Polaris Guidelines - Creating Training Data for Coding AI Agents

Table of Contents

Project Overview

In this project, your goal is to create training data for coding AI Agents.

What Training Data Consists Of:

- **An issue description** based on a pull request (PR) from GitHub
- **A Docker container** with the base commit, before the issue is fixed
- **A "fix patch"** based on the PR that fixes the issue
- **A "test patch"** with a new set of tests written by you to check the issue is fixed

How It Works:

The AI agent learns by interacting with the Docker container and trying to solve the issue. It gets a reward if it correctly solves the issue, which is judged by the new tests written by you.

Key Point:

The most important part of the project are the tests. It's absolutely crucial that the tests accurately measure when the agent has correctly solved the issue and when not. This means that the issue description needs to mention anything that's tested in tests in a clear and unambiguous way.

Time Expectations:

- **Expected annotation time:** 8-12 hours
- **Maximum time:** 15 hours

Your Task

You will base your annotations on public pull requests (PRs) from GitHub.

Workflow Steps:

1. **Choose a PR** from the list of approved PRs
2. **Describe all the changes** in the PR to ensure you understand them
3. **Write additional tests** for the PR, ensuring it covers all of PR's functionality
4. **Create a Docker container** with the pre-PR code
5. **Write an issue description** that accurately describes what is tested in your unit tests
6. **Run the agent** to verify the issue is sufficiently difficult
7. **Annotate agent runs** to ensure all test passes are correct solutions, and all test fails are incorrect solutions
8. **Map the issue description to tests**

> **Note:** These guidelines walk through one task end-to-end using **PR 3822: "Add PEP701 support"** from the Black repository as an example.

Step 1: Choosing a PR

The first step is to choose an appropriate PR from the annotation interface in the platform.

Example Repository: Black

Black is a common Python library to reformat Python code to follow a set of style guidelines. If you use Python, you most likely came across it, as it's pretty widely used.

Example PR: PR 3822: "Add PEP701 support"

Step 2: Fully Understanding the PR

Once you've picked the PR, you need to make sure you fully understand it. This means you'll need to dig deep into the codebase to understand what the fix involves and how to reproduce the issue.

Setting Up the Repository

1. Create a new folder on your machine and check out the repository at the base commit:

```bash
git clone https://github.com/psf/black.git
cd black
git checkout 944b99aa91f0a5afadf48b3a90cacdb5c8e9f858
```

2. Open the code in an editor of your choice.

Configure the Environment

We recommend using `uv` to manage your dependencies:

```bash
uv venv --python 3.12
source .venv/bin/activate
uv pip install -e .
uv sync --all-extras
uv pip install -r test_requirements.txt
```

> **Note:** If you have conda installed, deactivate it first with `conda deactivate` to avoid environment clashes.

Understanding the PR

- Look at the issue description and code in the original PR
- Try to understand what is going on
- **You can use Claude or Claude Code** to help you understand the code (but not other LLMs)

Writing the Fix Patch Description

Open the "Fix patch" section - this will contain the code from the PR. You will be asked to write an explanation of what the code does. **Only explain what the main code does** - we'll deal with the tests in the next step.

Example explanation structure:

- Describe the overall implementation approach
- Detail changes at each layer (tokenization, grammar, feature detection, line generation)
- Explain how different components work together

Step 3: Writing New Tests

⏱️ **Time Expectation:** 1-4 hours (this is the most time-consuming and complex step)

All PRs that you'll have access to either don't have tests, or have tests whose coverage is incomplete. Your job is to write comprehensive tests.

Organize Your Work

Create a git branch to manage your new code:

```bash
git checkout -b feature/tests-3822
```

Reproducing the Issue

Before writing tests, understand what the issue is and how to reproduce it.

Example for PEP 701 PR:

```bash
echo 'x=10; print(f"{f"{x}"}")' > test.py
python test.py  # should work and print out 10
black test.py   # should throw an error
cat test.py     # file is unchanged
```

Testing the Fix Patch

1. Download the fix patch as "fix.patch"
2. Apply the patch:

```bash
git apply fix.patch
```

3. Test that it's fixed:

```bash
echo 'x=10; print(f"{f"{x}"}")' > test.py
python test.py  # should work and print out 10
black test.py   # should work now
cat test.py     # file is reformatted
```

Starting on the Test Suite

Important: When constructing your test suite, make sure it **does NOT clash with any tests that the AI agent might write**.

Solution: Make a copy of existing PR tests and add **"oracle"** into their name.

```bash
cp tests/data/cases/pep_701.py tests/data/cases/pep_701_oracle.py
```

Commit to your branch:

```bash
git apply -R fix.patch  # revert the fix first
git add tests/data/cases/pep_701_oracle.py
git commit -m "Migrate tests from the PR"
```

Running the Unit Tests

Run tests with pytest:

```bash
pytest tests/test_format.py -k pep_701_oracle
```

Verify tests fail without the fix, and pass with the fix:

```bash
git apply fix.patch
pytest tests/test_format.py -k pep_701_oracle
git apply -R fix.patch
```

Adding New Tests with Claude Code

Highly recommended: Use Claude Code for writing tests - you might struggle otherwise.

Installation:

- Install Claude Code
- Start with Pro account ($20/month), but you'll likely need Max account for the project

Using Claude Code:

1. Run Claude Code with `claude` command
2. First prompt:

```
I am working on tests for a PR that adds PEP701 support to black. The PR is in the "fix.patch" file. Can you please have a look at this PR and examine if the test suite is comprehensive or not. If it is not comprehensive, please let me know what is missing.
```

3. Then instruct Claude Code to add tests:

```
Great, now please create a list of all the suggested test cases to add into pep_701_oracle.py.

Then, for each of your suggestions:
1. First confirm that it is not adequately covered in pep_701_oracle.py
2. Test that your suggestion is a valid python expression like this:

echo 'my_dict={"key": 11}; print(f"value: {my_dict["key"]}" )' > test.py
python test.py  # should work and print out "value: 11"
black test.py # should fail

3. If both 1 and 2 are true, add the suggested test case into pep_701_oracle.py

Repeat steps 1-3 for all of your suggestions in the list. When adding to pep_701_oracle.py, you don't need to add the data, just the f-string.
```

Tips for Using Claude Code Effectively:

- **Don't over-rely on Claude Code** - it might create a mess
- **Try it yourself first** before asking Claude Code
- If writing completely new test files, find similar test files in the repository and give them to Claude Code as examples
- **Always provide as much context as possible** and decompose each task into clearly defined subtasks

Verifying the Tests

After running Claude Code, verify the tests:

```
bash

git apply fix.patch
pytest tests/test_format.py -k pep_701_oracle
git apply -R fix.patch
```

Remove any test cases that fail even after the fix is applied (these indicate bugs in the PR itself).

Add verified tests to your branch:

```bash
git add tests/data/cases/pep_701_oracle.py
git commit -m "Add additional tests"
```

Final Remarks for Writing Tests

Create a final commit with your test patch:

```bash
git diff main > test.patch
```

Step 4: Creating the Docker Container

Uploading the Test Patch

Upload your `test.patch` file in the "Test patch" section of the interface.

Removing Tests from Fix Patch

Critical: Ensure the Fix Patch does NOT contain any tests. The platform will validate this automatically.

If the Fix Patch contains tests:

1. Download the fix patch
2. Manually edit it to remove all test-related changes
3. Re-upload the modified fix patch

Running Docker Validation

The platform will:

1. Create a Docker container with the base commit
2. Apply your test patch
3. Verify tests fail without the fix
4. Verify tests pass with the fix

Wait for validation to complete (may take several minutes).

Test Suite Completeness Validation

The system validates that:

- Tests fail on the base commit (reproducing the issue)
- Tests pass after applying the fix patch
- Test execution is deterministic and reliable

Step 5: Writing an Issue Description

The issue description is what the AI agent will receive when trying to solve the problem.

Writing the Base Issue Description

Goal: Write a clear description that:

- Explains what needs to be fixed
- Provides enough context
- Doesn't give away the solution
- Matches exactly what your tests validate

Structure:

1. Brief overview of the problem
2. Current behavior (what's broken)
3. Expected behavior (what should happen)
4. Specific requirements or constraints

Writing the Hints

Hints should:

- Guide the agent toward the solution without giving it away
- Reference relevant files or functions

- Mention key concepts or APIs
- Be progressive (start general, get more specific)

Reproducing the Issue

Include instructions for reproducing the issue:

- Minimal code example
- Expected error or wrong behavior
- Steps to verify the issue

Reproducing the Fix

Include verification steps:

- How to confirm the fix works
- What the correct behavior looks like

Step 6: Run the AI Agent

Click "Run Trial" to start the AI agent attempting to solve the issue.

The agent will:

1. Read your issue description
2. Explore the codebase
3. Attempt to implement a fix
4. Run the tests

Expectations:

- The agent should be able to solve the issue
- It should take multiple attempts (issue should be challenging but solvable)
- Typical solve rate: 40-70% on final attempts

Step 7: Review the AI Agent Attempts

Adjusting the Issue Description

Review the agent's attempts. If needed, adjust the issue description to:

- Add clarity where the agent got confused
- Remove misleading information
- Add hints if the agent is stuck
- Ensure description matches test requirements

Reviewing the Final AI Agent Attempts

For each attempt, verify:

- ✅ **Pass + Correct Solution** = Agent solved it correctly
- ❌ **Pass + Wrong Solution** = Tests are insufficient (revise tests)
- ✅ **Fail + Wrong Solution** = Tests correctly rejected bad solution
- ❌ **Fail + Correct Solution** = Tests are too strict (revise tests)

Final Notes on Reviewing AI Agent Attempts

- Review at least 5-10 attempts
- Ensure test outcomes align with solution quality
- Document any edge cases discovered

Step 8: Mapping Tests to the Issue Description

Map each test or test group to specific parts of the issue description:

1. Identify distinct requirements in the issue description
2. List which tests validate each requirement
3. Ensure every requirement has corresponding tests
4. Ensure every test corresponds to a requirement mentioned in the issue description

Format:

Requirement 1: [Description from issue]
- Tests: test_oracle_case_1, test_oracle_case_2

Requirement 2: [Another description]
- Tests: test_oracle_case_3, test_oracle_case_4

Using the Trial and Final Runners

Trial Runner

Use for iterative development:

- Quick feedback on issue description
- Test if agent can understand the problem
- Cheaper/faster than final runs
- Limited number of attempts

Final Runner

Use when ready for final evaluation:

- More comprehensive evaluation
- Generates multiple attempts
- Used for final quality assessment
- Counts toward project completion

Interface Buttons

- **Run Trial:** Start trial evaluation
- **Run Final:** Start final evaluation
- **View Attempts:** See all agent attempts
- **Download Logs:** Get detailed execution logs

Attempt Status Codes

- **Running:** Attempt in progress
- **Pass:** Tests passed
- **Fail:** Tests failed
- **Error:** System error (not test failure)
- **Timeout:** Agent exceeded time limit

Troubleshooting Common Issues

1. Running for a long time with no update

- Wait up to 10 minutes
- Check Docker container logs
- Verify test patch was applied correctly

2. "Query failed" error

- System issue - report to support
- Try running again after a few minutes

3. Test Patch cannot be applied

- Check for conflicts in test.patch
- Ensure patch is based on correct base commit
- Verify patch format is correct

4. Agent Diff is empty

- Agent didn't make any changes
- Issue description may be unclear
- Agent may have gotten stuck

5. Anything else

- Check documentation

- Contact support with attempt ID
- Provide detailed error description

Tips

Writing the Right Level of Detail in the Issue Description

Balance: Not too vague, not too specific.

Example 1: Too Vague ❌

> Fix the f-string handling in Black.

Example 1: Better ✅

> Add support for PEP 701 f-strings in Black's tokenizer and parser.
> The current implementation treats f-strings as simple STRING tokens,
> but PEP 701 requires proper grammar-level support for nested f-strings
> and complex format specifiers.

Example 2: Too Specific ❌

> In tokenize.py line 452, change the regex pattern from r'f"[^"]*"'
> to r'f"(?:[^"\\]|\\.)*"' and add FSTRING_START token type...

Example 2: Better ✅

> Update the tokenizer to recognize f-strings as distinct token sequences
> with proper support for nested braces and format specifiers. The tokenizer
> should track f-string nesting levels and distinguish between literal content
> and expression content.

Examining Attempts with Visual Changes

For PRs that affect UI/visual output:

- Include screenshots in issue description

- Provide visual comparison (before/after)
- Test for specific visual elements

What to Do if the PR Already Contains Tests

1. Use existing tests as a starting point
2. Add "oracle" suffix to avoid conflicts
3. Expand test coverage for gaps
4. Ensure tests are comprehensive

Adding Files to the Docker Container

If you need additional files:

1. Include them in your test patch
2. Use relative paths
3. Ensure files are in correct locations
4. Document any special setup needed

Repository-Specific Guides

Working with Grafana Repository

Grafana Local Setup

- Install Node.js and Go
- Follow Grafana's development setup guide
- Build frontend and backend separately

Grafana Docker and Test Configuration

- Use Grafana's official Docker setup
- Configure test database
- Set up test environment variables

Grafana Issue Description

- Focus on UI/UX changes
- Include panel/dashboard context
- Mention Grafana-specific terminology

Working with Discourse Repository

Discourse Local Setup

- Install Ruby and PostgreSQL
- Follow Discourse's development guide
- Set up test database

Discourse Docker and Test Configuration

- Use Discourse's Docker development environment
- Configure Redis and PostgreSQL
- Set up test fixtures

Discourse Issue Description

- Reference Discourse plugins/features
- Mention relevant models or controllers
- Include user workflow context

Summary

Key Takeaways:

1. **Tests are the most important part** - they determine if the agent solved the issue correctly
2. **Issue descriptions must match tests** - everything tested must be described clearly
3. **Use Claude Code for writing tests** - it significantly improves efficiency
4. **Verify everything thoroughly** - check tests pass/fail correctly before finalizing

5. **Expected time: 8-12 hours** - take your time to do it right

Success Criteria:

- Comprehensive test coverage
- Clear, accurate issue description
- Agent can solve the issue (40-70% success rate)
- Tests correctly validate solutions
- All requirements mapped to tests

Changelog

- **Oct 25, 2025:** Added "Writing the right level of detail in the Issue Description" section
- **Oct 15, 2025:** Added "Using the Trial and Final runners" section
- **Oct 13, 2025:** Updated pytest command and expanded reviewing section
- **Oct 10, 2025:** Increased expected time and max time
- **Oct 10, 2025:** Expanded test mapping section
- **Oct 08, 2025:** Major update - tutorial changed to psf/black
- **Sep 11, 2025:** Added note about unique test file names
- **Sep 09, 2025:** Added Discourse and Grafana guidance
- **Sep 04, 2025:** Added Grafana repository section
- **Aug 21, 2025:** Added guidance for PRs with existing tests

---

Questions or issues? Contact Stellar AI support through the platform.