benchmark for evaluating how well AI agents can accomplish complex tasks in a terminal environment
- Features multi-step tasks to be completed via a command line interface (CLI)
- Examples include:
  - Compiling and packaging a code repository
  - Downloading a dataset and training a classifier on it
  - Setting up a server

Your role as an EC is to create these tasks, along with an Oracle solution and associated tests that verify its correctness

- These tasks should be **challenging**, targeting a pass rate of <80% from SOTA models (e.g. GPT5 Codex, Claude Code)
- Tasks will be created locally and submitted to the Snorkel Expert Platform where they will undergo CI validation and peer review

Task Difficulty Ratings

The following ratings will be applied to tasks to categorize their difficulty. Ratings are based on the accuracy of GPT-5 and Sonnet 4.5 when attempting to complete the task.

Easy	Medium	Hard
60-80% Accuracy	40-60% Accuracy	< 40% Accuracy

NOTE: Since these ratings are based off model performance, you will not know how a task will be classified until after submitting.

If your task results in a “trivial” rating, you should iterate on the task to increase difficulty to be at least “easy”.

Task Components - Part 1 (Files) (old TB version)

- **Instruction/Task Description (task.yaml)**
  - Clear and self-contained description of the task to be accomplished
  - Includes references to relevant resources necessary for task completion, rules and constraints, and description of success criteria
  - Also contains metadata not available to the agent at runtime

- **Docker Environment**
  - **Dockerfile**
    - Base image that fully sets up environment, including all required tools, resources, and dependencies
    - Should run without privileged mode
  - **Docker-compose.yaml**
    - Config file that defines orchestration for the task
    - Should reference base image from Dockerfile and include any necessary environment variables or mounted resources

- **Oracle Solution (solution.sh)**
  - Step-by-step solution contained within a shell script
  - Reliably and accurately completes the task

Task Components - Files

- **Task Instructions (instruction.md)**
  - Clear and self-contained description of the task to be accomplished
  - Now a separate file from the task description

- **Task Description (task.toml)**
  - Configuration and metadata file (replaces task.yaml).
  - Uses TOML format with nested sections for task configuration, metadata, and environment settings.

- **environment/ Folder**
  - The environment definition must be placed in an environment/ folder. This prevents accidentally copying **task.toml**, **instruction.md**, or test files into the container.
  - *Required files based on environment type*
    - **For Docker Environment:** environment/Dockerfile **or** environment/docker-compose.yaml
      - **environment/Dockerfile**
        - Base image that fully sets up environment, including all required tools, resources, and dependencies
        - Should run without privileged mode
      - **Docker-compose.yaml**
        - Config file that defines orchestration for the task
        - Should reference base image from Dockerfile and include any necessary environment variables or mounted resources

Task Components - Part 2 (Files cont.) (old TB version)

- **run-tests.sh**
  - Script to execute and validate end-to-end evaluation of task
  - Coordinates running the agent within the defined environment, invokes test cases or success criteria checks, produces structured output that indicate task completion status
  - Must be deterministic and callable directly from project root
- **Python tests**
  - Series of deterministic Python scripts containing unit tests
  - Check task completion based on the final state of the environment
- **[Optional] Supporting DataFiles**
  - Any additional inputs (e.g. data, config files, etc.) that are required for the task
- **[Optional] Custom Tools**
  - Custom tools that can be used by the agent
  - Can be delivered in multiple ways, from source code files to companion containers

Task Components - Files (cont.)

- **Oracle Solution (solution/solve.sh)**
  - Expert-authored, step-by-step solution contained within a shell script
  - Reliably and accurately completes the task
  - Task must pass Oracle solution to be accepted

- **Tests (tests/test.sh)**
  - Script to execute and validate end-to-end evaluation of task
  - The test script must be named tests/test.sh (renamed from run-tests.sh).
  - This script installs test dependencies and verifies the agent completed the instruction.
  - The test script must produce a reward file in /logs/verifier/

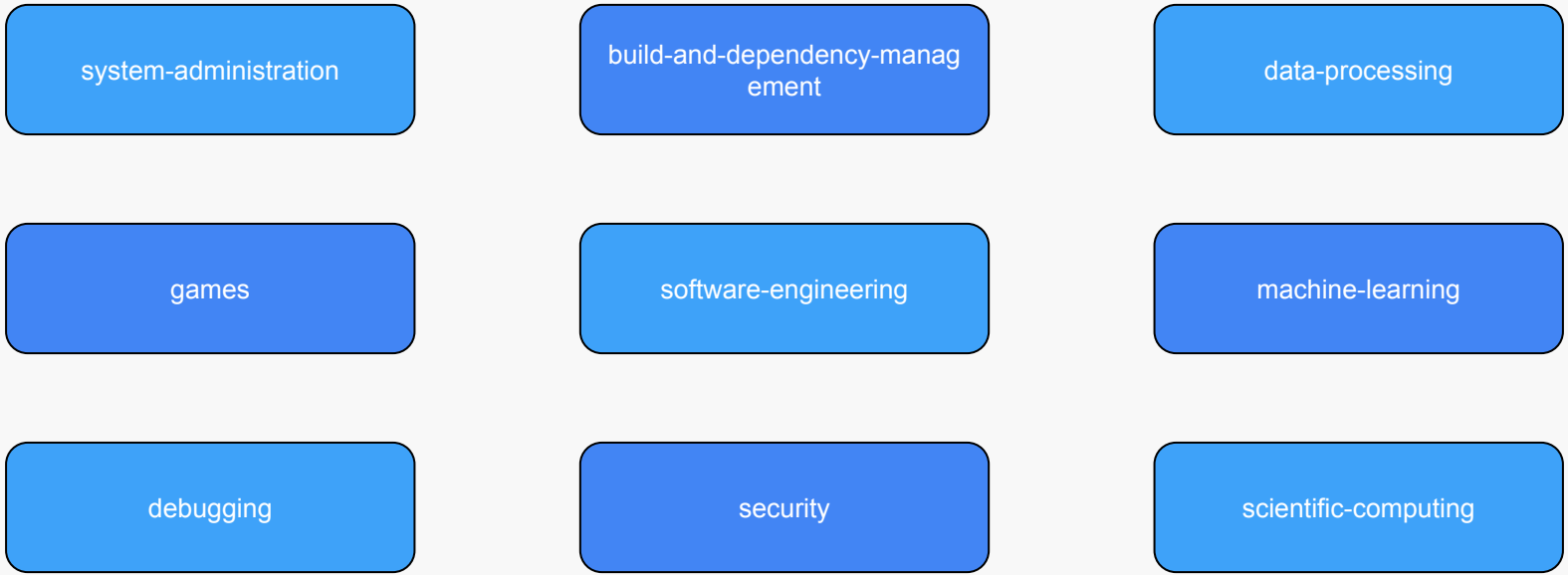
- **Python tests (tests/test_outputs.py)**
  - Series of deterministic Python scripts containing unit tests
  - Check task completion based on the final state of the environment

- **Optional Components**
  - **Supporting data files**
    - Any additional inputs (e.g. data, config files, etc.) that are required for the task
  - **Custom tools**
    - Custom tools that can be used by the agent
    - Can be delivered in multiple ways, from source code files to companion containers
  - **Multiple containers**
    - Complex multi-service environments

Task Components - Metadata

- **Pass Rate Difficulty (Model Performance)**
  - Determined by model performance post-submission
  - Leave as “unknown”
- **Difficulty (Time Estimate)**
  - Estimated time that completing the task would take for both a junior and a senior software engineer
  - "expert_time_estimate_min" and "junior_time_estimate_min"
- **Task Type**
  - Label for the task according to the standard 9-option taxonomy
  - Captures the primary theme, topic, or activity of the task
- **Task Tags**
  - 3-6 descriptive keyword tags that capture concepts relevant to the task (e.g. text-editing, vim, python, etc.)
  - No pre-existing taxonomy, come up with any relevant tags

Task Type Taxonomy



Example task.yaml (old TB version)

```
instruction: |
  Your task is to decompile a Python function in the format of bytecode, which is compiled in CPython
3.8, and save the function in decompiled.py in the same folder as task.yaml.
  The function name in the decompiled.py should be named "func".
  Note that the function is pickled using dill and serialized using base64 in the file called
func.serialized.
author_name: anonymous
author_email: anonymous
difficulty: hard
category: software-engineering
tags:
  - software-engineering
  - python
  - bytecode
  - serialization
parser_name: pytest
max_agent_timeout_sec: 360.0
max_test_timeout_sec: 60.0
run_tests_in_same_shell: false
disable_asciinema: false
estimated_duration_sec:
expert_time_estimate_min: 360
junior_time_estimate_min: 720
```

Example instruction.md

This file contains the task instructions in markdown format:

```
# Fix Empty Input Bug

Your task is to fix the bug in /app/main.py that causes
the application to crash when processing empty input.

## Requirements

The fix should:
1. Handle empty string input gracefully
2. Return an empty list instead of crashing
3. Not modify any other behavior

## Files

- Input: `/app/main.py`
- Output: Modified `/app/main.py`
```

Example task.toml

```
version = "1.0"

[metadata]
author_name,author_email -> "anonymous"
difficulty = "medium"
category = "debugging"
tags = ["python", "memory-leak", "debugging"]

[verifier]
timeout_sec = 120.0

[agent]
timeout_sec = 600.0

[environment]
build_timeout_sec = 600.0
docker_image = "some-org/some-name:some-tag"
cpus = 1
memory_mb = 2048
storage_mb = 10240
```

Task Design Requirements

- **Multi-Step**
  - Tasks requiring chaining multiple commands, handling intermediate states
  - Not solvable with a single command or episode of commands
- **Testable**
  - Must be fully specified so that agent can attempt to complete the task without ambiguity
  - Must be able to design unit tests to determine if final state of environment is correct
- **Unique**
  - Tasks should be unique and distinct from all existing tasks in the public Terminal-Bench benchmark
  - Should also be unique from other tasks submitted for this project
- **No Privileged Ops**
  - Tasks must not require root-level privileges or unsafe Docker settings like `--privileged`
- **Standalone**
  - Tasks must run to completion without additional user input after start
  - All parameters must be provided via files, flags, environment variables, or in the task.toml
- **Interacts with the Environment**
  - Looking for tasks that interact with the agent through the terminal
  - *For example: Avoid tasks based on data structures/algorithms.*
    - *OK for it to be a sub-component of the tasks, but shouldn't be the main component/focus*

Base requirements for a task to be accepted:

- All python unit tests all have to pass at least once across all 10 agent runs
- The Claude and ChatGPT agents cannot have 100% accuracy models can’t pass
- Oracle solution needs to pass
- NOP shouldn't pass

Task Submission Checklist

- All behavior checked in the test cases is described in the task instruction.
- All behavior described in the task instruction is checked in the unit tests.
- My test cases have informative docstrings that describe which behavior they check. It is hard for the agent to cheat on my task (e.g. by editing data files, looking inside files for strings that represent solutions, training on the test set, etc.).
- My task.toml was written by a human.
- My solution/solve.sh was written by a human (with minimal help from a language model).
- If the agent produces structured data (e.g. it is tasked with building an API) the exact schema is described in the task.toml or a separate file.
- If my task uses external dependencies (e.g. Docker images, pip packages, etc.) their versions are pinned to ensure reproducibility.
- I ran this task using agent with a powerful model (e.g., GPT-5). For failing runs (which are expected for harder tasks!), I've added an analysis below to confirm the task itself is valid. (Hint: tb tasks debug can help)
- I formatted and linted the task (Ruff).

High-Level Tasking Workflow

Tasking will be performed through the terminus-project-v2 project on the Snorkel Expert Platform.

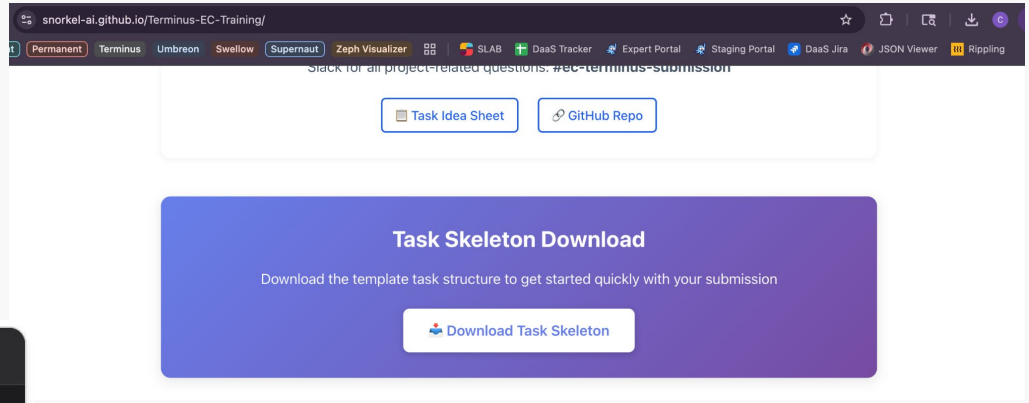
Once you are granted access, you should:

1. Install `harbor` so that you can use the commands used for running the agents and programmatic checks locally (pip install harbor)
2. Come up with a task idea or select one from the Task Gallery
3. Go to the training site and download the task file skeleton
4. Rename the task folder to match your intended task name
5. Create your task instructions, an Oracle solution that passes, and Python tests
6. Iterate on your submission until all CI/Evals pass
7. Create a ZIP file for your task folder
8. Submit your ZIP file on the Platform

We expect submission tasks to take 2-5 hours based on task difficulty.

Platform Workflow - Pt. 1

- Go to the Terminus EC Training Hub and download the ZIP file of the task skeleton

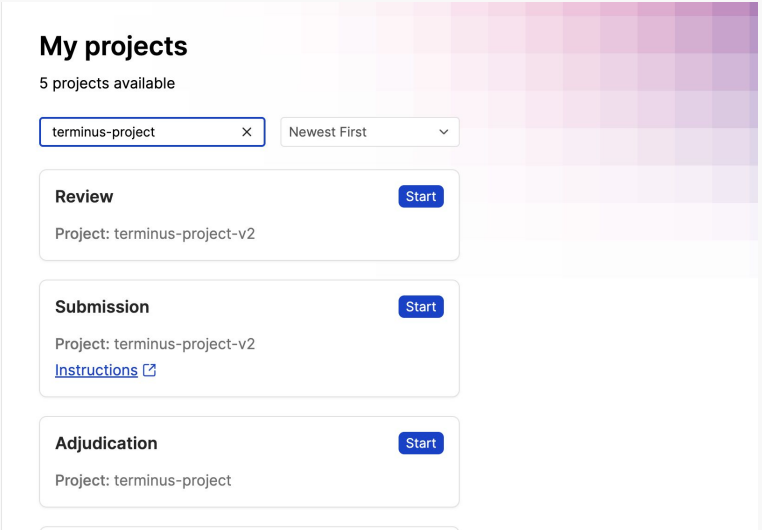


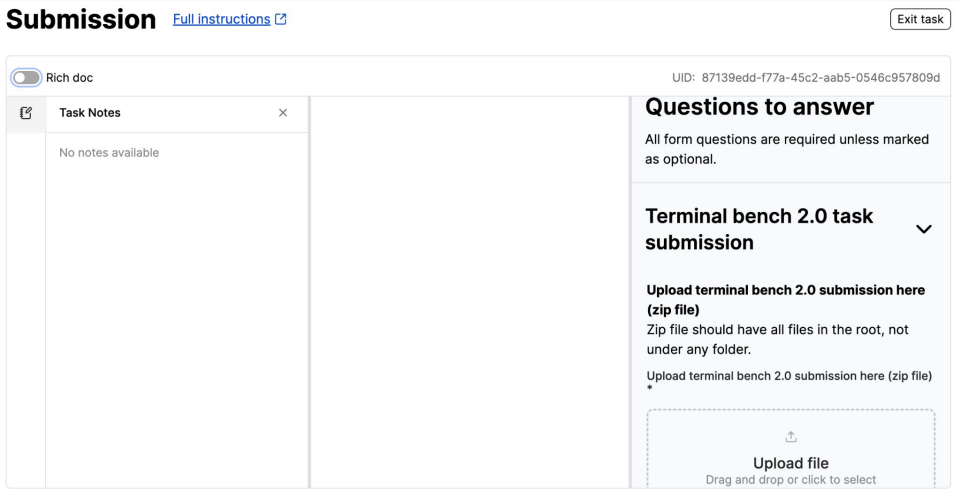

```
my-task/
├── instruction.md      # Task instructions (markdown)
├── task.toml           # Task configuration and metadata
├── environment/        # Environment definition folder
│   ├── Dockerfile      # OR docker-compose.yaml
│   └── [build files]   # Additional environment files
├── solution/           # Oracle solution (optional)
│   └── solve.sh        # Solution script + dependencies
├── tests/              # Test verification
│   ├── test.sh         # Test execution script
│   └── [test files]    # Test dependencies
└── [optional files]    # Data, configs, etc.
```

- Rename your task folder as desired, then implement your task locally (unchanged from GitHub repo flow). Your file structure will look something like this.

Platform Workflow - Pt. 2

- Go to your homepage on the Snorkel Expert Platform and click “Start” on the submission node for `terminus-project-v2` (not `terminus-project`)

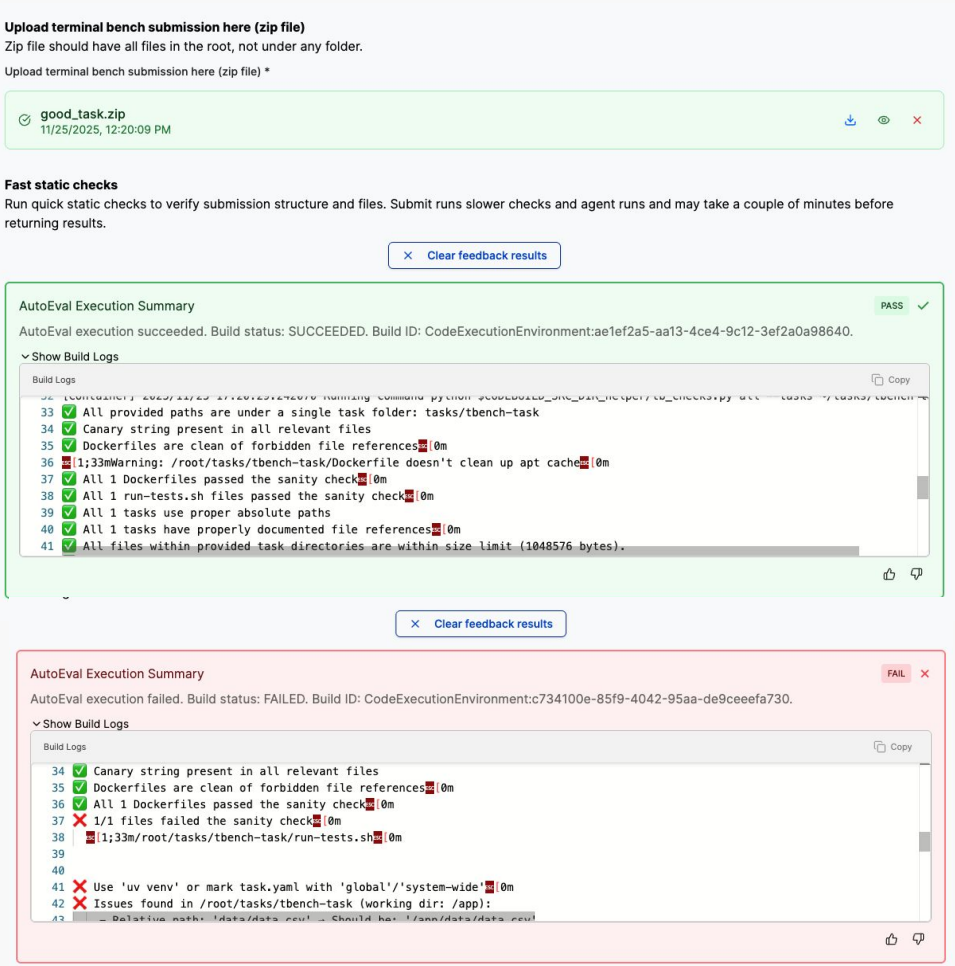




- This will load this Submission interface where you can drag and drop or browse to upload your task file

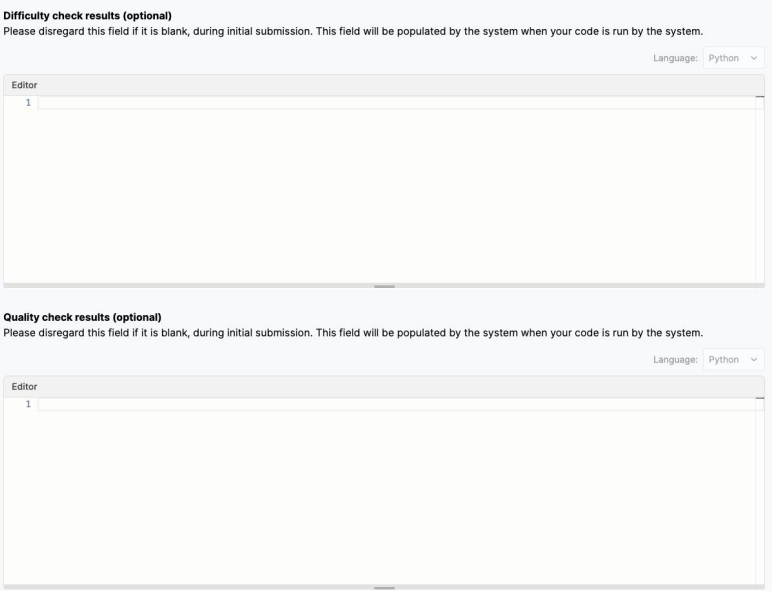
Platform Workflow - Pt. 3

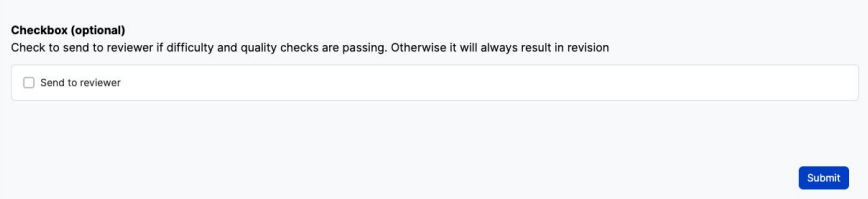
- After uploading the file and clicking the “Check Feedback” button, the fast CI checks will run on your file and return a summary of passes and failures
  - Note that this takes a minute or two to run
- If the checks all pass (green), you can continue to the next step
- If some of your checks fail (red), y**ou need to iterate on your task locally and then re-upload and run the checks again**



Platform Workflow - Pt. 4

- Nothing will appear in these fields on the right at this stage - these will be used later

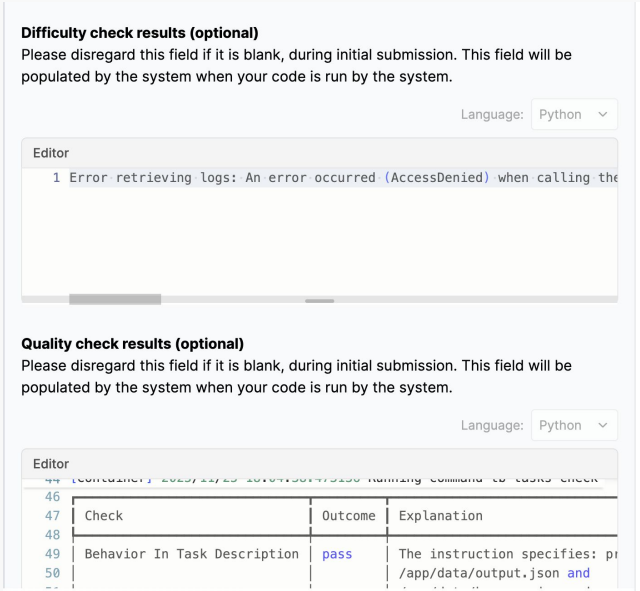


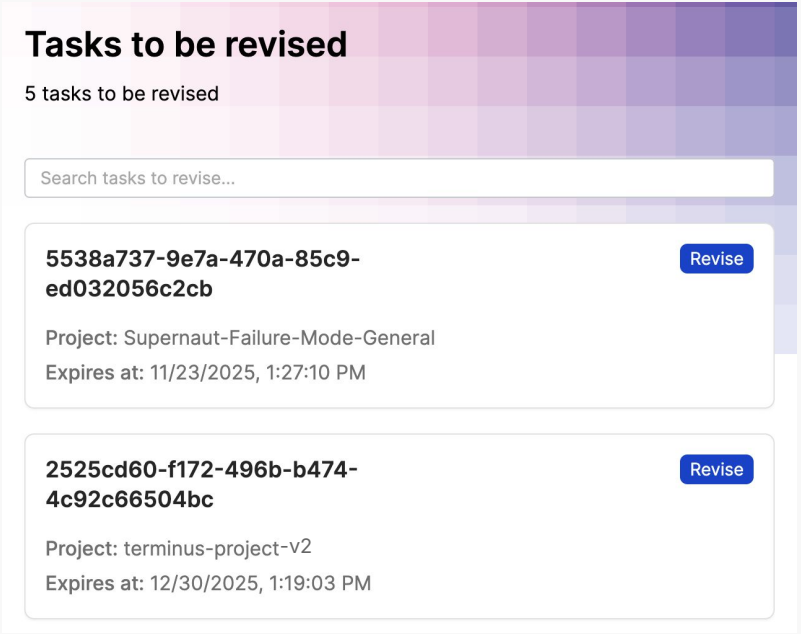


- Once all Fast Static Checks pass, click **Submit**.
- Do NOT check the ‘Send to Reviewer’ box at this point in time

Platform Workflow - Pt. 5

- After submitting, after 20-30 minutes, the task will appear on the right side under “Tasks to be revised” - click Revise here

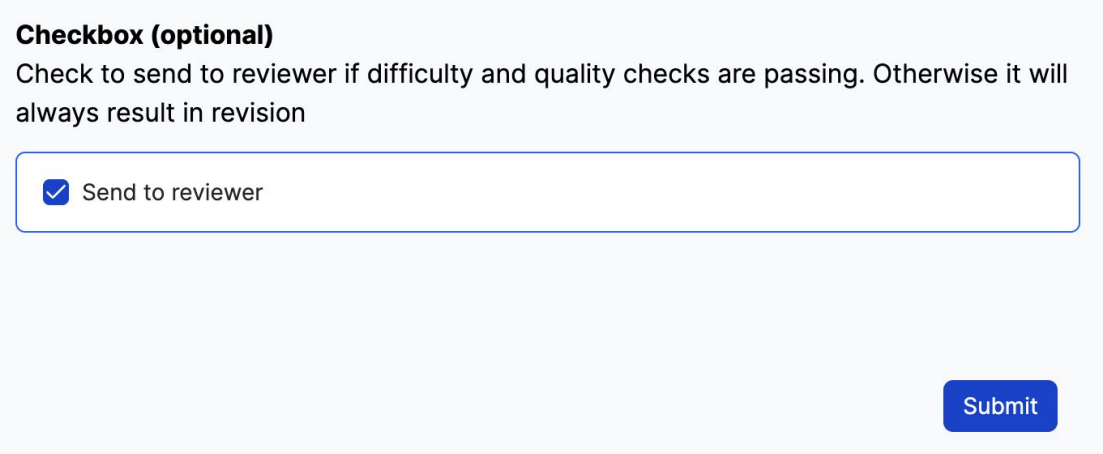




- Now these two fields will be populated with the results of the agent runs and the LLM quality checks
- If these don’t pass, **you should iterate on your task and submit again**

Platform Workflow - Pt. 6

- **Once all checks pass**, send the task to human review by checking the “Send to Reviewer” box and then clicking Submit
- After a human reviews, they will either accept or send back for revision - if they send back, it will again appear on the right side and you will need to iterate on your task and re-submit.



Completing a Task - Part 1

After you have followed the steps to create your task folder and skeleton, do the following to complete your task:

1. Edit the created Dockerfile using a text editor to set up your task environment
    a. Add any dependencies of the task, such as additional required packages
    b. If you require a multi-container environment or other custom configuration, see this page for more information on how to customize your docker-compose.yaml
    c. Docker Troubleshooting
        i. Ensure you have a recent installation of Docker Desktop.
        ii. On MacOS, enable the option in Advanced Settings: “Allow the default Docker socket to be used (requires password).”
        iii. Try the following:
            1. sudo dscl . create /Groups/docker
            2. sudo dseditgroup -o edit -a $USER -t user docker
2. Enter your task container in interactive mode: `harbor tasks start-env --path <task-folder> --interactive`

Completing a Task - Part 2

Continued from previous slide:

3. While interacting with your task container, test your solution idea to make sure that it works as expected
   a. Once solution is verified, record it and exit the container

4. Modify the solution file (solution/solve.sh) with the verified commands from the previous step
   a. This file will be used by the OracleAgent to ensure the task is solvable

5. Update the tests/test_outputs.py file to verify task completion
   a. Create pytest unit tests to ensure that the task was completed correctly
   b. If tests require any file dependencies, place them in the tests/ directory

6. Test your task solution passes and meets all the requirements specified in the tests: `harbor run --agent oracle --path <task-folder>`
   a. Note that you will need to install harbor from pypi in order to use these commands (`pip install harbor`)

Completing a Task - Part 3

Continued from previous slide:

7. Test your task solution with real agent
   a. Receive API key from Snorkel via email
   b. Update environment variables
      i. export OPENAI_API_KEY=<Portkey API key>
      ii. export OPENAI_BASE_URL=https://api.portkey.ai/v1
   c. Two models are available currently - GPT-5 and Claude Sonnet 4.5:
      i.
```
uv run harbor run \
-a terminus-2 \
-m openai/@openai-tbench/gpt-5 \
-p <task-folder>
```
      ii.
```
uv run harbor run \
-a terminus-2 \
-m openai/@anthropic-tbench/claude-sonnet-4-5-20250929 \
-p <task-folder>
```

8. Run CI/LLMaJ locally on your task
   a. `harbor run -a terminus-2 -m openai/@openai-tbench/gpt-5 -p <task-folder>`

Completing a Task - Part 4

Continued from previous slide:

9. Create a ZIP file of your task folder

10. Submit your task on the Snorkel Expert Platform in the **terminus-project-v2** project

Submission Quality Control

All submitted tasks are evaluated for quality and accuracy in three ways.

1. **LLM-as-Judge (LLMaJ)**
   a. Programmatic evaluators that run upon task submission
   b. Check for non-deterministic quality indicators (e.g. do the tests cover all intended behavior for the task?)
   c. Should iterate on submission until all of these pass

2. **Deterministic Continuous Integration (CI)**
   a. Programmatic checks that run upon task submission
   b. Check for deterministic adherence to structure and formatting (e.g. do all required files exist?)
   c. Should correct any issues that lead to CI failure

3. **Manual Peer Review**
   a. Once your task has passed both types of programmatic checks, it will be manually checked by an expert peer reviewer
   b. Peer reviewer will leave comments on the task indicating what needs to be added, removed, or corrected

LLMaJ - Part 1

- Behavior in Task Description
  - Checks whether all behaviors asserted by the tests are explicitly described in task.toml
  - Fails if the tests enforce implementation details that the task does not specify
- Behavior in Tests
  - Checks whether all behaviors described in the task.toml are actually tested (inverse of above)
  - Fails if one or more specified behaviors from the task is not fully tested
- Informative Test Docstrings
  - Checks for a clear and informative docstring describing each test
  - Fails if tests are missing docstrings or have poorly written ones
- Anti Cheating Measures
  - Checks that tests are executed after agent runs and there is no path to cheat by reading tests
  - Fails if there are ways for an agent to pass the tests by cheating instead of actually implementing functionality
- Structured Data Schema
  - If applicable, checks that any data files implied by the tests are specified in the task.toml or a separate schema file
  - Fails if any tests imply a data file that is not specified in the task

LLMaJ - Part 2

- Pinned Dependencies
  - Checks if required dependencies are pinned in the Dockerfile for reproducibility
  - Fails if any required dependencies are missing a pinned version
- Typos
  - Checks for typos in filenames, variables, and instructions
  - Fails if any typos are found
- Tests or Solution in Image
  - Checks to make sure that the Dockerfile does not copy tests or a solution into the image
  - Fails if Dockerfile does copy tests or solution into image
- Test Deps in Image
  - Checks to make sure that test dependencies are installed at test time (e.g., in tests/test.sh)
  - Fails if dependencies are installed into the image/during build
- Hardcoded Solution
  - Checks that the solution script writes a full source file implementing required behavior, not a hard-coded answer
  - Fails if the solution script merely echoes a final answer instead of implementing logic
- File Reference Mentioned
  - Checks that the required output file names given by the tests are specified in the task.toml
  - Fails if the task.toml does not mention the required output names that the tests check for

CI Evals - Part 1

- **Check Test File References**
  - **Operates on:** *test_outputs.py, task.toml*
  - **Checks:** all file references in test_outputs.yaml are mentioned in the task.toml - files that appear in both test and solution files but not in task.toml are flagged
  - **If fails:** either remove the files or add them to the task instructions
- **Validate Task Fields**
  - **Operates on:** *task.toml*
  - **Checks:** all required fields are present in the task.toml
  - **If fails:** add the missing required field
- **Check Canary String**
  - **Operates on:** *task.toml, Dockerfile, solution/solve.sh, test_outputs.py*
  - **Checks:** the canary string is present at the top of all required files
  - **If fails:** add the canary string to the top of all required files

CI Evals - Part 2

- **Check for Privileged Containers**
  - **Operates on:** *docker-compose.yaml*
  - **Checks:** for privileged containers in the docker-compose.yaml
  - **If fails:** remove any privileged container in the docker-compose.yaml
- **Check PR Files**
  - **Operates on:** *tasks/*
  - **Checks:** for any files not in the `tasks/` directory
  - **If fails:** delete or move all files not in the `tasks/` directory
- **Check task file sizes**
  - **Operates on:** all files
  - **Checks:** that every file is under 1MB
  - **If fails:** any files over 1MB should be removed or condensed

CI Evals - Part 3

- **Ruff**
  - **Operates on:** all files
  - **Checks:** for Linting errors
  - **If fails:** fix any Linting errors
- **Check tests/test.sh Sanity**
  - **Operates on:** *tests/test.sh, task.toml*
  - **Checks:** if the tests/test.sh file uses uv init or has a task.toml file with global or system-wide keywords
  - **If fails:** update tests/test.sh to contain a uv init or uv venv, or add global or system-wide keywords to the task.toml

CI Evals - Part 4

- **Check Task Absolute Paths**
  - **Operates on:** *task.toml*
  - **Checks:** that task instructions use absolute paths rather than relative paths
  - **If fails:** update instructions to use absolute paths
- **Check Dockerfile References**
  - **Operates on:** *Dockerfile*
  - **Checks:** if any of solution/solve.sh, tests/test.sh, or test_outputs.py are in the Dockerfile
  - **If fails:** remove the forbidden file(s) from the Dockerfile

Resources

- Training Site: https://snorkel-ai.github.io/Terminus-EC-Training/
- Project Guidelines: https://docs.google.com/document/d/1FjfHrOqHF-Uv16XEvS29gUlwu9jDxRmli36jsyzYLzk/edit?tab=t.0
- Slack Channels:
  - `#ec-terminus-submission`
  - `#ec-terminus-announcements`