Annotation guidance for debug nodes

Context

Once more we are working on MLE-bench. Now, we have LLM-generated code blocks that throw an error. We would like annotators to write a short description of what the bug and the fix is, and then fix the bug.

Task information:

- Each buggy code block comes from attempting to solve an MLE-bench task (identified in the metadata we provide).
- Annotators may view the actual task description at https://github.com/openai/mle-bench/tree/main/mlebench/competitions. Note that `description.md` has a direct paste of the Kaggle competition text, but the MLE-bench modified competition might be slightly different, in which case the MLE-bench modified version takes precedence.
- To get the data, implementers can install the mle-bench package, and run `mlebench prepare -c <competition-id>`. This will pull down data to `~/.cache/mle-bench/data`. This may be needed to replicate and fix the bug.
- To verify submission validity, implementers can run `python src/dojo/grade_code.py <path_to_your_python_file> <kaggle-competition-name> <output_logging_dir>/results.jsonl` and should provide the results of running this script on their submission for verification. **Client will provide an apptainer image and grading code. Annotators will use this to verify code is valid, and Client will use the same container and entry point.**

Client provides

A parquet file, consisting of fields:

- id: unique id of this generated code block.
- plan: model's plan to tackle the competition. This will only be a partial description as it could come from the middle of an agentic search trace, but may be useful to understand the code
- competition_id: which MLE-bench competition this code submission is for.
- analysis: the model's analysis of what the bug was. This may or may not be correct.
- code: the buggy code that the model wrote
- _term_out: the output that the code generated, including a traceback.
- exec_time: how long code execution took on our system. This is capped at 300s on our end.

Annotation flow

- First, annotators should verify the code is actually trying to solve the MLE-Bench competition:
    - Examples of code that does not do this: any sort of incomplete code (e.g. imports only, or code truncated before what looks like a complete block); natural language description of a plan without code (poor instruction-following).
    - Basically, if “debug this code block to generate a valid submission that maintains the original intent of the code” is not a possible instruction, reject the task and move on.
    - No gotchas here: assume that if there is complete code generating a submission.csv, it is targeting the correct task, etc.
- Next, annotators should replicate the bug in Client’s provided container.
    - If the code runs successfully without a bug, reject and move on to the next task..
    - If the code generates a different error than the one in the metadata and the annotator is sure this bug is not due to their environment (e.g. incorrect data paths / package imports, etc) mark this as the new buggy output, and continue this task attempting to resolve the new bug.
    - If the code runs for 6 minutes without throwing an error, it is considered not buggy. Most code will throw much faster so it should be obvious when the bug is fixed.
- Annotators should mark whether the bug is fixable.
    - Example non-fixable bugs:
        - Code attempts to load a model from the internet, but the container restricts internet access.
        - Code attempts to import a library / function / class that is hallucinated, **and it’s not obvious what the correct library should be.** If it’s clear from the rest of the context what the correct one should be (or there is a non-hallucinated library with the same functionality), annotators should consider this a fixable bug and replace it with a library that performs the right computations.
        - Code errors doing something that cannot be fixed without completely changing what the code does. For example, code densifies a sparse matrix and OOMs, but the rest of the algorithm requires a dense matrix and this can’t be worked around.
    - Example fixable bugs:
        - Incorrect arguments to a function.
        - Incorrect data loading calls.
        - Indexing errors of various kinds.
        - Many more.
- If the bug is not fixable, annotators should write a short description of why this is a dead end (this is entered in the `revised_analysis` field in the deliverable.
- Annotators should write a description of the bug’s cause, and a fix plan. The provided LLM-generated `analysis` can be used to bootstrap this but is not guaranteed to be correct. Likewise, the LLM-generated `plan` can be used but is not guaranteed to be correct.

    - Examples:
        - **Analysis**: “The code throws a `TypeError` because the instantiation of sklearn to `Pipeline` on line 17 includes a `input_transforms` parameter, which is not in the call signature.”
          Fix Plan: “To fix, I will replace the argument with the correct one, `transform_input`".
        - **Analysis**: “The code throws a `JSONDecodeError` because it is attempting to load a json lines (`.jsonl`) file as plain json.
          Fix plan: To fix, I will use the `jsonlines` package to load the file correctly.”
- **Annotators should additionally use our CWM tracing tool to generate an execution trace, and use it to write further analysis and a fix using this trace. Description of this tool and how to annotate with it is in the “Guidance for tracing code” tab in this document. This analysis will go in the `cwm_analysis` field in the deliverable.**
- Annotators should fix the root cause of the bug, making a minimal change to the code needed to do so. Annotators should not make other improvements to the code, stylistic or otherwise.
    - For example: If there is an inefficient for loop that can be vectorized and also an IndexError: fix the IndexError but leave the inefficient parts. However, if there is an inefficient loop causing an OOM which is causing the code to throw, that is the bug and should be fixed.
- To verify, annotators should run the fixed code in the container. We assume that any code which does not error out in 6 minutes of execution is correct.
- If the reported bug is fixed but a new bug is introduced, annotators should treat this as a new bug annotation (write a natural language description of the cause and fix plan, and provide fixed code). For such multi-step bugfixes, annotators should record the step index so we can reconstruct the order.
- Annotators should take care not to introduce new bugs while they fix existing ones.

Deliverable format

- Tabular data (parquet preferred)
- Columns included:
    - All rows in the source data
    - bug_confirmed: boolean
    - bug_fixed: boolean
    - revised_analysis: analysis of the bug’s cause (as above)
    - revised_plan: revised bugfix plan (as above)
    - **cwm_analysis**: analysis using the CWM code tracing tool.
    - debug_step: step in the debugging chain (this starts at 0 in fixing the first bug. If other bugs remain in the same code block, increment this, and add another row to the deliverable).

Technical Requirements

To ensure seamless integration with our provided Jupyter Apptainer environment, all annotation submissions must adhere to the following technical guidelines.

1. **Data Paths:** the apptainer launch exposes prepared MLE-bench competition data to ./data/train.csv and ./data/test.csv. The bug in LLM-generated code could be attempting to access data in a different location.
2. **External Libraries and Internet Access:** the environment operates without internet access. Therefore, scripts cannot download or install external libraries (e.g., Resnet50 with imagenet weights). Bugs caused by this should be marked and rejected.
3. **Output Submission File Name:** All Python scripts should generate a single submission CSV file named submission.csv. Any other file names will result in processing errors. These can be fixed.
4. **GPU Usage:** The code should be runnable on 1 GPU, 12 CPUs and 120GB memory and not throw an error for 6 minutes. If it runs longer than this, it is considered valid.

We provide a Python tracing utility that prints the intermediate variables of Python code.

It is provided as a self-contained library, totally independent from the Docker and the rest of the code and tools provided in this annotation workstream.

Assuming we are inside the directory with the tracing utility, these are the instructions to run it:

```
python -m venv venv
source venv/bin/activate
python -m pip install -r requirements.txt
python main.py # -> traces the code in "code_to_trace.py" using ">>
START_OF_TRACE"  as the tracing starting point
```

Code_to_trace.py looks like:

```
def f(d, k):
    for key, val in d.items():
        if key < k:
            new_d[key] = val
    return new_d
def main(): # << START_OF_TRACE
    return f({1: 2, 2: 4, 3: 3}, 3)
```

This marks where the tracing starts: `# << START_OF_TRACE`

This example should print (after some logs):

<|frame_sep|><|call_sep|>{}<|action_sep|>def main(): # << START_OF_TRACE

<|frame_sep|><|line_sep|>{}<|action_sep|>   return f({1: 2, 2: 4, 3: 3}, 3)
<|frame_sep|><|call_sep|>{"d": "{1: 2, 2: 4, 3: 3}", "k": "3"}<|action_sep|>def f(d, k):
<|frame_sep|><|line_sep|>{"d": "{1: 2, 2: 4, 3: 3}", "k": "3"}<|action_sep|>   new_d = {}
<|frame_sep|><|line_sep|>{"d": "{1: 2, 2: 4, 3: 3}", "k": "3", "new_d": "{}"}<|action_sep|>   for key, val in d.items():
<|frame_sep|><|line_sep|>{"d": "{1: 2, 2: 4, 3: 3}", "k": "3", "new_d": "{}", "key": "1", "val": "2"}<|action_sep|>   if key < k:
<|frame_sep|><|line_sep|>{"d": "{1: 2, 2: 4, 3: 3}", "k": "3", "new_d": "{}", "key": "1", "val": "2"}<|action_sep|>
new_d[key] = val
<|frame_sep|><|line_sep|>{"d": "{1: 2, 2: 4, 3: 3}", "k": "3", "new_d": "{1: 2}", "key": "1", "val": "2"}<|action_sep|>   for key, val in d.items():
<|frame_sep|><|line_sep|>{"d": "{1: 2, 2: 4, 3: 3}", "k": "3", "new_d": "{1: 2}", "key": "2", "val": "4"}<|action_sep|>   if key < k:
<|frame_sep|><|line_sep|>{"d": "{1: 2, 2: 4, 3: 3}", "k": "3", "new_d": "{1: 2}", "key": "2", "val": "4"}<|action_sep|>
new_d[key] = val
<|frame_sep|><|line_sep|>{"d": "{1: 2, 2: 4, 3: 3}", "k": "3", "new_d": "{1: 2, 2: 4}", "key": "2", "val": "4"}<|action_sep|>   for key, val in d.items():
<|frame_sep|><|line_sep|>{"d": "{1: 2, 2: 4, 3: 3}", "k": "3", "new_d": "{1: 2, 2: 4}", "key": "3", "val": "3"}<|action_sep|>   if key < k:
<|frame_sep|><|line_sep|>{"d": "{1: 2, 2: 4, 3: 3}", "k": "3", "new_d": "{1: 2, 2: 4}", "key": "3", "val": "3"}<|action_sep|>   for key, val in d.items():
<|frame_sep|><|line_sep|>{"d": "{1: 2, 2: 4, 3: 3}", "k": "3", "new_d": "{1: 2, 2: 4}", "key": "3", "val": "3"}<|action_sep|>   return new_d
<|frame_sep|><|return_sep|><|action_sep|>   return new_d
<|arg_sep|>"{1: 2, 2: 4}"<|frame_sep|><|return_sep|><|action_sep|>   return f({1: 2, 2: 4, 3: 3}, 3)
<|arg_sep|>"{1: 2, 2: 4}"<|frame_sep|>

This contains the local variable values at each time step. Note that the tracing should also work in case the code throws an exception. For example:

```
def f(d, k):
    new_d = {}
    for key, val in d.items():
        if key < k:
            new_d[key] = val
        new_d[key] += 1
    return new_d
def main(): # << START_OF_TRACE
    return f({1: 2, 2: 4, 3: 3}, 3)
```

<|frame_sep|><|call_sep|>{}<|action_sep|>def main(): # << START_OF_TRACE
<|frame_sep|><|line_sep|>{}<|action_sep|>   return f({1: 2, 2: 4, 3: 3}, 3)
<|frame_sep|><|call_sep|>{"d": "{1: 2, 2: 4, 3: 3}", "k": "3"}<|action_sep|>def f(d, k):
<|frame_sep|><|line_sep|>{"d": "{1: 2, 2: 4, 3: 3}", "k": "3"}<|action_sep|>   new_d = {}
<|frame_sep|><|line_sep|>{"d": "{1: 2, 2: 4, 3: 3}", "k": "3", "new_d": "{}"}<|action_sep|>   for key, val in d.items():
<|frame_sep|><|line_sep|>{"d": "{1: 2, 2: 4, 3: 3}", "k": "3", "new_d": "{}", "key": "1", "val": "2"}<|action_sep|>   if key < k:
<|frame_sep|><|line_sep|>{"d": "{1: 2, 2: 4, 3: 3}", "k": "3", "new_d": "{}", "key": "1", "val": "2"}<|action_sep|>
new_d[key] = val
<|frame_sep|><|line_sep|>{"d": "{1: 2, 2: 4, 3: 3}", "k": "3", "new_d": "{1: 2}", "key": "1", "val": "2"}<|action_sep|>
new_d[key] += 1
<|frame_sep|><|line_sep|>{"d": "{1: 2, 2: 4, 3: 3}", "k": "3", "new_d": "{1: 3}", "key": "1", "val": "2"}<|action_sep|>   for key, val in d.items():
<|frame_sep|><|line_sep|>{"d": "{1: 2, 2: 4, 3: 3}", "k": "3", "new_d": "{1: 3}", "key": "2", "val": "4"}<|action_sep|>   if key < k:
<|frame_sep|><|line_sep|>{"d": "{1: 2, 2: 4, 3: 3}", "k": "3", "new_d": "{1: 3}", "key": "2", "val": "4"}<|action_sep|>
new_d[key] = val
<|frame_sep|><|line_sep|>{"d": "{1: 2, 2: 4, 3: 3}", "k": "3", "new_d": "{1: 3, 2: 4}", "key": "2", "val": "4"}<|action_sep|>
new_d[key] += 1
<|frame_sep|><|line_sep|>{"d": "{1: 2, 2: 4, 3: 3}", "k": "3", "new_d": "{1: 3, 2: 5}", "key": "2", "val": "4"}<|action_sep|>   for key, val in d.items():

<|frame_sep|><|line_sep|>{"d": "{1: 2, 2: 4, 3: 3}", "k": "3", "new_d": "{1: 3, 2: 5}", "key": "3", "val": "3"}<|action_sep|> if key < k:
<|frame_sep|><|line_sep|>{"d": "{1: 2, 2: 4, 3: 3}", "k": "3", "new_d": "{1: 3, 2: 5}", "key": "3", "val": "3"}<|action_sep|> new_d[key] += 1
<|frame_sep|><|exception_sep|><|action_sep|> new_d[key] += 1
<|arg_sep|>"(<class 'KeyError'>, KeyError(3), <traceback object at 0x10f0384c0>)"<|frame_sep|><|return_sep|><|action_sep|> new_d[key] += 1
<|arg_sep|>"None"<|frame_sep|><|exception_sep|><|action_sep|> return f({1: 2, 2: 4, 3: 3}, 3)
<|arg_sep|>"(<class 'KeyError'>, KeyError(3), <traceback object at 0x10f038840>)"<|frame_sep|><|return_sep|><|action_sep|> return f({1: 2, 2: 4, 3: 3},

Annotators are asked to run this tool on the buggy code. Then, use the obtained trace as part of their reasoning to describe how and why the code is incorrect. They should cherry-pick/select the relevant parts (according to their judgement) of the tracing and interleave it with the reasoning. They are allowed to skip lines of the traces, but if they pick a line, they should keep the trace line as is, not changing the original format.

For example, using the example from above that led to an exception, this could be the reasoning:

```
Let’s analyze the code.

We call f with arguments f({1: 2, 2: 4, 3: 3}, 3):

<|frame_sep|><|call_sep|>{"d": "{1: 2, 2: 4, 3: 3}", "k": "3"}<|action_sep|>def f(d, k):

We initialize new_d:

<|frame_sep|><|line_sep|>{"d": "{1: 2, 2: 4, 3: 3}", "k": "3"}<|action_sep|>    new_d = {}

We iterate over d, and in each iteration we update new_d only if key < k, we also increment new_d::

<|frame_sep|><|line_sep|>{"d": "{1: 2, 2: 4, 3: 3}", "k": "3", "new_d": "{}"}<|action_sep|>  for key, val in d.items():
<|frame_sep|><|line_sep|>{"d": "{1: 2, 2: 4, 3: 3}", "k": "3", "new_d": "{}", "key": "1", "val": "2"}<|action_sep|>        if key < k:
<|frame_sep|><|line_sep|>{"d": "{1: 2, 2: 4, 3: 3}", "k": "3", "new_d": "{}", "key": "1", "val": "2"}<|action_sep|>            new_d[key] = val
<|frame_sep|><|line_sep|>{"d": "{1: 2, 2: 4, 3: 3}", "k": "3", "new_d": "{1: 2}", "key": "1", "val": "2"}<|action_sep|>        new_d[key] += 1

But this can cause an exception, if `key` is not already present in new_d:

<|frame_sep|><|line_sep|>{"d": "{1: 2, 2: 4, 3: 3}", "k": "3", "new_d": "{1: 3, 2: 5}", "key": "3", "val": "3"}<|action_sep|>        new_d[key] += 1
<|frame_sep|><|exception_sep|><|action_sep|>        new_d[key] += 1
<|arg_sep|>"(<class 'KeyError'>, KeyError(3), <traceback object at 0x10f0384c0>)"<|frame_sep|><|return_sep|><|action_sep|>        new_d[key] += 1

This is causing the exception. `new_d[key] += 1` should be inside the if block:

```

```
        if key < k:
            new_d[key] = val
            new_d[key] += 1

```

Let's verify this fixes the issue. We call f with arguments f({1: 2, 2: 4, 3: 3}, 3):

We initialize new_d:

<|frame_sep|><|line_sep|>{"d": "{1: 2, 2: 4, 3: 3}", "k": "3"}<|action_sep|>   new_d = {}

We iterate over d, and in each iteration we update new_d only if key < k, we also increment new_d, but now inside the `if` condition:

<|frame_sep|><|line_sep|>{"d": "{1: 2, 2: 4, 3: 3}", "k": "3", "new_d": "{}"}<|action_sep|>
for key, val in d.items():
<|frame_sep|><|line_sep|>{"d": "{1: 2, 2: 4, 3: 3}", "k": "3", "new_d": "{}", "key": "1",
"val": "2"}<|action_sep|>   if key < k:
<|frame_sep|><|line_sep|>{"d": "{1: 2, 2: 4, 3: 3}", "k": "3", "new_d": "{}", "key": "1",
"val": "2"}<|action_sep|>   new_d[key] = val
<|frame_sep|><|line_sep|>{"d": "{1: 2, 2: 4, 3: 3}", "k": "3", "new_d": "{1: 2}", "key": "1",
"val": "2"}<|action_sep|>   new_d[key] += 1
<|frame_sep|><|line_sep|>{"d": "{1: 2, 2: 4, 3: 3}", "k": "3", "new_d": "{1: 3}", "key": "1",
"val": "2"}<|action_sep|>   for key, val in d.items():

Yes, it worked. new_d[1] correctly incremented from 2 to 3. How does the execution continue?

|frame_sep|><|line_sep|>{"d": "{1: 2, 2: 4, 3: 3}", "k": "3", "new_d": "{1: 3}", "key": "2",
"val": "4"}<|action_sep|>   if key < k:
<|frame_sep|><|line_sep|>{"d": "{1: 2, 2: 4, 3: 3}", "k": "3", "new_d": "{1: 3}", "key": "2",
"val": "4"}<|action_sep|>   new_d[key] = val
<|frame_sep|><|line_sep|>{"d": "{1: 2, 2: 4, 3: 3}", "k": "3", "new_d": "{1: 3, 2: 4}", "key":
"2", "val": "4"}<|action_sep|>   new_d[key] += 1
<|frame_sep|><|line_sep|>{"d": "{1: 2, 2: 4, 3: 3}", "k": "3", "new_d": "{1: 3, 2: 5}", "key":
"2", "val": "4"}<|action_sep|>   for key, val in d.items():

Yes, it is updating correctly. At the end, it will return:

<|arg_sep|>"{1: 3, 2: 5}"<|frame_sep|>

The desired value. So the corrected code should be:

```
def f(d, k):
    new_d = {}
    for key, val in d.items():
        if key < k:
            new_d[key] = val
            new_d[key] += 1
    return new_d
def main(): # << START_OF_TRACE
    return f({1: 2, 2: 4, 3: 3}, 3)
```

```
```