Polaris Guidelines - Creating Training Data for Coding AI Agents

- Polaris Guidelines - Creating Training Data for Coding AI Agents
  - Changelog
  - Project Overview
  - Your Task
  - Step 1: Choosing a PR
  - Step 2: Fully Understanding the PR
    - Writing the Fix Patch Description
  - Step 3: Writing New Tests
    - Reproducing the Issue
    - Testing the Fix Patch
    - Starting on the Test Suite
    - Running the Unit Tests
    - Adding New Tests with Claude Code
    - Verifying the Tests
    - Final Remarks for Writing Tests
  - Step 4: Creating the Docker Container
    - Uploading the Test Patch
    - Removing Tests from Fix Patch
    - Running Docker Validation
    - Test Suite Completeness Validation
  - Step 5: Writing an Issue Description
    - Writing the Base Issue Description
    - Writing the Hints
    - Reproducing the Issue
    - Reproducing the Fix

    - Step 6: Run the AI Agent
    - Step 7: Review the AI Agent Attempts
        - Adjusting the Issue Description
        - Reviewing the Final AI Agent Attempts
        - Final Notes on Reviewing AI Agent Attempts
    - Step 8: Mapping Tests to the Issue Description
    - Summary
- Using the Trial and Final runners
    - Trial Runner
    - Final Runner
    - Interface buttons
    - Attempt status codes
    - Troubleshooting common issues
        - 1. Running for a long time with no update
        - 2. "Query failed" error
        - 3. Test Patch cannot be applied
        - 4. Agent Diff is empty
        - 5. Anything else
- Tips
    - Writing the right level of detail in the Issue Description
        - Example 1
        - Example 2
    - Examining attempts with visual changes
    - What to do if the PR already contains tests
    - Adding files to the Docker container
- Working with Grafana Repository
    - Grafana Local Setup
    - Grafana Docker and Test Configuration
    - Grafana Issue Description
- Working with Discourse Repository
    - Discourse Local Setup
    - Discourse Docker and Test Configuration
    - Discourse Issue Description

Changelog

- Oct 25, 2025 - Added the Writing the right level of detail in the Issue Description section
- Oct 15, 2025 - Added the "Using the Trial and Final runners" section
- Oct 13, 2025 - update pytest command in "Test execution script" to explicitly source the desired venv, expanded on the "Step : Reviewing AI agent Attempts" section
- Oct 10, 2025 - Increase in expected time and max time.
- Oct 10, 2025 - Expanded the "Test mapping" section to match the tutorial.
- Oct 08, 2025 - Major project updated - tutorial change to psf/black and many other changes to requirements
- Sep 11, 2025 - Added a note about creating test files with unique names to prevent conflicts when AI agent modifies files during evaluation in Your Task
- Sep 09, 2025 - Added Working with Discourse Repository and Grafana Issue Description
- Sep 04, 2025 - Added Working with Grafana Repository
- Aug 21, 2025 - Added explanation on what to do if the PR already contains tests

Project Overview

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

The training data consists of the following:

- 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

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

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 mention anything that's tested in tests in a clear and unambiguous way.

The **expected annotation time is 8-12h** with a **max of 15h**.

Your Task

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

Your workflow will have the following steps:

1. Choose a PR from our 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 descriptions that accurately describe 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 solution, and all test fails are incorrect solutions
8. Map the issue description to tests

To make this a bit more clear, 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.

You'll see the annotation interface in the platform.

For these guidelines, we'll use the "Black" repository as our example.

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.

For these guidelines we are going to use **PR 3822: "Add PEP701 support"** as our example.

Step 2: Fully Understanding the PR

Once we've picked the PR, we need to make sure we fully understand it.

This means we'll need to dig deep into the codebase to understand what the fix involves and how to reproduce the issue.

The very first step is to set up the repository on your local machine.

Start by creating a new folder on your machine, and then checking out the repository at the base commit:

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

Open the code in an editor of your choice.

Next, let's configure the environment for development. We recommend using uv to manage your dependencies.

Once you have installed uv, install the project like this:

```
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 also have conda installed, make sure to first deactivate conda with "conda deactivate" otherwise you might have a clash between different environments.

You should be now all set with the project.

Next, look at the [issue description and code in the original PR]() and try to understand what is going on.

You **can use Claude or Claude Code** to help you understand the code (but not other LLMs).

Once you've fully understood the PR, you can continue to the next step.

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. We only need to explain what the main code does, we'll deal with the tests in the next step.

Here is an example explanation for this PR:

```
The implementation add proper grammar-level support for f-
strings rather than being handled as simple string tokens,
requiring changes across multiple layers of the codebase.

At the tokenization level (`tokenize.py`), the most significant
changes occur. The tokenizer now recognizes f-strings as
distinct token sequences rather than treating them as simple
STRING tokens. New token types are introduced: `FSTRING_START`
(the prefix and opening quote), `FSTRING_MIDDLE` (the literal
text between expressions), `FSTRING_END` (the closing quote),
and related tokens like `BANG` for format specifiers. The
tokenizer maintains state with variables like `fstring_level`,
`inside_fstring_braces`, and `inside_fstring_colon` to track
when it's parsing f-string expressions versus literal content.
Complex regex patterns detect f-string boundaries and handle
edge cases like nested braces, backslash-N sequences, and
format specifiers. The tokenizer uses stacks (`endprog_stack`,
`parenlev_stack`) to handle nested f-strings properly.
```

The grammar file (`Grammar.txt`) receives updates to formally define f-string structures. The `atom` rule now accepts sequences of `STRING | fstring`, and new production rules define f-string components: `fstring` consists of `FSTRING_START`, zero or more `fstring_middle` elements, and `FSTRING_END`. The `fstring_middle` can be either literal text (`FSTRING_MIDDLE`) or a `fstring_replacement_field`, which contains the expression syntax including optional debug equals signs, format specifiers with `!`, and format specs that can recursively contain more f-string replacement fields. This recursive grammar structure properly captures PEP 701's support for nested f-strings and complex format specifications.

Feature detection (`__init__.py`) shifts from string parsing to AST-based detection. Previously, Black examined string token values to detect f-strings by looking for prefixes like `f"` and scanning for debug syntax with `=`. Now it traverses the AST looking for `token.FSTRING_START` nodes to detect f-strings and `token.RBRACE` nodes with `token.EQUAL` children to detect debug f-strings. This approach is cleaner and works correctly with the new grammar structure.

In the line generation phase (`linegen.py`), there's a temporary simplification: the new `visit_fstring` method converts f-string nodes back into simple string tokens using `_fstring_to_string`. This preserves Black's existing behavior while the infrastructure is being built—f-strings aren't reformatted yet, just passed through. There's commented-out code showing the intended future implementation that would properly format f-string components, normalize quotes in `fstring_middle` elements, and handle the new token structure throughout formatting. Supporting changes in `lines.py` ensure that empty `FSTRING_MIDDLE` tokens aren't truncated (important for preserving f-string structure), and `nodes.py` updates whitespace rules to handle the new tokens (no space before

`FSTRING_MIDDLE`, `FSTRING_END`, or `BANG`, and no space after `BANG` or when `LBRACE` starts an f-string replacement field).

Step 3: Writing New Tests

This step is the most time consuming and the most complex. We expect you to spend 1-4h on this step alone.

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

In this PR, there are already substantial tests, but they are incomplete. They don't test all the parts of the PR, and we want to have full coverage of the code in the PR.

Before we start, let's organize all of our new code into a git branch, so it's easier to manage:

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

You can name the branch however you want, and use any tools you are familiar with (e.g. SourceTree or directly from VSCode, etc).

Reproducing the Issue

Before we can write new tests, let's start by understanding what the issue is and how to reproduce it.

This PR adds support for PEP 701. This is a new feature of Python 3.12 allowing for more complex f-strings, such as `f"{f"{x}"}"` or `f"{'He said "hello\""}'`. Such expression would cause a syntax error in earlier Python version, and as such weren't supported by Black.

So, let's start by reproducing this issue locally. We are at the base commit, so we are expecting that such complex f-strings won't work.

Let's start by creating an example file on which we want to apply black formatting to:

```
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
```

So, we have code that's correct and runs fine in Python 3.12 (which we are using), but when we try to format it with black we get an error. This is the bug we want to fix.

Testing the Fix Patch

Let's confirm that the fix patch is working as expected on our simple reproduction example.

In the "Fix patch" section click on "Download" to download the fix patch as "fix.patch". Next, apply the patch:

```
git apply fix.patch
```

Next, let's try to use your reproduction to check it's fixed:

```
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 to be in two lines as per
python guidelines
```

Great, so we have now successfully reproduced the bug before the fix, and we confirmed that after the fix the bug is no longer present.

Even when the PR has unit tests already, you should always independently reproduce the bug on your own as well, to fully understand what the bug.

Starting on the Test Suite

Now that we can reproduce the issue, let's start working on the test suite.

The PR already has some tests in it. There are three files: pep_701.py, debug_visitor.out and test_black.py.

The bulk of the new tests is in pep_701.py. This test file contains a set of inputs and expected outputs and is run indirectly by "test_format.py" test file.

Some additional tests are also added into debug_visitor.out that's being run by test_black.py. This test checks how the black's AST (Abstract Syntax Tree) parser represents the internal structure of Python code.

The debug_visitor.out test is a very detailed test that's tied to the specific implementation in the PR. It assumes that each token is parsed and represented in a very specific way. For now we'll exclude this test from our test suite, and come back to it later.

The final change in test_black.py support the previous change in debug_visitor.out.

So, let's base our tests on pep_701.py which we'll augmented with further PEP 701 syntax that's currently missing.

When we construct our test suite, we want to make sure it **does NOT clash with any tests that the AI agent might write** as it tries to solve the issue.

To ensure this, **we'll make a copy of existing PR tests and add "oracle" into their name**.

So, let's do that:

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

This will be the basis of our test suite, so let's now commit this to our branch.

First, revert the fix patch changes with:

```
git apply -R fix.patch
```

Then, commit the three new files to our branch:

```
git add tests/data/cases/pep_701_oracle.py
git commit -m "Migrate tests from the PR"
```

Great, so now we have a clean branch just with our test suite.

Running the Unit Tests

Great, so now we have the test suite. But how do we run it?

We run the tests just with pytest:

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

Since we have reversed the Fix patch in the previous step, the tests fails as expected.

Next, let's check that when we apply the fix patch the tests are passing.

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

This should now show that both tests are passing after we apply the fix - so everything is working as expected. In the last line we once again revert the patch so we can continue with writing tests.

Adding New Tests with Claude Code

In this PR we already had some tests. In other PRs you might find that there are no tests, in which case you'll start here - by writing new tests.

We are going to add new tests with Claude Code. We **highly recommend using Claude Code otherwise you might struggle writing good tests**, especially when writing completely new tests.

First, you'll need to install Claude Code locally on your machine. You can get started on the Pro account ($20/month), but if you start working on the project you'll most likely need the Max account.

Once installed, run it with "claude" and then put in this 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.
```

Claude will analyse files and might also consult the official PEP-701 reference document. Once finished, Claude might give you an analysis identifying gaps in test coverage.

Now, let's instruct Claude Code to test its suggestions and add them to the test file. In the same Claude Code conversation as before, write this:

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

Then, for each of the 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.
```

Now Claude Code will proceed to test each of the suggestions and add them one-by-one into the test file.

It's very important that you don't over-rely on Claude Code, as it might create a total mess of test files.

We have a few tips on how to use it effectively:

- Before telling Claude Code to do something always try it yourself first. You cannot and shouldn't rely on it to just "figure it out". It won't. Instead, it'll most likely create a mess that'll take even more time to fix.
- If you need to write completely new test file (as opposed to adding to existing one), try to find a similar test file in the repository and give it to Claude Code as an example and template on what to do.
- Always provide as much context as possible and decompose each task into clearly defined subtasks.

If you follow these tips, you should be able to use Claude Code effectively.

Verifying the Tests

After running Claude Code, you should verify the tests. Let's confirm that all of these tests still pass after the fix is applied:

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

You may find some tests that fail even after the fix is applied. This often indicates bugs in the PR itself. This is OK - we want the AI Agent solution be *at*

least as good as the solution in the PR. So, remove any test cases that fail after the fix is applied.

After fixing any issues, add the tests to your branch:

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

Final Remarks for Writing Tests

The original PR might have additional implementation-specific tests (like debug_visitor.out in this example). These tests check that the actual tokens produced by the parser conform to a very strict schema. However, if the PR's implementation has issues, you might decide to omit such highly implementation-specific tests and focus on end-to-end behavior tests instead.

One last tip on finding gaps in tests: in some cases you might also find it useful to run the pytest coverage tool to find parts of code not covered by tests. You can do this by installing "pytest-cov" and then running pytest with the "--cov=src --cov-report=html" to get a nice browsable HTML to see what is covered and what not.

Step 4: Creating the Docker Container

The next step is to create a Docker container which the AI Agent will use to try to solve the issue.

This step involves:

1. Uploading the test patch
2. Removing tests from the fix patch (if there are any)
3. Reviewing the Dockerfile and test execution script - this you normally don't need to modify
4. Running the validation to ensure the tests fail before the fix patch is applied, and pass after.

Uploading the Test Patch

We first step of Docker setup is to add the tests that we've just written.

Execute this command in your repository to get the patch and then upload it in the "Test patch" section by clicking on "Import diff file":

```
git diff 944b99aa91f0a5afadf48b3a90cacdb5c8e9f858..HEAD --output=test.patch
```

Removing Tests from Fix Patch

Next, we'll need to make sure that the fix patch doesn't have any tests in it. We want the fix patch only to contain the solution, not the tests - as these might give away the desired functionality to the AI agent, making the problem unrealistic and too easy.

In the "Fix Patch" section click on "Edit diff". This will bring up the diff editor.

Next, delete all the lines in the diff that are related to the test files (in this case: pep_701.py, debug_visitor.out and test_black.py).

The **"Fix patch" should NOT have any tests in it!** All tests need to be in the "Test patch".

Click on "Save". Now you should have a clean "Fix patch" that only has the solution and none of the tests.

Running Docker Validation

The next step is to create a Docker container which the AI Agent will use to try to solve the issue.

In general, unless you are using a very old PR, you should be able to use the Dockerfile that we provided. Sometimes you might need to make some small modifications to make sure dependencies are correct, but in general this should not be necessary.

Next, let's look at the "Test execution script" section. This is the actual script we'll use inside Docker to run the tests and parse the results. You will normally only need to change the very last line of this script.

In our case, let's change the last line to be:

```
source .venv/bin/activate && pytest --junit-xml=/tmp/grading_results.xml tests/test_format.py
```

As you can see, we'll be running not only the pep_701_oracle.py test, but ALL the tests in the test_format.py test suite. This is because the AI agent will have to make some significant changes to the formatting logic and we want to make sure it doesn't accidentally break something.

So we'll use our pep_701_oracle.py test to verify PEP 701 functionality, and the remaining 172 tests as **regression tests**.

How many regression tests you want to include is up to you. If a PR is adding entirely new standalone functionality regression tests might not be necessary. If it's modifying a lot of common code you might want to do a bit of research which regression tests are good to include.

In general we normally do NOT want to run the full suite, because this is usually too slow. Instead, we want to run our new tests, plus any relevant regression tests.

Next, let's verify everything is working. Open the "Dockerfile Validation" section and click on "Pre-PR Validation" and then "Validate".

This will start validation before the fix patch is applied. It will build the Docker container, check out the base commit and then run the tests we've written. The expectation is that the the new test will fail. This will take a few minutes as we'll need to build the docker container and run the tests.

Once done, click on "Post-PR Validation" and again click on "Validate". This will now build the container with the base commit, but with the fix patch applied. So, the expectation is now that the all tests will succeed.

All tests should pass with no problems now.

Test Suite Completeness Validation

Great! Now that we have both the Fix Patch and Test Patch, we need to ensure that the tests you provided completely cover all changes from the Fix Patch.

To do this, we use **Test Suite Completeness Validation**. This process verifies that each chunk of code changes in your Fix Patch is covered by at least one test. It does this by removing various parts of the Fix Patch and seeing if the tests still pass.

If a part of the Fix Patch can be removed, but the tests still pass, this strongly suggests that this code is not actually tested. This could be because our tests are incomplete, or because the code is not actually being used.

So, let's go ahead and do this. Click on the "Test Suite Completeness Validation" section and then "Auto-Partition". This will use an LLM to automatically decompose the PR into smaller logical units that can be removed independently.

After the auto-partitioning is done, you should get 6-10 partitions covering various parts of the solution. You might get different partitions than those mentioned in this example, and it's sometimes worth re-running this command multiple times to get good partitions.

In most cases you won't need to edit the partitions, but if you do, there are buttons to edit the individual patches for each partition.

Click on the "Test Partitions" tab and click on "Validate".

Once the validation is finished, click on the "Test Results" tab to review the tests.

There will be a few partitions where the tests still pass even after code is removed. This is often dead code that's included in the PR but never actually used. In this specific instance, it's OK to leave it as part of the Fix Patch as it's untestable dead code.

However, sometimes PRs silently fix other testable issues as well. In this situation you would want to edit the Fix Patch so that the solution is only for this issue and not with other stuff mixed in.

For this example we'll need to override the validation. Click on "I acknowledge the coverage gaps and choose to override based on my analysis" checkbox on the "Test Results" tab. Then write an explanation of why you're overriding.

Step 5: Writing an Issue Description

The next step is to write an issue description that the AI agent will use to try to solve the problem.

The issue description should:

1. **Mention everything that's tested**. The most important thing is to make sure the issue description is aligned with the tests. Everything that is being tested needs to be mentioned in one way or the other. The issue description alone should be sufficient to create a solution that passes the tests, with no guesswork involved from the agent.
2. **Clearly explain the expected behavior**. The issue shouldn't use unclear or cryptic language. It should clearly explain what the issue is, in way that can clearly pass the tests.
3. **Be written in a natural style**. The issue description should closely match the style of explanation used in the PR or any associated issue. Do NOT make it sound LLM-generated. Make it sounds natural.

When training the AI agent, tests will be used to judge if the solution is correct, so it's vital that this signal is correct. In other words, there should be only one possible solution and it should exactly match what's being tested in tests.

Writing the Base Issue Description

Let's start with a minimalistic issue description:

```
# Support PEP 701: Syntactic formalization of f-strings in
Python 3.12
```

Python 3.12 is introducing [PEP 701](https://peps.python.org/pep-0701/) which fundamentally changes how f-strings work by formalizing their syntax. This enables several new f-string variations that our current parser can't handle at all.

The Problem

Right now, Black fails to parse valid Python 3.12 f-string syntax. For example:

\`\`\`python
>>> f"{f"{"f"}"}" # nested f-strings with reused quotes
'f'

>>> f"hello {f"world"} test" # mixing quote styles
'hello world test'

>>> f' \\' {f"'"} \\' ' # complex quote mixing
" ' ' ' "
\`\`\`

All of these are valid in Python 3.12 but Black chokes on them. For each of parsing simply fails and black is not able to reformat the file.

Expected behavior

Black support the full range of PEP 701 syntax and can successfully reformat a file with PEP 701 syntax.

You can run the code in /workdir/black with \`uv\`, there is a \`/workdir/black/.venv\` configured already (you can do \`source .venv/bin/activate\` to activate the env). You can run the tests with pytest.

We recommend starting with a minimalistic issue like this. Only include the information that's necessary. In our case, we are including some clear examples of strings that don't work, and we include a simple statement for expected behavior that we should support all of PEP 701.

We also gave some information on where the code is and how to use it. This is important because we don't want the AI Agent to fail just because it doesn't know how to run the code or the test. You should always include this type of information into the issue description.

Writing the Hints

The base issue description should contain only the minimal information to make the problem fully specified. The idea is that this will make for a difficult problem for the AI agent.

When the agent cannot solve the issue at all, we require you to also write hints for it to be able to solve it at least once. This is important to confirm that the issue is actually solvable based on your description and some hints that are inferrable from the code.

The hint SHOULD NOT resolve an ambiguity. If you are finding yourself writing a hint like "by X we mean Y", this indicates that the base description is ambiguous and this might be why it's difficult. The difficulty of the base description shouldn't rely on it being ambiguous, but on it requiring the AI Agent to examine the codebase and figure out what the problem is.

It's usually quite difficult to know which hints to include before you run the AI agent for the first time. So we'll start with something simple and generic and then add more specific hints once we see what mistakes the AI Agent makes.

Write this in the "Key hints" field:

```
- Comprehensively test your solution to ensure all edge cases
of PEP701 syntax are being tested and all work end-to-end
- Make sure no tests are broken by running the test suite for
test_format and making sure there are no regressions
```

You'll also need to explain why this is a good hint, so in the explanation put:

Starting with some generic hints to reinforce the requirement to full PEP701 support and no regressions, as those might be the most likely mistakes.

Reproducing the Issue

Great, so now we have an issue description and a test suite that's complete. We now need to document our procedure for reproducing the issue.

In most PRs you will be editing unit tests, which means you won't be able to automatically test the behavior end-to-end. This is why it's useful to have a way to manually test the behavior end-to-end to be absolutely certain that a solution either correct or incorrect.

If you remember from the beginning of these guidelines, we've used this simple script to reproduce the issue locally:

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

This is an end-to-end test because it tests the behavior as it would appear to the user, so it's the most reliable way to test a solution.

So let's record this.

Click on the "Reproduce issue" section and then "Start Playground". Wait for the VM to start and then click on "Build and run container" (while the default "pre-pr" option is selected).

This will build and run the docker container at the base commit. Once the container starts, you will see a bash console. This console will be running inside the container.

Now type in your reproduction steps and test that they work. Click on "Save screenshot" to take a screenshot of this view. Then write an explanation of what you did.

This shows that we have understood what the problem is and know how to reproduce it. This will be used by our QA to validate your submission.

Reproducing the Fix

Great, now, let's make sure we can manually verify that the issue has been fixed.

Click on "Post-PR" in the panel, and then on "Build and run container" again. This is now going to build the container with the fix patch applied and run it for us.

Now do the same reproduction steps and verify the fix works.

Save a screenshot of the result. Then write an explanation of what you did.

Step 6: Run the AI Agent

Now that we have a working Docker container, the issue description with hints, unit tests and an manual reproduction protocol, we are ready to run the AI Agent.

The purpose of running the AI Agent is two-fold:

- We want to verify that the issue description is not ambiguous, i.e. that there aren't other valid ways to solve the issues that are not tested by our tests
- We want to make sure the issue is sufficiently hard.

In terms of difficulty. You should aim for **the AI agent to have a 0-50% success rate on the base issue**.

If the agent has a 0% success rate, that's great - this means the PR is hard and that's what we want (this should be about 30% of all annotations). When the success rate is 0%, we require the hinted version to have success rate >0%. In other words, we want to verify that it's possible to solve it with your issue description and hints.

If the agent has a 10%-50% success rate, that's also good (this should be 70% of annotations). In this situation, we can just ignore the hinted version as we already know the issue is solvable.

You will notice that there are two agent runners: "Trial runner" and "Final runner". The "Trial runner" is running on our infra and is running a somewhat weaker model. We recommend using it to tune your issue description and do all the experimentation. So, if the issue description seems to easy for the "Trail runner" it will almost certainly also be too easy for the "Final runner".

You should only use the "Final runner" once you've done your iterations on the "Trial runner" and got to a good place with it (e.g. base description strictly <50% but ideally 0% because the Final runner might still solve it).

Since the principles of running the Trial and Final runner are exactly the same, we'll use the "Final runner" in this example.

So, in the "Final runner" section click on on "Start new agent run" to start the agent now.

As you'll see, we'll run agent on both versions of the issue (base, base + key hints) each 10 times.

Normally this will take between 20-30 minutes for the "Trial runner" and 1-6h for the "Final runner". If you are running the final runner, you can normally just stop your timer, leave the window open and come back it to later in the day or the next day.

For demonstration purposes in the interactive tutorial, the results are cached, but in actual annotations you'll need to wait for the runs to complete.

Step 7: Review the AI Agent Attempts

Once the Agent attempts have finished, you'll see that there are 2 sets of attempts. Let's review some attempts to understand where the AI agent is getting stuck.

Click on "Batch 1 (base)" to see how the AI agent solved this issue description. Look at the "attempt_1" in this batch. You will see multiple tabs:

- Git diff - the diff of the solution produced by the AI Agent
- Test results - the results of running the tests in structure format
- Test output - the full output of running tests (useful to debug test running issues)
- Transcript - the full step-by-step transcript of the AI Agent solving the issue
- Your explanation - your explanation of why the tests passed/failed plus a container explorer
- AI review - automatic AI review of the attempt, comparing the AI solution to the gold patch

Let's start with looking at "AI review".

As we can see, the AI Agent took a much simpler approach. Instead of creating a state machine with a grammar, it just created a new function to parse the f-strings.

It tested some simple PEP 701 which succeeded, but it fails on some more complex solutions.

Next, let's look at the "Test results". As we can see, 157 tests passed and 15 failed (including the one we introduced). So, the agent both failed to solve the issue and introduced new bugs. Pretty bad performance.

Next, look at the "Transcript" and click on "Load Transcript". Here you can see the exact steps the AI agent did. Once again, this confirms that it does solve it partially but not fully.

Next, look at attempt_2. This one is better, only the 1 test fail that we introduced, and actually some of the simpler f-string seems to work. So this is getting there, but still incorrect.

Finally, at the bottom of the list of attempts, look at the "AI Review Summary". This is the summary of all the reviews. As you'll see the pattern we see in attempt_1 is a common pattern across all attempts.

Next, look at Batch 2 (key hints) attempts. As you'll see, our hints didn't really help much, and the failures are quite similar.

Adjusting the Issue Description

Once the Agent attempts have finished, you'll see that there are 2 sets of attempts and the success for both based and hinted is 0%. While the base pass rate of 0% is OK, we need the hinted version to have at least 10% success rate, to ensure the issue is solvable.

So, let's review some attempts to understand where the AI agent is getting stuck.

After reviewing the attempts, we can see the AI Agent is struggling because it's not testing all the PEP 701 features comprehensively. So let's help out the agent a bit, by giving more examples and ways to reproduce the issue:

```
# Support PEP 701: Syntactic formalization of f-strings in
Python 3.12

Python 3.12 is introducing [PEP 701]
(https://peps.python.org/pep-0701/) which fundamentally changes
how f-strings work by formalizing their syntax. This enables
several new f-string variations that our current parser can't
handle at all.

## The Problem

Right now, Black fails to parse valid Python 3.12 f-string
syntax. For example:

\`\`\`python
>>> f"{f"{"f"}"}"  # nested f-strings with reused quotes
'f'

>>> f"hello {f"world"} test"  # mixing quote styles
'hello world test'
```

```
>>> f' \\' {f"'"} \\' '  # complex quote mixing
" ' ' ' "
```

All of these are valid in Python 3.12 but Black chokes on them. For each of parsing simply fails and black is not able to reformat the file.

Expected behavior

Black support the full range of PEP 701 syntax. For example, this should work:

```
echo 'x=10; print(f"{f"{x}"}")' > test.py
black test.py # should reformat into two lines instead of giving an error
```

All of these others should work using the same code block (replacing into "echo"):

Quote Reuse - Use same quote type inside and outside f-strings:

```python
f"result: {my_dict["key"]}"
f"nested: {data["x"]["y"]}"
```

Backslashes in Expressions - Now allowed in expression parts:

```python
f"joined: {'\\n'.join(lines)}"
f"match: {re.match(r'\\d+', text)}"
```

```
**Comments in Expressions** - Comments work inside f-string
expressions:
\`\`\`python
f"result: {
    x +  # comment about x
    y    # comment about y
}"
\`\`\`

**Nested F-Strings** - Deep nesting with quote reuse:
\`\`\`python
f"foo {{ {f"a  {2 + 2}  b"}bar {{ baz"
f'{f"{f'{x}'}"}'
\`\`\`

**Multi-line Expressions** - Expressions can span lines:
\`\`\`python
f"foo {
    2 + 2
} bar baz"
\`\`\`

**Empty F-Strings** - Valid syntax (normalized to \`f""\`):
\`\`\`python
f""
\`\`\`

You can run the code in /workdir/black with \`uv\`, there is a
\`/workdir/black/.venv\` configured already (you can do
\`source .venv/bin/activate\` to activate the env). You can run
the tests with pytest.
```

This now gives the AI agent more things to test and should enable it to make a more complete solution.

For the hints, you might update them to:

```
- Start with a simple implementation, e.g. the first example in
the "Expected behavior"
- Then go one-by-one example in "Expected behavior" and ensure
all of them work
- Once you are finished with that, generate further edge cases
for PEP701 and ensure all of them work as well
- Do not give up early, keep going until all of them are
correct.
- Make sure no tests are broken by running the test suite for
test_format and making sure there are no regressions
```

In the explanation for the hints:

```
The biggest problem seems to be the AI agent not going through
all the examples one-by-one to ensure they are all solved. So
gave some hints to help it plan its solution better. These
hints do not resolve any remaining ambiguity, they simply tell
the agent how to approach solving this issue.
```

After making these adjustments, run the agent again to see if the updated description helps.

Reviewing the Final AI Agent Attempts

Now the pass rates should look better. For our example, we might see something like 10% pass rate for base and 60% for the key hints.

These are now both good, but are they correct?

We need to look at all of the runs in Batch 1 in detail to ensure all of them are correct if tests pass, and incorrect if they fail.

Let's start with Batch 1, attempt_1. This might be the only attempt where the tests pass. But does it actually work?

We can verify this using our reproduction protocol.

Click on the "Your explanation" tab. Then click on "Start playground". This will open the same VM we used for the reproduction, but on a virtual desktop

that's dedicated to this attempt.

Then click on "Build and run container". This will apply both the Git diff from the AI agent for this attempt, and our test patch.

Once it's finished, you'll be inside the docker container. Now let's run our reproduction:

```
cd /workdir/black
source .venv/bin/activate
echo 'x=10; print(f"{f"{x}"}")' > test.py
black test.py # should work
cat test.py # file is reformatted
```

As we can see, this indeed works. That's great. So, not only the tests pass, but also our sanity check reproduction also works.

Click on "Save screenshot for attempt_1" to save a screenshot of this as proof. In the "Your explanation" write your explanation of this attempt. For example:

```
The agent takes a very different approach from the PR, but
produces the correct end behavior. The full range of PEP 701
strings is correctly supported and the code containing PEP 701
no longer fails.

The agent adds functions to detect f-strings and then outputs
them as strings. This solve the issue, but might not be the
most general solution for the future when we might want to
reformat strings. The PR takes a more general and future-proof
approach, but according to the issue description such future-
proof approach is not necessary, so the AI agent solution is
adequate.
```

Let's also look at the "AI review" for attempt_1. It might claim the solution is incorrect and that this is a "false positive". However, as we've just verified above this would be incorrect. This is a good reminder that you should not solely rely on the AI review, you should always manually verify as well. The AI

review is good at summarizing what the agent did and how it's different from the PR, but it might not be good in judging if the solution is actually correct (if it was, we wouldn't need to annotate anything!).

Next, let's look at attempt_2. Here all the tests pass, except the new one we added.

Let's verify it's actually incorrect. Click on "Your explanation" again, "Start playground" and "Build and run container".

Now, let's again run our reproduction code from above. This time, the reproduction fails. We get an error when running "black test.py", indicating the issue is not fixed.

Once again, click on "Save screenshot for attempt_2" to document this. Then write this explanation in the textbox above the VM:

```
The agent attempts a solution that's more similar to the PR,
with a custom grammar and parsing logic. However, the solution
does not work. Although no regressions were introduced, the PEP
701 f-string parsing does not work, as verified by my
independent reproduction (see screenshot).

The parsing logic the AI agent wrote does not correctly parse
all the possible PEP 701 f-strings, and fails even on some of
the examples explicitly given in the issue description. The
custom grammar solution seems incomplete, with missing
Grammar.txt changes.

All in all, a clear requirement given in the issue description
was not met, so the agent failure is not due to ambiguity in
the issue description, but is a genuine failure to implement a
clear requirement from the issue description.
```

You'll need to do this for all 10 attempts. Each time reproducing the issue in the VM, taking a screenshot and writing an explanation of your verdict.

Final Notes on Reviewing AI Agent Attempts

Reviewing AI agent attempts is a critical part of the annotation. You will spend a lot of time reviewing attempts, adjusting the issue description and hints based on how the agent does.

It is **critical that tests pass only when the solution is correct, and test fail only when the solution is incorrect**.

No abiding by this leads to **the most common mistake** we see: a low pass rate because something from tests is not mentioned in the issue description.

For example:

- **In frontend tests**. Let's say tests expect a button with a label "Submit now", but this label is not mentioned in the issue description. The agent might make a correct frontend but with a button that says "Submit" and thus fail the tests although the solution is correct.
- **In backend tests**. Let's say tests expect that a new parameter "parsing_logic" is added to function "parse", but this is not mentioned in the issue description explicitly. The agent makes a correct solution, but calls this parameter "logic" and the test fail, despite the solution being correct

It can be quite difficult to catch these missing details the first time. But, if you spend time manually investigating the AI agent solutions, you'll soon find them. As you do more tasks, you'll get better at including them from the beginning.

Reviewing AI agent attempts also helps with writing better hints. If you see the AI agent making the same mistake repeatedly, you can tell it what to do in the hints. For example, if it struggled to find the piece of code that's the cause of the bug, you can give it a hint where this code is.

Step 8: Mapping Tests to the Issue Description

This is the final step in the annotation.

After you've reviewed the agent runs and tweaked the issue description you should have 100% confidence that the issue description is unambiguous and

complete.

Now we need to document this further.

We've built a tool that allows you to highlight portions of the issue description and map them back to tests.

If an issue description is a list of feature or things to fix, you can map these requirements directly back to test. Try to make this as fine-grained as possible, e.g. if you have multiple tests you wrote, map them to exact position in the issue description where the corresponding requirements are mentioned.

For example, in our Black PR tutorial, we would select the whole "Expected behavior" section and map it to our comprehensive pep_701_oracle.py test suite with a detailed explanation like:

```
A comprehensive test suite of PEP 701 syntax, including:

## Core F-String Formatting

1. **Basic quote variations** - Single, double, and triple
quotes (both `'` and `"`)
2. **Escaped braces** - Double braces `{{` and `}}` for literal
braces
3. **Expression embedding** - Simple expressions like `{2 + 2}`
inside f-strings
4. **Whitespace preservation** - Ensuring `FSTRING_MIDDLE`
whitespace isn't stripped

## Multi-line F-Strings

5. **Line breaks in expressions** - Expressions split across
multiple lines
6. **Triple-quoted multi-line strings** - Complex formatting
with line breaks
7. **Triple quotes at index zero** - Edge case for ending
position
```

Nested F-Strings

8. **Single-level nesting** - F-strings within f-strings
9. **Multi-level nesting** - Up to 5 levels deep: `f"{f"{f"{f"{f"{x}"}"}"}"}"`
10. **Mixed quote nesting** - Using different quote types in nested f-strings

PEP 701 Specific Features

11. **Quote reuse** - Using same quote type inside expressions: `f"result: {my_dict["key"]}"`
12. **Backslashes in expressions** - `{'\n'.join(lines)}`, `{path.split('\\')}`
13. **Comments in expressions** - Comments within the `{}` expression part
14. **Empty f-strings** - `f""` and `f''`

Format Specifications

15. **Format specs** - Alignment, padding: `{2+2:=^72}`
16. **Nested format specs** - `{x:{y}d}`, `{2+2:=^{foo(x+y**2):something else}}`
17. **Datetime formatting** - Multi-line format specs with `%Y %m %d`
18. **Debug expressions** - `{2+2=}` with various spacing

String Prefixes

19. **Raw f-strings** - `rf'foo'` and `fr'foo'`
20. **Raw f-strings with quote reuse** - `rf"raw: {data["key"]}"`

Special Cases

21. **Conversion flags** - `{X!r}` for repr conversion

```
22. **Walrus operator** - `{(abc:=10)}`
23. **Unicode names** - `\N{GREEK CAPITAL LETTER DELTA}`
24. **Escape sequences** - Testing escaped quotes and
backslashes
25. **Long strings with reflow** - Testing formatter's line-
length handling
```

If you're using regression tests, you should also select the portion of the issue description that mentions running tests and map it to all the regression tests with an explanation like:

```
We are using the remaining test_format.py tests are regression
tests because the AI agent will need to change some core
parsing functionality and we want to make sure it doesn't break
any of the existing behavior.
```

Summary

That's it! We've done one full annotation end-to-end.

To summarize, we've started by choosing a PR from our approved list.

We then examined the PR, reproduced the issue locally and wrote a new set of unit tests using Claude Code.

We then wrote the issue description with key hints to provide insights on how to solve it.

We then ran the AI agent to ensure that the issue description is sufficient to solve the issue, i.e. that it's complete and unambiguous. It's extremely important that the tests only fail if the solution is incorrect, and only pass if the solution is correct.

Finally, we wrote some explanations for attempt to ensure this is true, and map the issue description to tests.

At this point, we should be 100% certain that the PR (i.e. the fix patch), the tests and issue description are all in sync.

The next step will be to do one full annotation yourself, but feel free to look back at this example as a reference for your annotation. If you want to work on a different repository, you'll also find information in the Tips section about how to set them up and tips on how to write tests etc.

Your next step is to do one full annotation yourself as part of the qualification. We have pre-selected a small number of PRs for which should be of appropriate level of difficulty and should be possible to write tests. In some cases tests already exist, but are incomplete, in which case you'll just need to make sure they fully cover all the implemented functionality.

For this qualification, there is also a Forum that you can use to ask questions during the qualification phase. Access it on the usual "Forum" link in the navbar.

Using the Trial and Final runners

The Trial and Final runner run the AI agent to see if it can solve the issue. The agent is running inside the Docker container you prepared and is given the issue description (without and with hints). The agent doesn't have access to the internet, so everything that is needed to solve the issue needs to be inside the Docker container.

After the AI agent is finished, we take a "git diff" of the changes, we apply the Test Patch and run the Test Execution Script. The Test Path and Test Execution Script are not accessible to the agent. The agent is running as the "model" username, while all the testing is done as the "root" username. This is to ensure that the agent cannot "hack" the issue by looking at how it's being evaluated and then using that to solve the issue instead of the issue description you prepared.

Trial Runner

The Trial Runner is running in our infra and is using a weaker, but faster model. It's usually faster to run, but some of the attempts might fail. We are working to improve reliability, but for now you can assume that in most cases some

attempts might fail. Nonetheless, the majority of attempts should still complete.

It's always worth exploring individual attempts, even if there are some failed ones. For example, you might find that the remaining successful attempts are working as intended. For example, out of 5 attempts, 2 might produce an error, but 3 might work correctly and out of these in 1 tests pass and in 2 tests fail. This implies a 33% test pass rate, which is within our range of 0-50%.

We recommend using the Trial Runner for the fast iteration, and only progressing to the Final Runner if the Trial Runner is not working, or if the Trial runs are within expected bound.

Final Runner

The Final Runner in running on client infra. Historically it's been a bit less table, with frequent outages, but in the last week or so it has been quite stable with no major outages. This is a high priority project for the client, so we are expecting a near-100% uptime in the future with rapid improvements.

While using the Trial Runner is optional, you need to use the Final Runner to get the final runs that need to be annotated, and that form part of what we are delivering to the client.

If you are finding that the Final Runner is not working, it's best to wait between 15 minutes and a few hours and try again. If it's still not working, please report this to us and we'll escalate, but we expect long outages to be relatively rare in the future.

Interface buttons

The interface has the following buttons:

- **Start new agent run** - this will start a fresh agent run, discarding all the existing data
- **Resubmit failed attempts** - this will only resubmit the incomplete and failed attempts, while keeping the attempts that succeeded. You should always try to use this first to avoid having to redo successful attempts

- **Sync with new data** - this button appears if there is some new data on the backend. Normally this should happen automatically, but sometimes the run shows as failed, while in fact it's running in the background. If this happens, you can manually pull new data.
- **Cancel** - this will cancel any attempts that are currently running.
- **Disconnect** - if cancelling a run for some reason fails, you can force disconnect so you can start a new run. Use this as last resort.

In general, your workflow should look like this:

1. If you update the issue description/tests/dockerfile etc you should click on "Start new agent run" to start a new run
2. Monitor if it's running, by click on any attempt, then "Transcript" then "Load Transcript". If you repeat this in a few minutes, you should see new steps appearing at the end. If new steps are not appearing for a long time (e.g. 5-10 minutes), it's possible the run is stuck
3. If the run appears to be stuck, you can just leave it, as in most cases it recovers, or you can use "Cancel" followed by "Resubmit failed attempts" to re-run the ones that failed.
4. If the "Sync with new data" button appears, use it to pull any new data that's available. This might help recover some runs that otherwise look like they have failed.
5. If you are not getting anywhere, wait some time (e.g. 15-60 minutes) and then click on "Start new agent run" to start a fresh run
6. If you still have a problem, email us on `suppoer@joinstellar.ai` or post in the forum with the screenshot of the issue (but don't post your annotation link)

Attempt status codes

The attempt should have one of these statuses:

- **Tests passed** - means the attempt finished correctly, Test Patch was applied, tests were run and they all passed
- **Tests failed** - means the attempt finished correctly. Test Patch was applied, tests were run and some of the tests failed

- **Running** - means that the AI Agent is currently running and we are waiting for it to finish its solution
- **Incomplete** - means that the AI Agent is no longer running, but we don't have output from running tests. This means either that the AI agent died while trying to solve the issue, or there was some problem with applying the Test Patch or running the tests
- **Error** - means that there was an error with the AI Agent and that it failed to run. The only way to recover from an error like this is to use "Resubmit failed attempts" or "Start a new run"

Troubleshooting common issues

1. Running for a long time with no update

Sometimes run can take a long time. Normally a run takes 30-90 minutes on Trial Runner, and 1-3h for the Final Runner.

While it's running, you can select one attempt, click on "Transcript" and "Load Transcript". You'll see the transcript so far. If you do this again in 5 minutes, you should see additional steps appearing at the bottom. If you stop seeing additional steps appearing, it's possible the run is stuck.

If you think the run is stuck, click on "Cancel" and then "Resubmit failed attempts". This will resubmit the incomplete or failed attempts, while keeping the complete attempts.

If the "Sync with new data" button appears, use it to pull any new data that's available. This might help recover some runs that otherwise look like they have failed.

If this doesn't work, you can also use "Start new agent run" to start a completely new run.

2. "Query failed" error

This is a generic error that just says that something has gone wrong.

You should look at the actual attempts to see what the problem is. Look at the Attempt status codes section to understand what the different status codes mean for attempts.

3. Test Patch cannot be applied

The agent attempt might fail with an error like:

```
Tests failed .... Checking patch ... Patch failed to apply
... File <some_filename.py> already exists.
```

To check the AI agent solution, we apply the Test Patch and then run the Test Execution Script. This error indicates that the Test Patch did not apply correctly, and that this is because a filename already exists.

There are two possible ways how this can happen:

1. You didn't name your tests with "oracle" in the name, so both you and the AI agent created `some_filename.py`. So, now because this file has already been created by the AI agent, adding your test file would overwrite it, so applying the Test Patch fails. You should always name your tests with oracle (or something else unique) in the name, to minimize tha change of a clash happening.
2. You accidentally included the name of your test file in the Issue Description. If you mentioned that the agent should created `some_filename.py` in the issue description, and you also add this file, then again there will be a clash.

If in doubt, you can click on "Transcript" and then "Load Transcript" to see what exactly did the AI Agent do, and where this file comes from. The transcript contains all the steps the agent did and is very useful for debugging issue like this.

4. Agent Diff is empty

Sometimes you might get an error that the "Run is Incomplete" because the AI Agent Diff is empty.

You might look at the Transcript and see that the agent actually did make many changes, but they are not reflected in the diff. I

f you look deeper, you might see that the AI agent actually creates a new commit. This then interferes with us creating the diff, because we use "git diff" to create the diff from **unstaged** changes. So, if the agent create a new commit, there will be no unstaged changes and the diff will be empty.

This sometimes happen if your issue description mentions "Creating a PR" or similar language that makes the AI Agent think it needs to commit the changes, instead of just making them. You should remove language like this from the Issue Description, and if the AI Agent still does this, you can also explicitly tell it not to commit any changes using git.

5. Anything else

If you have a different error you cannot understand, please email us on `support@joinstellar.ai` or post in the forum (please just don't put the link to your annotation) with the screenshot of the issue and we'll try to assist as soon as possible.

Tips

Writing the right level of detail in the Issue Description

One of the most important requirement of this task is that the Issue Description is aligned with the test suite.

This means that the AI agent shouldn't need to guess anything that's being tested - it should either be clearly mentioned in the issue description or it should be reasonably inferrable from the issue description and the codebase.

Let's look at some examples

Example 1

Let's assume this function was added in our Fix Patch into `multiply.py`:

```
def multiply_by_2(x=10):
  return x*2
```

Then we wrote this test in `test_multiply_oracle.py`:

```
import multiply from multiply_by_2

def test_multiply():
  # default value
  assert multiply_by_2() == 20

  # non-default value
  assert multiply_by_2(x=2) == 4
```

What do you think is the best Issue Description to describe this? Look at the 3 Issue Descriptions below and try to decide for yourself:

Issue Description #1

```
# Implement a multiply-by-two function

Add a multiply-by-two function, with a default value of 10.
```

Issue Description #2

```
# Implement a multiply-by-two function

Add a multiply-by-two function added into multiply.py, with
a default parameter of x with default of 10.
```

Issue Description #3

```
# Implement a multiply-by-two function

Add a multiply-by-two function added into multiply.py named
`multiply_by_2`, with a default parameter of x with default
of 10.
```

The correct answer is Issue Description #3. This is because in the tests we specifically import the function from the `multiply` python module, so this name needs to be mentioned, we also specifically assume the name is `multiply_by_2` (and not some other valid name), so that needs to be mentioned as well. Finally, in our second test we specifically assume that the parameter is named `x` and that it has a specific default value, so that needs to be mentioned as well.

If we go with Issue Descriptions #1 and #2, it's possible the AI agent will come up with a valid solution according to the description (e.g. name the function `multiply2`) but that will nonetheless fail our tests because that's not what we expect. This is bad, because we want tests to ONLY FAIL when the SOLUTION IS INCORRECT ACCORDING TO THE ISSUE DESCRIPTION. And we want it to only pass when the solution is correct according to the issue description.

Example 2

Now let's look at another example.

Let's assume this function added in our Fix Patch into `multiply.py`:

```
def multiply_numbers(x=10, y=3):
  return x*y
```

Then we wrote this test in `test_multiply_oracle.py`:

```
import multiply from multiply_numbers

def test_multiply():
```

```
# default value
assert multiply_numbers() == 30

# non-default value
assert multiply_numbers(3, 2) == 6
```

What do you think is the best Issue Description to describe this? Look at the 3 Issue Descriptions below and try to decide for yourself:

Issue Description #1

```
# Implement a number multiplication function

Add a number multiplication function, with a default value
of 10.
```

Issue Description #2

```
# Implement a multiply-by-two function

Add a number multiplication function `multiple_numbers` into
multiply.py, with defaults of 10 and 3.
```

Issue Description #3

```
# Implement a multiply-by-two function

Add a number multiplication function `multiple_numbers` into
multiply.py, with two parameters, x and y with defaults 10
and 3.
```

Now, the best one is Issue Description #2. In our tests, we only ever call the function like this `multiply_numbers(3, 2)`, i.e. we never use the x/y

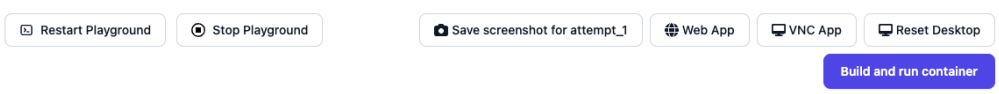
parameter names, therefore, there is no need to mention that they need to be specifically named x and y.

With Issue Description #3 there is a risk of a false positive. Let's say the agent implements the correct solution but for some reasons decides to name the variable a and b. This solution with variables a and b is incorrect according to our issue description (as we are explicitly asking for x and y), however the tests will pass because we never use x and y. So, the solution will be incorrect according to the issue description, but the tests will pass, which is a false positive. We want TESTS TO ONLY PASS IF THE ISSUE IS CORRECTLY SOLVED ACCORDING TO THE ISSUE DESCRIPTION.

Examining attempts with visual changes

Certain repositories, such as Grafana and Discourse will have both frontend and backend. We have built some tools to make it easier to connect to the frontend and even run the app visually inside the container.

Here is how the controls for the VM look like:



Here is what these buttons do:

- **Build and run container** - run the container with the AI attempt diff and Test patch applied
- **Web app** - opens Firefox that connects to the frontend inside the container.
- **VNC app** - opens a VNC client to view the remote desktop inside the container.
- **Reset desktop** - open the desktop associated with the current attempt. Use only if you accidentally switch desktops.
- **Save screenshot** - saves the current screenshot to be given as evidence for your explanation

The "Web app" and "VNC app" are the most useful here.

For the "Web app", we expose the server running in the container on a unique port, and then connect to this part with a Firefox running in the VM. This makes it easy to then manually test the frontend.

For the "VNC app", we offer a remote desktop to the windows manager running in the container. This is the exact same environment that the AI agent is seeing (you'll probably recognize it from the screenshots in transcripts). If you have a desktop app like Godot, you would run it like this, directly from the window manager inside the container.

What to do if the PR already contains tests

Some PRs already have incomplete tests in them. If this is the case, make sure to move these tests from the "Fix Patch" to "Test Patch". To do this:

1. In "Fix Patch", click on "Download diff".
2. Apply the diff to your branch with "git apply fix.patch"
3. Use "git checkout {non_test_files_changed_in_PR}" to revert the non-test changes from the PR. Now you should have the new tests on your branch. So you can commit these changes and create a new "Test Patch" to upload.
4. In "Fix Patch", click on "Edit diff" and delete any changes that are related to tests.
5. Rename all the test file added/modified in the original "Fix Patch" to have "oracle" in their name - this is to ensure that there are no clashes with any files the AI agent might write or edit.
6. Further edit the "oracle" test files to complete the test suite
7. Upload your new "Test Patch"

This will ensure that all of the tests, i.e. the tests from the PR and the newly written tests by you are being used to judge if the AI Agent attempt is correct or not.

Adding files to the Docker container

Sometimes issue description might need to contain screenshots to better explain what the problem is. When this is the case, you need to upload the screenshot into the Docker container so it's available to the agent. The agent doesn't have access to the internet.

To upload a file, do the following:

1. Open the "Docker configuration" section by clicking on this
2. Click on "Build Context Files"
3. Click on "Add File" to add a file to the build context. This will make it available for the "docker build" command
4. Add a command inside the Dockerfile to include it into the image:

```
# ... usual dockerfile content

# put at the end of the Dockerfile
COPY myscreenshot.png /workdir/
```

Now you can rebuild the image by clicking on "Validate" in the "Dockerfile validation" section. You should now see that the png was included in the docker container. You should tell the AI agent where the image is, and you should be able to see the AI agent opening the image in the Transcript of actions.

Working with Grafana Repository

Grafana is a data visualization and monitoring platform built with Go (backend) and TypeScript/React (frontend). The annotation workflow is the same as Godot - the main differences are the programming languages and build tools (`go`/`yarn` instead of `scons`).

You can learn more about Grafana fundamentals and concepts at Grafana Tutorials.

Grafana Local Setup

See [Grafana Developer Guide]() for setup instructions. After installing Go, Node.js and GCC, run:

```
# Frontend setup
corepack enable
yarn install --immutable
yarn start

# Backend setup (in another terminal)
make run
```

Access Grafana at `http://localhost:3000/` with credentials `admin/admin`.

To run tests:

```
# All tests
make test-go       # Backend (Go)
yarn test:ci       # Frontend (TypeScript) - non-interactive
mode

# Specific tests
go test ./pkg/services/featuremgmt/
yarn test:ci packages/grafana-
ui/src/components/Button/Button.test.tsx
```

Grafana Docker and Test Configuration

The annotation interface provides default Dockerfile and Test Execution Script templates for Grafana.

Note: Remember to follow the test file naming guidelines from [Your Task]() section when writing tests for Grafana PRs.

In the Test Execution Script:

- **Backend PRs**: Set `GO_PACKAGES` to your specific Go package paths (example: `"./pkg/services/featuremgmt/"`) and optionally `GO_TEST_RUN` to filter specific tests (example: `"-run TestSpecificFunction"`)
- **Frontend PRs**: Set `JS_FILES` to your specific test file paths (example: `"packages/grafana-ui/src/components/Button/Button.test.tsx"`)
- **Full-stack PRs**: Configure both Go and TypeScript tests with your specific paths

The script automatically merges test results into a single JUnit XML output.

Tip: For frontend-only PRs, set `GO_PACKAGES=""` and optionally comment out Go compilation in the Dockerfile to speed up builds.

Grafana Issue Description

When creating issue descriptions for Grafana PRs, include test execution instructions similar to the Godot example:

```
If you need to run tests, please use the following commands:
- For React/TypeScript tests: `yarn test:ci <test-file-path>`
- For Go tests: `go test <package-path>` or `go test -run
<TestName> <package-path>`
```

You should always include this information as we don't want to make the issue artificially difficult by not giving setup instructions.

Working with Discourse Repository

Discourse is a modern forum and community discussion platform built with Ruby on Rails (backend) and Ember.js (frontend). The workflow is similar to other repositories - the main differences are the programming languages (Ruby/JavaScript instead of C++/Go) and build tools (`bundle`/`pnpm` instead of `scons`/`yarn`).

You can learn more about Discourse development at [Discourse Developer Guide](#) and [Discourse GitHub](#).

Discourse Local Setup

See [Discourse Development Environment](#) for detailed setup instructions. The basic setup requires Ruby, Node.js, PostgreSQL, and Redis.

After installing dependencies, run:

```
# Clone and setup
git clone https://github.com/discourse/discourse.git
cd discourse

# Install Ruby dependencies
bundle install

# Install JavaScript dependencies
pnpm install --frozen-lockfile

# Setup database
bundle exec rake db:create db:migrate
LOAD_PLUGINS=1 bundle exec rake db:migrate

# Start development server
bundle exec rails server
# Access Discourse at http://localhost:3000
```

To run tests:

```
# Backend tests (Ruby/RSpec)
bundle exec rspec                    # All backend tests
bundle exec rspec spec/models/       # Specific directory
bundle exec rspec spec/models/user_spec.rb  # Specific file
```

```
# Frontend tests (JavaScript/QUnit)
bin/rake qunit:test                  # All frontend tests
LOAD_PLUGINS=1 TARGET=all bin/rake qunit:test  # With
plugins
FILTER="Unit | Model | user" bin/rake qunit:test  # Specific
tests

# System tests (full browser automation)
bundle exec rspec spec/system/      # All system tests
```

Discourse Docker and Test Configuration

The annotation interface provides default Dockerfile and Test Execution Script templates for Discourse.

Note: Remember to follow the test file naming guidelines from Your Task section when writing tests for Grafana PRs.

In the Test Execution Script:

- **Backend PRs**: Set `BACKEND_TEST_FILES` to your specific RSpec test file paths (example: `"spec/requests/admin/groups_controller_spec.rb"`)
- **Frontend PRs**: Set `FRONTEND_TEST_FILTER` to your specific QUnit test filter (example: `"Unit | Admin Groups Navigation"`)
- **Full-stack PRs**: Configure both backend and frontend tests with your specific paths

The script uses TAP conversion to generate JUnit XML output from QUnit tests and merges results from both backend and frontend tests.

Key Environment Variables:

- `RAILS_ENV=test` - Ensures test environment
- `DISCOURSE_DB_NAME=discourse_test` - Test database name
- `TESTEM_FIREFOX_PATH=/usr/bin/firefox-esr` - Browser for QUnit tests

- `LOAD_PLUGINS=1` - Include plugin tests when needed

Discourse Issue Description

When creating issue descriptions for Discourse PRs, include test execution instructions similar to the Godot example:

```
If you need to run tests, please use the following commands:
- For Ruby/RSpec tests: `bin/rspec <test-file-path>` or
`bin/rspec <test-file-path>:<line-number>`
- For JavaScript/QUnit tests: `bin/rake qunit:test` (all) or
`FILTER='<test-filter>' bin/rake qunit:test`
- For plugin tests: `LOAD_PLUGINS=1 bin/rspec <plugin-test-
path>` or `LOAD_PLUGINS=1 TARGET=all FILTER='<filter>' bin/rake
qunit:test`
```

Tip: The test script handles PostgreSQL and Redis service startup automatically. For frontend-only PRs, you can set `BACKEND_TEST_FILES=""` to skip backend tests.