Polaris (Fully Tested PRs) Guidelines

- Polaris (Fully Tested PRs) Guidelines
  - Changelog
  - Project Overview
  - Your Task
  - Step 1: Choosing a PR
  - Step 2: Fully Understanding the PR
    - Writing the Fix Patch Description
  - Step 3: Creating the Test Patch - Environment Setup
    - Environment Setup
    - Extracting the Test Suite
    - Testing the New Oracle Test Suite
    - Uploading the Final Test and Fix Patches
  - Step 4: Running Docker Validation
  - Step 5: Writing the Issue Description
    - Zero False Negatives and Zero False Positives
    - How to Write the Issue Description
    - Continuing to Write the Issue Description
    - Using Claude Code to Summarize Test Requirements
    - Magic Handling Requirements
    - File Formatting Requirements
    - Complex Requirements: Jupyter Dependencies
    - Final Issue Description with Regression Tests
  - Step 6: Writing Hints
  - Step 7: Run the AI Agent
  - Step 8: Review the AI Agent Attempts
    - Using AI Review (In Depth)
    - Adjusting the Issue Description

        - Reviewing the Final AI Agent Attempts
        - Common Issues to Watch For
        - Final Notes on Reviewing AI Agent Attempts
    - Using AI Effectively
- Using the Final runner
    - Final Runner
    - Interface buttons
    - Attempt status
    - Troubleshooting common issues
        - 1. Running for a long time with no update
        - 2. "Query failed" error
        - 3. Test Patch cannot be applied ("Incomplete" error)
        - 4. Agent Diff is empty
        - 5. Anything else
- Tips
    - Examining attempts with visual changes
    - 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

- Nov 6, 2025 - Initial version

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 the existing test suite from the PR

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 running the test suite from the PR.

The most important part of this project is the issue description. It's absolutely crucial it contains all the necessary information to pass the tests.

The **expected annotation time is 4-8h** with a **max of 12h**.

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. Create a Docker container with the pre-PR code
4. Write an issue descriptions that accurately describe what is tested in your unit tests
5. Run the agent to verify the issue is sufficiently difficult
6. Annotate agent runs to ensure all test passes are correct solution, and all test fails are incorrect solutions

To make this a bit more clear, these guidelines walk through one task end-to-end using **PR 2357: "Jupyter notebook 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 2357: "Jupyter notebook support"**.

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 4dd100bff283b3c60ce67491dd53f47b45f68ac2
```

We now have the base commit. Let's also apply the PR patch to it, so we can examine the changes.

In the "Fix Patch" section click on "Download". This will download the "fix-patch.diff" file. Rename this file into "pr.patch", move it to the "black" directory and then run:

```
git apply pr.patch
```

Now you should have the PR patch applied as an unstaged commit. Open the codebase in the editor of your choice and examine changes.

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:

```
The implementation treats Jupyter notebooks as JSON files
containing code cells that need formatting, while handling
IPython-specific syntax (magics) that standard Python parsers
can't handle.

## Key Components

### 1. **Magic Handling (`handle_ipynb_magics.py`)**
This is the heart of the implementation. The strategy is to
"mask" IPython magics before formatting:

- **Masking**: Replace IPython magics (like `%matplotlib`,
`!ls`, `%%time`) with randomly generated tokens that are valid
Python
- **Format**: Run Black's standard formatter on the masked code
- **Unmask**: Replace tokens back with original magic syntax

The module identifies magics by:
- Using IPython's `TransformerManager` to transform magics into
standard Python calls (e.g., `!ls` →
`get_ipython().system('ls')`)
```

- Using AST visitors (`MagicFinder`, `CellMagicFinder`) to locate these transformed calls
- Replacing them with tokens, formatting, then reversing the process

2. **Cell-by-Cell Processing**
The `format_ipynb_string()` function:
- Parses the notebook JSON
- Validates it's a Python notebook (checks metadata)
- Iterates through cells, formatting only code cells
- Preserves trailing newlines if present
- Returns formatted JSON

Each cell goes through `format_cell()` which:
- Removes trailing semicolons (Jupyter convention for suppressing output)
- Masks magics
- Formats the masked code
- Unmasks and restores semicolons
- Strips trailing newlines

3. **Safety Checks**
The implementation is conservative:
- Skips cells with syntax errors (might be automagics or multiline magics)
- Skips cells already containing transformed magics (to avoid double-transformation)
- Skips non-Python cell magics (like `%%bash`, `%%html`)
- Skips non-Python notebooks entirely

4. **Integration Points**
- **CLI**: Adds `--ipynb` flag to force notebook mode
- **File detection**: Updates `DEFAULT_INCLUDES` regex to match `.ipynb` files
- **Mode system**: Adds `is_ipynb` flag to the `Mode` dataclass
- **Dependency checking**: Gracefully handles missing

```
IPython/tokenize-rt dependencies with
`jupyter_dependencies_are_installed()`
- **Pre-commit**: Adds separate `black-jupyter` hook with
`[jupyter]` dependencies

### 5. **Diff Support**
Special `ipynb_diff()` function that:
- Parses both notebook versions
- Generates diffs cell-by-cell for code cells only
- Shows meaningful diffs rather than raw JSON changes
```

Please put this explanation in the textbox at the bottom of the "Fix patch" section.

Step 3: Creating the Test Patch - Environment Setup

Each PR already has a comprehensive test suite. What we'll need to do is to make sure the test suite can be run, and separate it out from the original PR.

So, let's start by making sure we can run the test suite.

We provide default setup instructions for each repository, however, these might not always work, as the PR might be old and might require a specific set of old dependencies to run.

This is the case for our chosen PR as well. The PR is from a few years ago, and our default script (that you can see in the "Dockerfile Configuration") section won't just work.

In this project we use uv for environment management. Please [download and setup uv from here](). Please do not use any other environment management solution.

Next, we **highly recommend using Claude Code to help with your annotation**, otherwise you might struggle with certain setup and code analysis

tasks that Claude Code can do pretty easily, this will save you many hours of work spent debugging configuration and other annoying issues.

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.

OK, so now that we have uv and Claude Code installed, we are ready to start setting up the environment for this repository.

Environment Setup

Open the "Dockerfile configuration" section in your annotation. This will have the default dockerfile provided by us. At the end you'll see some pip install commands that are used to setup the uv environment in the container. The AI agent will run in this container, so it's important that the environment is correctly configured, so the agent can focus on the issue and not on env setup.

Let's just start by trying to setup the environment the same way as in the Docker container. (if you are using conda, please remember to first do "conda deactivate" as uv and conda might interfere with each other)

```
uv venv --python 3.11
source .venv/bin/activate
uv pip install -e .
uv sync --all-extras
uv pip install -r test_requirements.txt
```

As you'll notice, the "uv sync --all-extras" command will fail, because these extras were only added later to the repository. Nonetheless, let's just try to run the tests:

```
pytest
```

You should get an error "ImportError: Error importing plugin "tests.optional": No module named '_pytest.store'". This is because our environment is

misconfigured. We are using a newer version of pytest that's incompatible with this old PR.

So, we'll first need to fix this environment so we can run the tests. Let's remove the old environment and then start Claude Code to help us.

```
deactivate
rm -rf .venv/
claude
```

Now you should have Claude Code started. Let's ask it to fix this environment:

```
I want to set up a uv environment to run the repository at the
current directory, that correspond to an old PR. Ideally I want
to use python 3.11, but could downgrade down to 3.9 if necessary
(but first try to get it working with 3.11). I want to be able
to run tests with "pytest", including the new tests that I just
added (see unstaged changes).

The script I have is this, but it does not work for me (it looks
like the pytest is not at the right version):
```

uv venv --python 3.11 source .venv/bin/activate uv pip install -e . uv sync --all-extras uv pip install -r test_requirements.txt

```
Can you try this and investigate what the problem is. If any
libraries need down grading, please do so. At the end, tell me
what are the correct commands to set up the env. The setup is
complete when pytest is running fully.
```

Claude Code will try our script, and then continue debugging until it finds the right versions of all the dependencies. This should take a few minutes. In the end, Claude Code should recommend using something like this:

```
uv venv --python 3.11
uv pip install -e .
```

```
uv pip install -r test_requirements.txt
uv pip install "click>=7.1.2,<8.0" "pytest>=6.0,<7.0"
"pytest-xdist>=2.0,<3.0" "typing_extensions"
"ipython>=7.8.0,<8.0" "tokenize-rt>=3.2.0,<5.0"
```

Great, so now our environment is set up. Let's update the Dockerfile with this information as well.

Open the "Dockerfile Configuration" section and replace the dependencies installation RUN command with this:

```
# Install Python dependencies using uv
RUN cd /workdir/black && \
    uv venv --python 3.11 && \
    uv pip install -e . && \
    uv pip install -r test_requirements.txt && \
    uv pip install "click>=7.1.2,<8.0" "pytest>=6.0,<7.0"
"pytest-xdist>=2.0,<3.0" "typing_extensions"
"ipython>=7.8.0,<8.0" "tokenize-rt>=3.2.0,<5.0"
```

Now the Docker container should also be all configured to be able to run the test suite.

Extracting the Test Suite

The original "Fix Patch" from the PR contains both the fix and the tests. We want to remove the tests from the "Fix Patch" and put them into a separate "Test Patch". This "Test Patch" will be run after the AI agent tries to solve the issue to verify if the solution is correct or not. While we'll use the "Fix Patch" just to verify that our tests work (i.e. fail before the patch is applied, success after it's applied).

So, let's start with moving the tests from "Test Patch" to "Fix Patch". The PR adds test_ipynb.py, test_no_ipynb.py and modifies test_black.py.

Since the "Test Patch" will be used to judge if the AI Agent solution is correct or not, we need to make sure that the "Test Patch" doesn't clash with any changes that the AI agent might make. While solving the issue, the AI agent might

actually itself modify test_black.py, or write test files with the same name. This would then result in a clash, and our test wouldn't work.

When moving the test suite from the Fix Patch to the Test Patch **always follow these rules**:

1. Any newly added test file should be renamed to have _oracle in its name
2. Any modified test file should be copied to have _oracle in its name

The idea of adding "_oracle" to the name is to ensure that these new test files are unique.

Renaming and copying files might be complicated, because there might be e.g. multiple files, some of which reference newly introduced data files, etc. So, it can be a lot of work, but it's a pretty mundane task that once again Claude Code is very competent at.

So, let's ask Claude Code again to do this for us:

```
I have applied the patch in `pr.patch` in the current directory.
I now want to make a copy of all the modified and newly
introduced test files so they have _oracle in the name. So copy
test_black.py to test_black_oracle.py, copy test_ipynb.py to
test_ipynb_oracle.py, copy test_no_ipynb.py to
test_no_ipynb_oracle.py. Also copy all the associated data files
to have _oracle in the name and modify any references in the
copied files to use these new data files with _oracle in the
name.
```

After a few minutes, you should have all the files and associated data files renamed. Now, let's try to run them to verify that all the tests are passing.

Let's start by running the new `test_ipynb_oracle.py` test suite:

```
pytest tests/test_ipynb_oracle.py
```

The suite suite should work, but you should get a single failure in test_set_input(). This is unexpected because we have the PR patch applied, so it

should be working. As it turns out, this seems to be a problem with the PR. This sometimes happens. As we want all tests to pass once the issue is fixed, we'll remove this test. Edit the "tests/test_ipynb_oracle.py" to remove just this test test_set_input().

Next, let's run the second file:

```
pytest tests/test_no_ipynb_oracle.py
```

You should see that both tests in this file are being skipped. This is because this test is supposed to be run in a different environment where no jupyter dependencies are installed. Since our project does not allow us to run tests in multiple environments, we are just going to remove this file from the suite. So just delete the file "tests/test_no_ipynb_oracle.py".

In general, you should never remove anything from the test suite unless: (1) It's broken, (2) It cannot be run. We want to keep the test suite as close as possible to what's in the PR, but sometimes it's just not possible. In 99% of PRs, you'll want to keep all the tests that are part of the test suite.

Finally, let's run the last file:

```
pytest tests/test_black_oracle.py
```

This file contains the main part of the test suite from Black. As such, it's useful as a **regression test suite**. While the "test_no_ipynb_oracle.py" ensures that the new functionality works correctly, this "test_black_oracle.py" regression suite ensures that the existing functionality is not broken. This is quite important.

It's quite important to have a regression test suite, as we normally want existing functionality not to be broken. It's up to you to choose what part of the original test suite you want to run. In general you don't want to run everything, as this might require additional dependencies or additional setup. You should choose as your regression suite a subset of tests that is most likely to measure any "adjacent" functionality that the AI agent might inadvertently break. As such, test_black_oracle.py is a very good candidate.

Once you run this, you'll once again see that one test fails: test_docstring_reformat_for_py27(). Edit the test_black_oracle.py to remove this test.

Great, so now we have a clean test suite with tests for new functionality (test_no_ipynb_oracle.py) and old functionality that should be preserved (test_black_oracle.py).

Testing the New Oracle Test Suite

At the moment, in our local copy of the repository we have both the PR diff applied, and our new test suite.

Now we want to just separate our new "_oracle" test suite.

To keep ourselves organized, as we might need to modify the suite further, let's start by making a new branch:

```
git checkout -b feature/tests-2357
```

Next, let's revert the PR diff, so we are only left with our test suite:

```
git apply -R pr.patch
```

This will revert the PR patch, but leave our oracle files, which you can verify by running "git status".

Now, let's commit these to a new commit:

```
git add tests/
git commit -m "Adding test suite"
```

Good, so we've verified that the tests succeed when the PR diff is applied. But what about the base commit? Since we've for "git apply -R" above, we are now back at the base commit (plus our test suite).

So, let's try to run the test suite in the base commit:

```
pytest tests/test_ipynb_oracle.py tests/test_black_oracle.py
```

As we can see, the test suite is now completely broken, we are getting: "ModuleNotFoundError: No module named 'black.handle_ipynb_magics'". This is because our test suite imports this file that doesn't exist in the base commit. This is not great, because we still want to be able to run the test suite even if this file doesn't exist.

Once again, let's turn to Claude Code to help us out. In Claude Code, write this prompt:

```
I'm trying to run the test suite with `pytest
tests/test_ipynb_oracle.py tests/test_black_oracle.py` but there
is an error. I think the problem is that test_ipynb_oracle.py is
importing some modules that don't (yet) exist. Can you try to
run these tests yourself, then make this import conditional, so
that even if this module doesn't exist we can still run the
tests. Do not skip the tests if the module doesn't exist, but
make the corresponding tests fail (e.g. by mocking the missing
functions with None).
```

Claude Code will make some changes that will make the tests actually fail, instead of getting the test collection error. You can [download the version of test_ipynb_oracle.py after the Claude Code changes here](#).

Add these changes to your branch and commit them:

```
git add tests/
git commit -m "Make sure tests run even if modules don't
exist"
```

Great, so now we have a test suite that works in the base commit (where some tests fail because the issue is not fixed), and that fully passes once the PR patch is applied.

Uploading the Final Test and Fix Patches

So now we have a local version of the test suite that works. Next we need to upload it to the platform.

Let's create a patch from our local repository:

```
git diff 4dd100bff283b3c60ce67491dd53f47b45f68ac2..HEAD --
output=test.patch
```

This is going to create a [Test Patch that looks like this (download)](#).

Open the "Test Patch" section in the interface and click on "Import Diff" and choose the "test.patch" file you just created (or if you are having trouble generating this file you can use our version from above).

Since we made some changes to the Test Suite, we need to explain what we did. In "Comment code changes" textbox at the end of the "Test Patch" section write this:

```
Removed the test_no_ipynb.py test, as it requires re-running
tests in a different env. Also removed
test_ipynb.py::test_set_input().because it was failing in post-
PR, and test_black.py::test_docstring_reformat_for_py27()
because it was failing in the base commit.

The import code in test_ipynb.py was refactored, so the tests
can be run even if the required functions such as
jupyter_dependencies_are_installed and format_cell don't exist
yet.
```

Next, we'll need to modify the "Fix Patch" to remove all the test files from it. In the "Fix Patch" section click on "Edit Diff" and then remove all the tests from the diff. Alternatively, you can download the original PR using "Download diff", edit on your computer and then re-upload with "Import diff file". Your [final "Fix Patch" should look like this (download)](#).

So, now you should have a "Fix patch" that just contains the fix, without any tests. And you should have a "Test patch" that only contains our "_oracle" tests in it.

Step 4: Running Docker Validation

The next step is validate that the Docker contains is working as intended.

So far, we've modified the Dockerfile with our custom setup script (where we got Claude Code to figure out the right dependencies), and we've uploaded the Fix and Test Patches. [Your Dockerfile should look like this (download)](#).

Now, 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_ipynb_oracle.py tests/test_black_oracle.py
```

So, the only thing we've done here is to put in the names of the test files we want to run.

Now we are ready to verify that everything is working. Open the "Dockerfile Validation" section and click on "Pre-PR Validation" and then "Validate".

This will start validation in the scenario before the fix patch is applied. It will build the Docker container, check out the base commit and then run the tests from the Test Patch. The expectation is that the tests will fail. This will take a few minutes as we'll need to build the docker container and run the tests. If you've done all the steps so far, you should find that the test correctly fail.

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. Again, if you've done everything correctly so far, you'll see that the tests pass.

Great, so this means our container is working as intended. The environment works, we can run tests at base commit (which then fail), we can apply the test patch and run the tests (which then pass).

Everything is now ready for the AI agent to try to solve the issue.

Step 5: Writing the 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 all information that's necessary to pass the tests**. 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.

Writing a good issue description is the core of this task, and is the most complex part of it. For other parts we could use Claude Code with minimal intervention, but here we'll really need to go deep in order to be able to do this well.

Zero False Negatives and Zero False Positives

Writing the Issue Description is the most important and critical part of the task.

In the Issue Description we need to provide **all the necessary information** for the AI agent to be able to pass the tests.

The most important requirement you need to remember for this whole project is **ZERO FALSE NEGATIVES and ZERO FALSE POSITIVES**:

1. **ZERO FALSE NEGATIVES** - a false negative is a situation where the tests fail (hence "negative"), but the AI agent has correctly solved the task we described in the Issue Description. It's a "false" negative because the agent did what it was asked it to do, but the tests still failed. We want to ensure this DOES NOT HAPPEN.
2. **ZERO FALSE POSITIVES** - a false positive is a situation where all the tests pass (hence "positive"), but the AI agent has not solved the problem as we described it. This can happen because we included things in the Issue Description that are not actually tested, or if we didn't include a regression test suite (so our tests cannot detect when the agent messes up existing code).

In this project we are very strict on this, and both you and our QA team will be spending the majority of your time making sure we meet these two requirements.

How to Write the Issue Description

To meet the the zero false negatives / zero false positives requirement, we'll need to make sure the Issue Description is perfectly aligned with the tests.

We can start by checking the PR description and the associated solved GitHub issue. This has some information about the PR, but nearly enough to pass the tests.

To ensure that the issue description is perfectly aligned with the test suite, we recommend using the test suite as a basis for writing the issue description.

Let's start by looking at the "tests/test_ipynb_oracle.py" file. The first two tests: test_noop() and test_trailing_semicolon() can be quite useful to explain the expected behavior. We can convert them into a full example (i.e. with all the imports, etc) of what the expected behavior should look like.

So, let's do exactly that. Let's look at those tests, and convert them into a clear example of Expected Behavior. Let's start by writing the "Issue Description" like this:

```
# Issue: Add Jupyter Notebook Formatting Support to Black

## Overview

Add support for formatting Jupyter notebook (`.ipynb`) files
in Black, preserving IPython magic commands and Jupyter-
specific behaviors while applying Black's formatting rules
to Python code in cells.

## Expected behavior

Here is a minimal example of the new expected behavior

from black import (
    NothingChanged,
    format_cell,
    format_file_contents,
    format_file_in_place,
)

# new mode for jupyter notebooks
JUPYTER_MODE = Mode(is_ipynb=True)

# Already correctly formatted should raise `NothingChanged`
src = 'foo = "a"'
format_cell(src, fast=True, mode=JUPYTER_MODE)   # should
raise `NothingChanged` as it's already correctly formatted

# Reformatting should work the same as in other places in
black
src = 'foo = "a" ;'
```

```
result = format_cell(src, fast=fast, mode=JUPYTER_MODE)
assert results == 'foo = "a";'   # reformated code
```

This minimal example is good because it let's us convey a lot of information - that there should be a new function "format_cell", that there should be a new "Mode(is_ipynb=True)", that if nothing changes this should raise a "NothingChanged" exception, etc.

This really helps to narrow down the solution, and exactly specifies how these functions and parameters should look like to align with our tests. If we just described this in words, it's possible the AI agent would find some other way of naming functions that would be correct based on our description but would still fail the tests (hence resulting in false negative).

So, this is a good start, so let's continue writing the Issue Description to cover all the tests.

Continuing to Write the Issue Description

Let's continue writing the issue description.

It's clear that "format_cell()" is the major new function being tested in all of our tests. So, let's explain a bit more what it does.

Apply the "Fix Patch" locally on your repository with "git apply fix.patch" and then open the function definition for "format_cell()". As you'll see there is a helpful explanation of what the function does, we can use that in our explanation. We'll also need to explain what exactly the fast parameter does.

Add this to your existing issue description from last step:

```
## Cell formatting

The cell formatting function should have this signature:

def format_cell(src: str, *, fast: bool, mode: Mode) -> str:

The expected behavior is that:
```

```
    - if cell has trailing semicolon, remove it;
    - if cell has IPython magics, mask them;
    - format cell;
    - reinstate IPython magics;
    - reinstate trailing semicolon (if originally present);
    - strip trailing newlines.

    Cells with syntax errors will not be processed, as they
    could potentially be automagics or multi-line magics, which
    should not be supported yet.

    The `fast` parameter is True, it should skip the stability
    and equivalence steps we would normally perform.

    When nothing is changed, the function should raise the
    `NothingChanged` exception.
```

This explains briefly what the expected behavior of the function is. It describes this behavior in terms of how the function is actually implemented, which is not ideal (as it might introduce requirements we don't directly test - therefore risking a false positive). However, in our case this is OK as there isn't a good other way to explain expected behavior.

Nonetheless, you should be careful not to mention something in the issue description that's not directly tested, not to risk a false positive.

Using Claude Code to Summarize Test Requirements

OK, so let's continue looking at the "tests/test_ipynb_oracle.py" file. We've covered the first two tests "test_noop" and "test_trailing_semicolon" with our example of expected behavior. Now, these are followed by a bunch of tests that test various edge cases of semicolon processing.

We can use Claude Code to help us summarize what the expected behavior is, as specified in these tests.

Let's use Claude Code with this prompt:

```
Can you summarize the expected behavior in these trailing
semicolon tests in as few words as possible:

def test_noop() -> None:
    src = 'foo = "a"'
    with pytest.raises(NothingChanged):
        format_cell(src, fast=True, mode=JUPYTER_MODE)


@pytest.mark.parametrize("fast", [True, False])
def test_trailing_semicolon(fast: bool) -> None:
    src = 'foo = "a" ;'
    result = format_cell(src, fast=fast, mode=JUPYTER_MODE)
    expected = 'foo = "a";'
    assert result == expected


def test_trailing_semicolon_with_comment() -> None:
    src = 'foo = "a" ;  # bar'
    result = format_cell(src, fast=True, mode=JUPYTER_MODE)
    expected = 'foo = "a";  # bar'
    assert result == expected


def test_trailing_semicolon_with_comment_on_next_line() -> None:
    src = "import black;\n\n# this is a comment"
    with pytest.raises(NothingChanged):
        format_cell(src, fast=True, mode=JUPYTER_MODE)


def test_trailing_semicolon_indented() -> None:
    src = "with foo:\n    plot_bar();"
    with pytest.raises(NothingChanged):
        format_cell(src, fast=True, mode=JUPYTER_MODE)
```

```
def test_trailing_semicolon_noop() -> None:
    src = 'foo = "a";'
    with pytest.raises(NothingChanged):
        format_cell(src, fast=True, mode=JUPYTER_MODE)
```

You might need to play around with this prompt a bit to get the right level of detail. For example, you might ask Claude to increase or decrease the level of detail. We want the behavior to be clearly explained and not be composed of long list of requirements that are impossible to understand.

After a few iterations, you might come up with this to add to your Issue Description:

```
### Semicolon handling

Semicolons should only be formatted if:
1. They have spaces before them that need removal, OR
2. The code itself needs formatting

If the code is already well-formatted and the semicolon has
no space before it, raise NothingChanged even if a semicolon
is present.

So for example, spaces before trailing semicolons are
removed, but before comments preserved, so `foo = "a" ;  #
bar` should become `foo = "a";  # bar`. And this input
`import black;\n\n# this is a comment` should not be changed
and should raise NothingChanged.
```

Notice how we've provided some minimal examples to make it really crystal clear what we mean. It's fine to provide example, but DO NOT overdo it. We DO NOT want the whole issue description to be just code snippets from the test suite.

Magic Handling Requirements

Let's continue with looking at further tests in "tests/test_ipynb_oracle.py" and converting them into requirements.

The next set of tests are checking magics handling. Let's once again use Claude Code to summarize the expected behavior:

```
Now, can you summarize the expected behavior for magics here in
as few words as possible (while keeping it precise):

def test_cell_magic() -> None:
    src = "%%time\nfoo =bar"
    result = format_cell(src, fast=True, mode=JUPYTER_MODE)
    expected = "%%time\nfoo = bar"
    assert result == expected


def test_cell_magic_noop() -> None:
    src = "%%time\n2 + 2"
    with pytest.raises(NothingChanged):
        format_cell(src, fast=True, mode=JUPYTER_MODE)


@pytest.mark.parametrize(
    "src, expected",
    (
        pytest.param("ls =!ls", "ls = !ls", id="System
assignment"),
        pytest.param("!ls\n'foo'", '!ls\n"foo"', id="System
call"),
        pytest.param("!!ls\n'foo'", '!!ls\n"foo"', id="Other
system call"),
        pytest.param("?str\n'foo'", '?str\n"foo"', id="Help"),
        pytest.param("??str\n'foo'", '??str\n"foo"', id="Other
help"),
        pytest.param(
            "%matplotlib inline\n'foo'",
            '%matplotlib inline\n"foo"',
```

```
            id="Line magic with argument",
        ),
        pytest.param("%time\n'foo'", '%time\n"foo"', id="Line
magic without argument"),
    ),
)
def test_magic(src: str, expected: str) -> None:
    result = format_cell(src, fast=True, mode=JUPYTER_MODE)
    assert result == expected


@pytest.mark.parametrize(
    "src",
    (
        "%%bash\n2+2",
        "%%html --isolated\n2+2",
    ),
)
def test_non_python_magics(src: str) -> None:
    with pytest.raises(NothingChanged):
        format_cell(src, fast=True, mode=JUPYTER_MODE)


def test_input_already_contains_transformed_magic() -> None:
    src = '%time foo()\nget_ipython().run_cell_magic("time", "",
"foo()\n")'
    with pytest.raises(NothingChanged):
        format_cell(src, fast=True, mode=JUPYTER_MODE)


def test_magic_noop() -> None:
    src = "ls = !ls"
    with pytest.raises(NothingChanged):
        format_cell(src, fast=True, mode=JUPYTER_MODE)
```

```
def test_cell_magic_with_magic() -> None:
    src = "%%t -n1\nls =!ls"
    result = format_cell(src, fast=True, mode=JUPYTER_MODE)
    expected = "%%t -n1\nls = !ls"
    assert result == expected


def test_cell_magic_nested() -> None:
    src = "%%time\n%%time\n2+2"
    result = format_cell(src, fast=True, mode=JUPYTER_MODE)
    expected = "%%time\n%%time\n2 + 2"
    assert result == expected


def test_cell_magic_with_magic_noop() -> None:
    src = "%%t -n1\nls = !ls"
    with pytest.raises(NothingChanged):
        format_cell(src, fast=True, mode=JUPYTER_MODE)


def test_automagic() -> None:
    src = "pip install black"
    with pytest.raises(NothingChanged):
        format_cell(src, fast=True, mode=JUPYTER_MODE)


def test_multiline_magic() -> None:
    src = "%time 1 + \\\n2"
    with pytest.raises(NothingChanged):
        format_cell(src, fast=True, mode=JUPYTER_MODE)


def test_multiline_no_magic() -> None:
    src = "1 + \\\n2"
    result = format_cell(src, fast=True, mode=JUPYTER_MODE)
    expected = "1 + 2"
```

```
    assert result == expected


def test_cell_magic_with_invalid_body() -> None:
    src = "%%time\nif True"
    with pytest.raises(NothingChanged):
        format_cell(src, fast=True, mode=JUPYTER_MODE)
```

After a bit of tuning and iteration, this should produce following text you can append to your Issue Description:

```
### Magic handling

In general we want to preserve IPython magic syntax and
format Python code that follows magics:
- Cell magics (%%time, %%bash) stay at the start; Python
code after them is formatted
- Line magics (%, !, ?, ??) are preserved; code on the same
line or after is formatted
- Non-Python cell magics (%%bash, %%html) skip formatting
entirely
- Automagics (magic commands without % prefix), multiline
magics (with \), and invalid Python after magics trigger no
changes
- Already-transformed magics (e.g.,
get_ipython().run_cell_magic()) are left alone
- Nested cell magics are both preserved while Python code is
formatted

So, for example:
- `ls =!ls` should be formatted as `ls = !ls`
- `%%t -n1\nls = !ls` should not be reformatted (should
raise NothingChanged)
- `%time 1 + \\\n2` should not be reformatted (should raise
NothingChanged)
```

Once again, we opted for a clear list of requirements, with some minimal examples to further contextualize what we mean. This should be precise enough for an expert programmer to understand exactly what is the expected behavior, with no ambiguity at all.

File Formatting Requirements

The next set of tests are dealing with calling formatting on files. Once again let's use Claude Code, and now also specifically tell it to inspect any associated files to really understand the expected behavior is:

```
Great, next can you summarize the expected behavior in these
tests in as few words as possible. Especially focus on the new
requirements that weren't covered before, and that stem from
processing whole files. These tests have external data files.
Open each one of them to fully understand what's the expected
behavior.
```

```
def test_entire_notebook_empty_metadata() -> None:
    with open(
        os.path.join("tests", "data",
"notebook_empty_metadata_oracle.ipynb"), "rb"
    ) as fd:
        content_bytes = fd.read()
    content = content_bytes.decode()
    result = format_file_contents(content, fast=True,
mode=JUPYTER_MODE)
    expected = (
        "{\n"
        ' "cells": [\n'
        "  {\n"
        '   "cell_type": "code",\n'
        '   "execution_count": null,\n'
        '   "metadata": {\n'
        '    "tags": []\n'
        "   },\n"
        '   "outputs": [],\n'
```

```
        '   "source": [\n'
        '    "%%time\\n",\n'
        '    "\\n",\n'
        '    "print(\\"foo\\")"\n'
        "   ]\n"
        "  },\n"
        "  {\n"
        '   "cell_type": "code",\n'
        '   "execution_count": null,\n'
        '   "metadata": {},\n'
        '   "outputs": [],\n'
        '   "source": []\n'
        "  }\n"
        " ],\n"
        ' "metadata": {},\n'
        ' "nbformat": 4,\n'
        ' "nbformat_minor": 4\n'
        "}\n"
    )
    assert result == expected


def test_entire_notebook_trailing_newline() -> None:
    with open(
        os.path.join("tests", "data",
"notebook_trailing_newline_oracle.ipynb"), "rb"
    ) as fd:
        content_bytes = fd.read()
    content = content_bytes.decode()
    result = format_file_contents(content, fast=True,
mode=JUPYTER_MODE)
    expected = (
        "{\n"
        ' "cells": [\n'
        "  {\n"
        '   "cell_type": "code",\n'
```

```
        '   "execution_count": null,\n'
        '   "metadata": {\n'
        '    "tags": []\n'
        "   },\n"
        '   "outputs": [],\n'
        '   "source": [\n'
        '    "%%time\\n",\n'
        '    "\\n",\n'
        '    "print(\\"foo\\")"\n'
        "   ]\n"
        "  },\n"
        "  {\n"
        '   "cell_type": "code",\n'
        '   "execution_count": null,\n'
        '   "metadata": {},\n'
        '   "outputs": [],\n'
        '   "source": []\n'
        "  }\n"
        " ],\n"
        ' "metadata": {\n'
        '  "interpreter": {\n'
        '   "hash":
"e758f3098b5b55f4d87fe30bbdc1367f20f246b483f96267ee70e6c40cb185d8"\n'
        "  },\n"
        '  "kernelspec": {\n'
        '   "display_name": "Python 3.8.10 64-bit (\'black\':
venv)",\n'
        '   "name": "python3"\n'
        "  },\n"
        '  "language_info": {\n'
        '   "name": "python",\n'
        '   "version": ""\n'
        "  }\n"
        " },\n"
        ' "nbformat": 4,\n'
        ' "nbformat_minor": 4\n'
```

```
        "}\n"
    )
    assert result == expected


def test_entire_notebook_no_trailing_newline() -> None:
    with open(
        os.path.join("tests", "data",
"notebook_no_trailing_newline_oracle.ipynb"), "rb"
    ) as fd:
        content_bytes = fd.read()
    content = content_bytes.decode()
    result = format_file_contents(content, fast=True,
mode=JUPYTER_MODE)
    expected = (
        "{\n"
        ' "cells": [\n'
        "  {\n"
        '   "cell_type": "code",\n'
        '   "execution_count": null,\n'
        '   "metadata": {\n'
        '    "tags": []\n'
        "   },\n"
        '   "outputs": [],\n'
        '   "source": [\n'
        '    "%%time\\n",\n'
        '    "\\n",\n'
        '    "print(\\"foo\\")"\n'
        "   ]\n"
        "  },\n"
        "  {\n"
        '   "cell_type": "code",\n'
        '   "execution_count": null,\n'
        '   "metadata": {},\n'
        '   "outputs": [],\n'
        '   "source": []\n'
```

```
        "  }\n"
        " ],\n"
        ' "metadata": {\n'
        '  "interpreter": {\n'
        '   "hash":
"e758f3098b5b55f4d87fe30bbdc1367f20f246b483f96267ee70e6c40cb185d8"\n'
        "  },\n"
        '  "kernelspec": {\n'
        '   "display_name": "Python 3.8.10 64-bit (\'black\':
venv)",\n'
        '   "name": "python3"\n'
        "  },\n"
        '  "language_info": {\n'
        '   "name": "python",\n'
        '   "version": ""\n'
        "  }\n"
        " },\n"
        ' "nbformat": 4,\n'
        ' "nbformat_minor": 4\n'
        "}"
    )
    assert result == expected


def test_entire_notebook_without_changes() -> None:
    with open(
        os.path.join("tests", "data",
"notebook_without_changes_oracle.ipynb"), "rb"
    ) as fd:
        content_bytes = fd.read()
    content = content_bytes.decode()
    with pytest.raises(NothingChanged):
        format_file_contents(content, fast=True,
mode=JUPYTER_MODE)
```

```
def test_non_python_notebook() -> None:
    with open(os.path.join("tests", "data",
"non_python_notebook_oracle.ipynb"), "rb") as fd:
        content_bytes = fd.read()
    content = content_bytes.decode()
    with pytest.raises(NothingChanged):
        format_file_contents(content, fast=True,
mode=JUPYTER_MODE)


def test_empty_string() -> None:
    with pytest.raises(NothingChanged):
        format_file_contents("", fast=True, mode=JUPYTER_MODE)


def test_unparseable_notebook() -> None:
    msg = (
        r"File
'tests[/\\]data[/\\]notebook_which_cant_be_parsed_oracle\.ipynb'
"
        r"cannot be parsed as valid Jupyter notebook\."
    )
    with pytest.raises(ValueError, match=msg):
        format_file_in_place(
            pathlib.Path("tests") /
"data/notebook_which_cant_be_parsed_oracle.ipynb",
            fast=True,
            mode=JUPYTER_MODE,
        )
```

After a bit of tuning and iteration, this should produce following text you can append to your Issue Description:

```
## Formatting files

Functions `format_file_contents` and `format_file_in_place`
```

should work on jupyter notebooks, which is controlled by the same `mode` parameter like `format_cell`.
We want to apply the same cell formatting, but also these file-level formatting rules:
- JSON normalization: Reformat entire notebook JSON with consistent 1-space indentation across all nested structures (cells, metadata, outputs, etc.)
- Trailing newline preservation: Output matches input's trailing newline status - if input file ends with \n, output does too; if input has no trailing newline, output omits it
- Metadata handling: Accept and properly format notebooks with empty metadata {} or fully populated metadata (interpreter, kernelspec, language_*info)*
- *Language detection: Skip formatting entirely for non-Python notebooks (detected via language*_info metadata or kernelspec) - raise NothingChanged
- Invalid JSON handling: Raise ValueError with descriptive message for notebooks that can't be parsed as valid JSON
- Empty file handling: Raise NothingChanged for empty string input
- No-op detection: Raise NothingChanged if notebook is already properly formatted (both JSON structure and Python code in cells are correct)

Formatting via the CLI

Make sure that the CLI with the `--diff` parameter on a jupyter notebook does this:
- With changes: Display unified diff showing Python code changes within cells (e.g., 'foo' → "foo"), not JSON structure changes. The example output should include a diff like this: `"@@ -1,3 +1,3 @@\n %%time\n \n-print('foo')\n+print(\"foo\")\n"`.
- Without changes: Report "1 file would be left unchanged" when notebook is already properly formatted

```
  Also include a new `--ipynb` CLI option:
  - `--ipynb flag`: Force notebook formatting for files with
  non-standard extensions (not .ipynb)
  - Mutual exclusivity: --pyi and --ipynb flags cannot be used
  together - CLI should exit with an error: "Cannot pass both
  `pyi` and `ipynb` flags!\n"
```

Complex Requirements: Jupyter Dependencies

The next set of tests are pretty complex. The tests are using monkey patching, where they essentially replace "jupyter_dependencies_are_installed()" with a static function that just returns True or False.

This approach is needed for the tests, but it's not really needed for the functionality to work. It's also quite complicated to explain in words how this should work. We'll start by using Claude Code to summarize the expected behavior:

```
Can you summarize the expected behavior here:

def test_cache_isnt_written_if_no_jupyter_deps_single(
    monkeypatch: MonkeyPatch, tmpdir: local
) -> None:
    # Check that the cache isn't written to if Jupyter
dependencies aren't installed.
    jupyter_dependencies_are_installed.cache_clear()
    nb = os.path.join("tests", "data",
"notebook_trailing_newline_oracle.ipynb")
    tmp_nb = tmpdir / "notebook.ipynb"
    with open(nb) as src, open(tmp_nb, "w") as dst:
        dst.write(src.read())
    monkeypatch.setattr(
        "black.jupyter_dependencies_are_installed", lambda
verbose, quiet: False
    )
    result = runner.invoke(main, [str(tmpdir /
"notebook.ipynb")])
```

```
    assert "No Python files are present to be formatted. Nothing
to do" in result.output
    jupyter_dependencies_are_installed.cache_clear()
    monkeypatch.setattr(
        "black.jupyter_dependencies_are_installed", lambda
verbose, quiet: True
    )
    result = runner.invoke(main, [str(tmpdir /
"notebook.ipynb")])
    assert "reformatted" in result.output


def test_cache_isnt_written_if_no_jupyter_deps_dir(
    monkeypatch: MonkeyPatch, tmpdir: local
) -> None:
    # Check that the cache isn't written to if Jupyter
dependencies aren't installed.
    jupyter_dependencies_are_installed.cache_clear()
    nb = os.path.join("tests", "data",
"notebook_trailing_newline_oracle.ipynb")
    tmp_nb = tmpdir / "notebook.ipynb"
    with open(nb) as src, open(tmp_nb, "w") as dst:
        dst.write(src.read())
    monkeypatch.setattr(
        "black.files.jupyter_dependencies_are_installed", lambda
verbose, quiet: False
    )
    result = runner.invoke(main, [str(tmpdir)])
    assert "No Python files are present to be formatted. Nothing
to do" in result.output
    jupyter_dependencies_are_installed.cache_clear()
    monkeypatch.setattr(
        "black.files.jupyter_dependencies_are_installed", lambda
verbose, quiet: True
    )
```

```
    result = runner.invoke(main, [str(tmpdir)])
    assert "reformatted" in result.output
```

However, since it's really difficult to explain the behavior of jupyter_dependencies_are_installed() and since this is not part of the core solution, but it's a requirement for the testing framework used, we are just going to give this full function in the Issue Description. We'll also explain in detail what is the expected behavior that's controlled by this function, and where exactly it needs to be plugged in.

```
## Jupyter dependencies

To facilitate testing, please add this function to the
`black.handle_ipynb_magics` module:

from functools import lru_cache

@lru_cache()
def jupyter_dependencies_are_installed(*, verbose: bool,
quiet: bool) -> bool:
    try:
        import IPython  # noqa:F401
        import tokenize_rt  # noqa:F401
    except ModuleNotFoundError:
        if verbose or not quiet:
            msg = (
                "Skipping .ipynb files as Jupyter
dependencies are not installed.\n"
                "You can fix this by running ``pip install
black[jupyter]``"
            )
            out(msg)
        return False
    else:
        return True
```

Also export this function as `black.jupyter_dependencies_are_installed` and also `black.files.jupyter_dependencies_are_installed` so it can be used directly with these types of import as well.

Calling `jupyter_dependencies_are_installed.cache_clear()` should reset the state so we can simulate a different scenarios as above, ie when dependencies are installed (monkeypatch returning True) and not installed (monkeypatch returning False). When Jupyter dependencies are not installed, jupyter files are excluded from any reformatting and this alternative message from above should be displayed.

For this to work, you'll need to modify `black.get_sources()` to exclude `.ipynb` if the `jupyter_dependencies_are_installed(verbose=verbose, quiet=quiet)` returns false.

We also need these new parameters `black.gen_python_files(..., verbose=False, quiet=False)` that are then passed to `jupyter_dependencies_are_installed` specifically for .ipynb files. You also need to add these new parameters to `black.get_sources()`.

This should lead to graceful degradation following this logic:
- Dependency detection: Before processing .ipynb files, check if required Jupyter dependencies (ipython, tokenize_*rt, etc.) are installed via jupyter*_dependencies_*are*_installed() function given above
- Missing dependencies behavior:
  - .ipynb files are completely skipped/ignored as if they don't exist
  - Output message: "No Python files are present to be formatted. Nothing to do"
  - No formatting attempted, no errors raised

```
- Cache behavior:
  - When dependencies are missing, cache is NOT written for
.ipynb files
  - Prevents marking notebooks as "processed" when they were
actually skipped
  - Ensures notebooks will be re-checked if dependencies are
installed later
- With dependencies available:
  - .ipynb files are processed normally
  - Output shows "reformatted" for changed notebooks
  - Cache is written as usual
- Applies to both modes: Works identically whether
formatting a single .ipynb file directly or scanning a
directory containing notebooks
```

Alternative, we could've just removed these tests from the suite, but that would be a worse solution. These tests are testing important behavior of the solution, so we should always try to retain as many tests as possible, to ensure our testing framework is as comprehensive as possible.

Final Issue Description with Regression Tests

We've now covered all the tests in tests/test_ipynb_oracle.py. We also need to add a small section "No regressions" at the bottom, where we also explain where the repository is, that the environment has already been set up, how to run tests, etc. You should always include information on environment setup and how to run tests:

```
# Issue: Add Jupyter Notebook Formatting Support to Black

## Overview

Add support for formatting Jupyter notebook (`.ipynb`) files
in Black, preserving IPython magic commands and Jupyter-
specific behaviors while applying Black's formatting rules
to Python code in cells.
```

Expected behavior

Here is a minimal example of the new expected behavior

```python
from black import (
    NothingChanged,
    format_cell,
    format_file_contents,
    format_file_in_place,
)

# new mode for jupyter notebooks
JUPYTER_MODE = Mode(is_ipynb=True)

# Already correctly formatted should raise `NothingChanged`
src = 'foo = "a"'
format_cell(src, fast=True, mode=JUPYTER_MODE)   # should raise `NothingChanged` as it's already correctly formatted

# Reformatting should work the same as in other places in black
src = 'foo = "a" ;'
result = format_cell(src, fast=fast, mode=JUPYTER_MODE)
assert results == 'foo = "a";'   # reformated code
```

Cell formatting

The cell formatting function should have this signature:

```
def format_cell(src: str, *, fast: bool, mode: Mode) -> str:
```

The expected behavior is that:

```
- if cell has trailing semicolon, remove it;
- if cell has IPython magics, mask them;
- format cell;
- reinstate IPython magics;
- reinstate trailing semicolon (if originally present);
- strip trailing newlines.

Cells with syntax errors will not be processed, as they
could potentially be automagics or multi-line magics, which
should not be supported yet.

The `fast` parameter is True, it should skip the stability
and equivalence steps we would normally perform.

When nothing is changed, the function should raise the
`NothingChanged` exception.

### Semicolon handling

Semicolons should only be formatted if:
1. They have spaces before them that need removal, OR
2. The code itself needs formatting

If the code is already well-formatted and the semicolon has
no space before it, raise NothingChanged even if a semicolon
is present.

So for example, spaces before trailing semicolons are
removed, but before comments preserved, so `foo = "a" ;  #
bar` should become `foo = "a";  # bar`. And this input
`import black;\n\n# this is a comment` should not be changed
and should raise NothingChanged.

### Magic handling

In general we want to preserve IPython magic syntax and
```

format Python code that follows magics:
- Cell magics (%%time, %%bash) stay at the start; Python code after them is formatted
- Line magics (%, !, ?, ??) are preserved; code on the same line or after is formatted
- Non-Python cell magics (%%bash, %%html) skip formatting entirely
- Automagics (magic commands without % prefix), multiline magics (with \), and invalid Python after magics trigger no changes
- Already-transformed magics (e.g., get_ipython().run_cell_magic()) are left alone
- Nested cell magics are both preserved while Python code is formatted

So, for example:
- `ls =!ls` should be formatted as `ls = !ls`
- `%%t -n1\nls = !ls` should not be reformatted (should raise NothingChanged)
- `%time 1 + \\\n2` should not be reformatted (should raise NothingChanged)
-

Formatting files

Functions `format_file_contents` and `format_file_in_place` should work on jupyter notebooks, which is controlled by the same `mode` parameter like `format_cell`.
We want to apply the same cell formatting, but also these file-level formatting rules:
- JSON normalization: Reformat entire notebook JSON with consistent 1-space indentation across all nested structures (cells, metadata, outputs, etc.)
- Trailing newline preservation: Output matches input's trailing newline status - if input file ends with \n, output does too; if input has no trailing newline, output omits it

- Metadata handling: Accept and properly format notebooks with empty metadata {} or fully populated metadata (interpreter, kernelspec, language_*info)*
- *Language detection: Skip formatting entirely for non-Python notebooks (detected via language*_info metadata or kernelspec) - raise NothingChanged
- Invalid JSON handling: Raise ValueError with descriptive message for notebooks that can't be parsed as valid JSON
- Empty file handling: Raise NothingChanged for empty string input
- No-op detection: Raise NothingChanged if notebook is already properly formatted (both JSON structure and Python code in cells are correct)

Formatting via the CLI

Make sure that the CLI with the `--diff` parameter on a jupyter notebook does this:
- With changes: Display unified diff showing Python code changes within cells (e.g., 'foo' → "foo"), not JSON structure changes. The example output should include a diff like this: `"@@ -1,3 +1,3 @@\n %%time\n \n-print('foo')\n+print(\"foo\")\n"`. Also ensure that each cell has a trailing newline before diff-ing, and diff each cell separately and that `@@ -1,3 +1,3 @@` and a header are both on separate lines. You can achieve this by using the `black.output.diff()` function.
- Without changes: Report "1 file would be left unchanged" when notebook is already properly formatted

Also include a new `--ipynb` CLI option:
- `--ipynb flag`: Force notebook formatting for files with non-standard extensions (not .ipynb)
- Mutual exclusivity: --pyi and --ipynb flags cannot be used together - CLI should exit with an error: "Cannot pass both `pyi` and `ipynb` flags!\n"

Jupyter dependencies

To facilitate testing, please add this function to the `black.handle_ipynb_magics` module:

```python
from functools import lru_cache

@lru_cache()
def jupyter_dependencies_are_installed(*, verbose: bool, quiet: bool) -> bool:
    try:
        import IPython  # noqa:F401
        import tokenize_rt  # noqa:F401
    except ModuleNotFoundError:
        if verbose or not quiet:
            msg = (
                "Skipping .ipynb files as Jupyter dependencies are not installed.\n"
                "You can fix this by running ``pip install black[jupyter]``"
            )
            out(msg)
        return False
    else:
        return True
```

Also export this function as `black.jupyter_dependencies_are_installed` and also `black.files.jupyter_dependencies_are_installed` so it can be used directly with these types of import as well.

We would use this for testing using these two imports:
```python
```

jupyter_dependencies_are_installed.cache_clear()
monkeypatch.setattr(
        "black.jupyter_dependencies_are_installed", lambda verbose, quiet: False
)
\`\`\`
and
\`\`\`python
jupyter_dependencies_are_installed.cache_clear()
monkeypatch.setattr(
        "black.files.jupyter_dependencies_are_installed", lambda verbose, quiet: False
)
\`\`\`

In other words, we should be able to monkeypatch directly in `black` to be able to monkeypatch single file inputs, and also in `black.files` to be able to monkeypatch when an input is a directory (because files need to be discovered first).

Calling `jupyter_dependencies_are_installed.cache_clear()` should reset the state so we can simulate a different scenarios as above, ie when dependencies are installed (monkeypatch returning True) and not installed (monkeypatch returning False). When Jupyter dependencies are not installed, jupyter files are excluded from any reformatting and this alternative message from above should be displayed.

For this to work, you'll need to modify `black.get_sources()` to exclude `.ipynb` if the `jupyter_dependencies_are_installed(verbose=verbose, quiet=quiet)` returns false.

We also need these new parameters `black.gen_python_files(..., verbose=False, quiet=False)`

that are then passed to `jupyter_dependencies_are_installed` specifically for .ipynb files. You also need to add these new parameters to `black.get_sources()`.

This should lead to graceful degradation following this logic:
- Dependency detection: Before processing .ipynb files, check if required Jupyter dependencies (ipython, tokenize_rt, etc.) are installed via jupyter_dependencies_are_installed() function given above
- Missing dependencies behavior:
  - .ipynb files are completely skipped/ignored as if they don't exist
  - Output message: "No Python files are present to be formatted. Nothing to do"
  - No formatting attempted, no errors raised
- Cache behavior:
  - When dependencies are missing, cache is NOT written for .ipynb files
  - Prevents marking notebooks as "processed" when they were actually skipped
  - Ensures notebooks will be re-checked if dependencies are installed later
- With dependencies available:
  - .ipynb files are processed normally
  - Output shows "reformatted" for changed notebooks
  - Cache is written as usual
- Applies to both modes: Works identically whether formatting a single .ipynb file directly or scanning a directory containing notebooks

No regressions

Make sure the existing functionality is not broken. You can run regression tests using `pytest`. The repository is at

```
`/workdir/black` with a `uv` venv already set up (you won't
need to install any new dependencies or modify the env in
any way).
```

Great, now it looks like we've covered all the aspects of expected behavior, and also explained the special jupyter_dependencies_are_installed() that's needed for testing. It looks like we've provided enough information, so we can proceed to the next step.

Step 6: Writing Hints

So, to reiterate - the Issue Description needs to contain all the necessary and sufficient information to pass the tests.

Sometimes however the AI agent won't be able to solve the issue even when provided with all the information. This could be because it simply makes a mistake, loses track of its own work, it incorrectly tests their solution, accidentally breaks something, gets confused, etc...

When this happen we still need to somehow verify that the issue is solvable. This is why we have **Hints**.

The idea of hints is to provide **optional but useful information** to the AI agent. This information needs to be strictly optional because if it's not optional, it should be in the Issue Description.

Hints can be quite prescriptive. They can tell the AI agent where to look in the codebase, what approach to take, or even provide code snippets. The key requirement is that the hints should not resolve any ambiguity - they should only help the agent to solve an issue that's already fully specified in the Issue Description.

Step 7: 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.

Step 8: Review the AI Agent Attempts

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.

Click on "Batch 1 (base)" to see how the AI agent solved this issue description. Look at the "attempt_1" in this batch that's almost correct but has 3 tests failing. 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
- **AI review (in depth)** - a special, more in-depth review of each of the test failures

Using AI Review (In Depth)

Let's start with looking at "AI review (in depth)".

This is an in-depth review of test failures. For this attempt it will be preloaded already. But in your annotation you'll need to load it on demand and it might take ~5 minutes to generate.

So, let's look at the analysis of the 3 failing tests.

- **test_magic** - The `?str` help magic isn't recognized because the regex only matches `%` prefixes and the fallback condition checks for `!` and `??` but not single `?`, causing NothingChanged to be raised instead of formatting the Python code that follows. This requirement was pretty clear in the issue description, so this is a genuine failure by the agent.
- **test_cell_magic_with_magic** - The `%%t` magic (a `%%time` alias) isn't in the hardcoded PYTHON_CELL_MAGICS whitelist, so it's treated as non-Python and returns unformatted, when it should recognize `%%t` contains Python code and format `ls =!ls` to `ls = !ls`. This requirement was also pretty clear in the issue description as the AI agent should know about this magic, so again the agent made a mistake.
- **test_cache_isnt_written_if_no_jupyter_deps_dir** - The monkeypatch logic seems to be unclear to the agent. This has been explained in the issue description, but the expected behavior is not entirely clear, as it works for one of the 2 similar tests. **Therefore, we probably want to make this is a bit more clear in the issue description.**

Next, let's look at "attempt_5" which has only 2 failed tests. Once again let's look at the "AI review (in depth)" tab. This is what we can see:

- **test_ipynb_diff_with_change** - The agent's `diff_for_ipynb()` function doesn't add trailing newlines (`+ "\n"`) to cell sources before diffing, causing "No newline at end of file" markers to appear in the output. Looking back at the issue description, adding newlines to cells was not mentioned, and it's not something that's obvious from other provided information. **Therefore, this is a problem with the issue description, and we need to fix it.**
- **test_ipynb_flag** - This test fails for the exact same reason as test_ipynb_diff_with_change; the `--ipynb` flag itself is correctly implemented with proper mutual exclusivity and mode handling, but the underlying diff formatting issue causes the output format mismatch.

By looking at a few more attempts, we'll see that there is a similar pattern of failures. Sometimes the AI agent correctly guesses the monkeypatch logic, and sometimes it correctly guesses the newlines, but neither of these have been fully explained.

Looking at other attempts we can also notice that especially the monkeypatch tests are frequently failing. So we should try to give some extra explanation and some more hints on that.

Alright, so we have some action items to do here - we need to improve the issue description and the hints.

For each attempt, you need to verify that:

- If tests passed, the solution is actually correct
- If tests failed, the solution is actually incorrect

This is critical to ensure zero false positives and zero false negatives.

Adjusting the Issue Description

So, let's modify the issue description based on examining the AI agent attempts. We'll add more information on the monkeypatch logic and also the newlines for cell diffs.

The changes we need to make:

1. **In the "Formatting via the CLI" section**, add: "Also ensure that each cell has a trailing newline before diff-ing, and diff each cell separately and that `@@ -1,3 +1,3 @@` and a header are both on separate lines. You can achieve this by using the `black.output.diff()` function."
2. **In the "Jupyter dependencies" section**, add a new paragraph explaining how monkeypatching will work:

```
We would use this for testing using these two imports:
\`\`\`python
jupyter_dependencies_are_installed.cache_clear()
monkeypatch.setattr(
        "black.jupyter_dependencies_are_installed", lambda
verbose, quiet: False
    )
\`\`\`
and
\`\`\`python
jupyter_dependencies_are_installed.cache_clear()
monkeypatch.setattr(
        "black.files.jupyter_dependencies_are_installed",
lambda verbose, quiet: False
    )
\`\`\`

In other words, we should be able to monkeypatch directly in
`black` to be able to monkeypatch single file inputs, and
also in `black.files` to be able to monkeypatch when an
input is a directory (because files need to be discovered
first).
```

For the hints, let's give even more information on how monkeypatches will be used:

```
- For the monkeypatch to work, we should be able to
`jupyter_dependencies_are_installed.cache_clear()`, followed by
`monkeypatch.setattr("black.jupyter_dependencies_are_installed",
lambda verbose, quiet: False)`, then invoke the runner with
`result = runner.invoke(main, [str(tmpdir / "notebook.ipynb")])`
and this should simulate a situation where dependencies are not
installed, and therefore the notebook is not reformatted.
- If we then do
`jupyter_dependencies_are_installed.cache_clear()` and
`monkeypatch.setattr("black.jupyter_dependencies_are_installed",
lambda verbose, quiet: True)`, we should then be able to run the
runner with `result = runner.invoke(main, [str(tmpdir /
"notebook.ipynb")])` and this will reformat the notebook.
- The same strategy should also work on directory inputs, e.g.
`monkeypatch.setattr("black.files.jupyter_dependencies_are_installed",
lambda verbose, quiet: False)` followed by `result =
runner.invoke(main, [str(tmpdir)])` should produce a "No Python
files are present to be formatted. Nothing to do" message, while
doing it
`monkeypatch.setattr("black.files.jupyter_dependencies_are_installed",
lambda verbose, quiet: True)` and `result = runner.invoke(main,
[str(tmpdir)])` should reformat.
```

This information is not necessary (as it should be inferrable from the new paragraph we added to the Issue Description), but is still helpful. Therefore, it provides a nice reasoning shortcut for the AI agent.

And in the explanation textbox:

```
I provided additional information on how monkey patching is
expected to work. This is inferrable from the Issue Description,
but it makes it just super clear to the agent how exactly this
is used.
```

Once you've updated the Issue Description, Hints and the explanation, click on "Start new agent run". Again you'll see the results right away, because they are cached in this tutorial.

Reviewing the Final AI Agent Attempts

Now the pass rates look better. For base we get 10% pass, and for the key hints we also get 10%. Since the base pass rate is >0% we can ignore the hinted version (hinted version is only used when the base pass rate is 0%).

This now confirms that the issue is solvable based on our Issue Description, so that's good!

We'll now need to review all the attempts again.

Let's start by looking at attempt_3 which passes all the tests. Once again let's look at the AI review (in-depth). As we can see, all tests pass, but the AI agent takes a very different approach. For this project this is OK, although it's not ideal (the PR's approach with using the ipython library is probably a much better solution).

Now click on on the "Your explanation" tab. Here we'll write our explanation for this attempt. We need to confirm that this is not a false positive and not a false negative. Let's write this explanation:

```
All tests pass, but the AI agent takes a fundamentally different
approach. The AI agent takes a regexp based approach instead of
using the `tokenize_rt` directly. This produces the correct
behavior, as measured by the test suite, although the
implementation can be considered a bit more "hacky".

The requirement to use tokenize_rt was hinted at in the Issue
Description, but it's not a strict requirement based on the
issue description. Therefore, it's acceptable for the AI agent
to use this alternative solution. The solution also passes a
comprehensive suite of ~150 regression tests, meaning that it
does not break any important functionality.

Based on this, we can conclude that this is a true positive (and
not a false positive). It's a valid solution that implements all
the required new functionality without breaking existing
functionality.
```

When you write these explanation, please make sure they are standalone and self-sufficient. I.e. do not reference the AI review, or anything else in the annotation.

Now, let's look at attempt_1.

Once again, let's look at the "AI review (in-depth)". As we can see, two tests fail, but both of them look like genuine implementation failures of something that was clearly explained in the Issue Description.

In the "Your explanation" tab write this explanation:

```
Two tests fail:
- test_trailing_semicolon_with_comment_on_next_line - this exact
example with expected output was quoted verbatim in the Issue
Description. Despite this, the AI agent solution is incorrect.
Therefore, this is a genuine implementation mistake from the
agent, and not an ambiguity.
- test_cache_isnt_written_if_no_jupyter_deps_dir - this test
fails despite extensive explanations in the issue description.
In particular, the issue description specifies that the
monkeypatch will be used with
`black.files.jupyter_dependencies_are_installed`, but this does
not seem to work with the AI agent's solution. Therefore, once
again, this was a clear requirement that just wasn't fulfilled
by the agent.

Both of these are genuine mistakes by the agent, and were clear
requirements in the issue description. Therefore, this is a true
negative.
```

While you write these explanations you'll notice that the AI review is not always accurate. It sometimes provides alternative explanations and wrong suggestions. Therefore, you should not entirely rely on it, but always critically look at the output, look back at the issue description and ensure everything is correct.

The "Your explanation" explanation **needs to be written by you, and not an LLM**. If we suspect uncritical LLM usage, your annotation will be rejected and

you will be removed from the project.

This is extremely important and this is really the main part of the task where we need your expertise to make AI Agents better.

When you do the annotation, you'll need to write explanations for all 10 attempts (or 20 if you run both base and hinted). Each time reproducing the issue in the VM, taking a screenshot and writing an explanation of your verdict.

Common Issues to Watch For

After reviewing the attempts, you may find issues:

- **Tests fail but solution seems correct** (false negative) - This means something tested is not mentioned in the issue description. You need to add this to the issue description.
- **Tests pass but solution seems incorrect** (false positive) - This means either your tests are incomplete, or you mentioned something in the issue description that's not actually tested. You need to either add more tests or remove the requirement from the issue description.

If you find issues, adjust the issue description and run the agent again.

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**.

Not 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.

Using AI Effectively

When working on this project, here are recommendations for where to use AI:

- **Environment setup**. Use Claude Code. It's good in solving complex environment issues that can otherwise take hours to debug.
- **Splitting the PR into Fix Patch and Test Patch**. Use Claude Code. This is usually a mundane task of renaming files and references to files, that Claude Code is pretty good at.
- **Tuning the Test Suite**. Do manually. You should run the tests yourself with/without the Fix Patch, decide which tests to include in the regression suite, remove any broken tests etc. It's important to do this carefully.
- **Writing the Issue Description**. Do manually with AI help. You can use Claude/Claude Code to help you understand the tests, summarize requirements, but you should always verify everything yourself and make sure it's aligned with tests.
- **Writing hints**. Do manually. If you see repeated mistakes that the AI agent makes in many attempt, and the requirement is clearly explained in the Issue Description, then this is a strong candidate for a hint. You can make hints pretty prescriptive and even give code chunks, as long as you are certain that the behavior is already well explained in the Issue Description.

Using the Final runner

The 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.

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

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][#attempt-status] section to understand what the different

status codes mean for attempts.

3. Test Patch cannot be applied ("Incomplete" error)

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

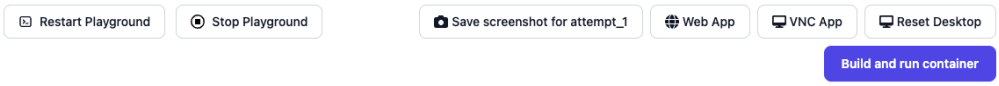
If you have a different error you cannot understand, please email us on `support@joinstellar.ai` or post in the forum (please just don't put the link to your annotation) with the screenshot of the issue and we'll try to assist as soon as possible.

Tips

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.

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.