Milestone 3 - ML4SE - Group-B

Group Members


```
Ashwin Patil <anpatil2@illinois.edu>
Chinmay Saraf <csaraf2@illinois.edu>
Kedar Takwane <takwane2@illinois.edu>
Maahi Patel <maahidp2@illinois.edu>
```


Repository Link

- Main Repository: https://github.com/theashwin/ml4se
- Milestone-3 Code and Data: https://github.com/theashwin/ml4se/tree/main/milestone-3
- Main Dataset Used: https://github.com/github/CodeSearchNet

Introduction

In this project, we're evaluating ChatGPT's effectiveness to reason about a code snippet's execution with different control and dataflow, localization, and detection of bugs for a given function. We use Python and Java repositories from the CodeSearchNet corpus. We use OpenAI API to interact with ChatGPT and then we engineer prompts to elicit the required outputs from the model. We finally evaluate the generated reasoning, tests, and methods by setting up and running them. We present our observation and results overview in this report.

Methodology

1. **Data Collection** - We selected 6 repositories from CodeSearchNet : 3 each for Python and Java (repositories listed further down). From these 6 repositories, we selected 50 functions for Python and Java each. Limiting the source repositories to 6 helps in testing the generated test cases as the project setup becomes relatively easier.
2. **Data Categorization** - To help us understand the accuracy and effectiveness of ChatGPT to solve the 3 tasks we categorized the functions into three categories based on the type of inputs, outputs, and the body of the function by whether it's primitive or non-primitive. *We concede there might be better ways to categorize the dataset but these are quite good approximations.*
    a. Easy - If the used variables are of primitive datatypes.
    b. Medium - If the used variables contain both primitive and non-primitive datatypes.
    c. Hard - If the used variables are of non-primitive datatypes.
3. **Prompt Engineering** - Using **random** data points from the dataset we used multiple prompts to elicit the required response from the model. **For a better understanding, the prompts are further categorized by tasks.**
4. **Automating Runs -** Once the dynamic prompts were finalized we ran these prompts for 50 functions each for Python and Java. Outputs can be found at `milestone-3/out`
    a. **Dynamic** - We've simulated back-and-forth conversations with ChatGPT by breaking down the task into parts which are known to help the model understand the task better and elicit better responses.
    b. **Markdown** - As the output given by ChatGPT is in markdown format (which was quite an interesting observation), we save the response from the model in `.md` format which is visually pleasing and a better way to present our work.
5. **Generating Code with Changed Data Flow and Control Flow and their Reasoning** -
    a. **Generation of two variants** - We generated two variants of a function - data flow changed and control flow changed through ChatGPT

   b. **Reasoning generated variant -** After the generation of the variants, we use prompts from task-1 of Milestone-2 to get their reasoning.

6. **Bug Localization and Detection of mutated functions -** As a part of task-2, we are generating 5 mutants of both Java and Python functions using PIT rules.
   a. **Generating 5 mutants** - We have referred to PIT (https://pitest.org/quickstart/mutators/) rule to apply mutation on the functions. We have created 5 mutants for each of the 25 functions for both Java and Python. The mutated function is stored in the same data object with new key-value pair as `mutants: ["mutant1", "mutant2", "mutant3", "mutant4", "mutant5"]`
   b. **Bug Localization** - For each of the generated mutants for a function, we ask ChatGPT to localize the bug in it. If ChatGPT fails to localize the bug, we ask ChatGPT the same question with the updated prompt.
   c. **Bug Detection -** If ChatGPT succeeds in localizing the bug, we ask ChatGPT to detect the bug.
7. **Report Generation -** With everything else out of the way we generate this report, to sum up, our observations and give statistics on what works and what does not and for what cases the model struggle, and in what part it excels.

Important Links

1. Data - https://github.com/theashwin/ml4se/tree/main/milestone-3/prompt/json - `java.json` and `python.json`
2. Multi-Turn Prompts - https://github.com/theashwin/ml4se/tree/main/milestone-3/prompt/json - `bug-localization.json` and `data-control-mutation.json`
   a. Init = Initial Prompt
   b. 1…N = N-Turn Prompts
3. Final Output - https://github.com/theashwin/ml4se/tree/main/milestone-3/out
4. Automation Script Code - The main script which is executing all prompts is gpt.py. This script is calling:
   a. data_control_mutation.py - This script executes to generate two variants of a code - with changed data flow and with changed control flow
   b. reasoning.py - This script executes the reasoning prompt for both variants of code
   c. bug_localization.py - This script executes the bug localization and detection on the 5 mutants of each function.

Design Decisions

Markdown Output

We realized that the output that OpenAI API returns are in the markdown format so instead of storing the output in a textual format our output is in markdown format. **This part is also autogenerated so needs no manual effort.** This has many benefits like,

1. The output is displayed exactly like it was intended by ChatGPT
2. We can manipulate the text file to highlight the important parts or add additional information to the file to make the report per function.
3. It is much easier to read and understand. For instance, tables are actually shown in a tabular format instead of text with dashes.

Tasks are Independent

To avoid wasting up the context space which is already limited in transformer-based models. We decided to treat each one of the tasks as **independent.** That means that after each task for a particular function, the context is cleaned up. This allows the tasks to be run independently but on the downside, the context (for instance, the method under test, etc.) needs to be provided again to the model for each task.

Corner Case - No Return Statement

There were a few cases where there are no return statements at all. In this case, we ask for the behavior of the function itself and move away from the input-output type of checking which we do in other cases.

Corner Case - No Input Params

We've also handled cases where there are no input parameters in the method under test. In this case, we've added another field in the dataset to include variables of interest that will be passed to the model alongside the code itself. The intuition behind this is to give the model as much information as possible and the variables we've chosen are global variables that potentially can take up any value.

Installation

Requirements

The one and only requirement of this project is a pip package named `openai` which provides convenient access to the OpenAI API from applications written in the Python language. It includes a pre-defined set of classes for API resources that initialize themselves dynamically from API responses which makes it compatible with a wide range of versions of the OpenAI API.

```
pip install --upgrade openai
```

This would require an OpenAI API key which can be obtained from `https://platform.openai.com/account/api-keys`. Save the key in a new file named `config.json` in `milestone-3` folder in the format given below.

```
{
    "OPENAI_API_KEY": "<KEY>"
}
```

Execute

After you've added `config.json` with the required OpenAI API. To run the code you need to execute `gpt.py` with optional parameters `--lang=python/java` by default the language is considered as `Java` and `--n=50` or `--number=50` by default the number of functions is `5` (As it takes quite a long time, due to API limits). **Running this file will run all three tasks for n code snippets for that particular language Generated outputs can be found in `milestone-3/out/task_<task_no>/<lang>/#.md` Where # is the serial number of the function.**

> 💡 Unlike last time we choose to make the tasks separate for ease of evaluation

```
gpt.py --lang=python --n=5 #For Python

gpt.py --lang=java --n=5 #For Java
```

Data

The dataset can be found in `milestone-3/prompt/json/java.json` or `milestone-3/prompt/json/python.json` Content of the JSON is as follows:

Java

	Java	Python
Easy	18	32
Medium	15	12
Hard	17	7
Total	**50**	**51**

Java Example

```
{
    "func_id": 1,
    "project_repo": "https://github.com/Esri/geometry-api-java",
    "location": "src/main/java/com/esri/core/geometry/Bufferer.java",
    "function_name": "progress_",
    "function": "private void progress_() {
                \n\t\tm_progress_counter++;\n\t\tif
                (m_progress_counter % 1024 == 0) {\n\t\t\tif ((m_progress_tracker
                != null)\n\t\t\t\t\t\t&& !(m_progress_tracker.progress(-1, -1)))
                \n\t\t\t\tthrow new RuntimeException(\"user_canceled\");\n\t\t}\n\t}",
    "non_primitive_data_objects": "",
    "primitive_data_objects": "",
    "input": "m_progress_counter",
    "expected_output": "",
    "flag": "hard",
    "mutants": [
      "private void progress_() {\r\n\t\tm_progress_counter++;\r\n\t\tif (m_progress_counter % 1024 != 0) {\r\n\t\t\tif ((m_progress_tracke
      "private void progress_() {\r\n\t\tm_progress_counter++;\r\n\t\tif (m_progress_counter % 1024 != 0) {\r\n\t\t\tif ((m_progress_tracke
      "private void progress_() {\r\n\t\tm_progress_counter++;\r\n\t\tif (m_progress_counter % 1024 == 0) {\r\n\t\t\tif ((m_progress_tracke
      "private void progress_() {\n\t\tm_progress_counter++;\n\t\tif (true) {\n\t\t\tif ((m_progress_tracker == null)\n\t\t\t\t\t\t&& !(m_pro
      "private void progress_() {\n\t\tm_progress_counter++;\n\t\tif (m_progress_counter % 1024 == 0) {\n\t\t\tif (true\n\t\t\t\t\t\t&& !(m_p
    ]
  }
```


Python Example


```
"func_id": 2,
      "project_repo": "https://github.com/limix/numpy-sugar/blob/4bdfa26913135c76ef3cd542a332f4e5861e948b/numpy_sugar/linalg",
      "location": "numpy-sugar/numpy_sugar/linalg/cho.py",
      "function_name": "cho_solve",
      "function": "from numpy import asarray, empty\n\n\ndef cho_solve(L, b):\n    from scipy.linalg import cho_solve as sp_cho_solve\n\n
L = asarray(L, float)\n    b = asarray(b, float)\n    if L.size == 0:\n        if b.size != 0:\n            raise ValueError(\"Dimension
 mismatch between L and b.\")\n        return empty(b.shape)\n    return sp_cho_solve((L, True), b, check_finite=False)",
      "non_primitive_data_objects": "",
      "primitive_data_objects": "",
      "input": "",
      "expected_output": "",
      "flag": "hard",
      "mutants": [
        "from numpy import asarray, empty\n\n\ndef cho_solve(L, b):\n    from scipy.linalg import cho_solve as sp_cho_solve\n\n    L = as
array(L, float)\n    b = asarray(b, float)\n    if L.size == 0:\n        if b.size == 0:\n            raise ValueError(\"Dimension mismat
ch between L and b.\")\n        return empty(b.shape)\n    return sp_cho_solve((L, True), b, check_finite=False)",
        "from numpy import asarray, empty\n\n\ndef cho_solve(L, b):\n    from scipy.linalg import cho_solve as sp_cho_solve\n\n    L = as
array(L, float)\n    b = asarray(b, float)\n    if L.size == 0:\n        if b.size > 0:\n            raise ValueError(\"Dimension mismatc
h between L and b.\")\n        return empty(b.shape)\n    return sp_cho_solve((L, True), b, check_finite=False)",
        "from numpy import asarray, empty\n\n\ndef cho_solve(L, b):\n    from scipy.linalg import cho_solve as sp_cho_solve\n\n    L = as
array(L, float)\n    b = asarray(b, float)\n    if L.size == 0:\n        if b.size >= 0:\n            raise ValueError(\"Dimension mismat
ch between L and b.\")\n        return empty(b.shape)\n    return sp_cho_solve((L, True), b, check_finite=False)",
        "from numpy import asarray, empty\n\n\ndef cho_solve(L, b):\n    from scipy.linalg import cho_solve as sp_cho_solve\n\n    L = as
array(L, float)\n    b = asarray(b, float)\n    if L.size == 0:\n        if b.size < 0:\n            raise ValueError(\"Dimension mismatc
h between L and b.\")\n        return empty(b.shape)\n    return sp_cho_solve((L, True), b, check_finite=False)",
        "from numpy import asarray, empty\n\n\ndef cho_solve(L, b):\n    from scipy.linalg import cho_solve as sp_cho_solve\n\n    L = as
array(L, float)\n    b = asarray(b, float)\n    if L.size == 0:\n        if b.size <= 0:\n            raise ValueError(\"Dimension mismat
ch between L and b.\")\n        return empty(b.shape)\n    return sp_cho_solve((L, True), b, check_finite=False)"
      ]
```


Projects Setup

We set up a total of 6 projects: 3 Python and 3 Java projects to test the execution of generated tests and semantically equivalent methods. The projects are listed below:

Java

1. `https://github.com/Esri/geometry-api-java`
2. `https://github.com/locationtech/jts`
3. `https://github.com/looly/hutool`

Python

1. `https://github.com/limix/numpy-sugar/`
2. `https://github.com/scipy/scipy`
3. `https://github.com/numpy/`

Observations

Prompt Engineering

Listed below are the part of prompts that we've included/excluded from our final prompt and our reasoning/intuition behind why they might or might not work and why there are important to the overall prompt. There are four sections below: three task-specific sections and one general section which relates to the formatting of the responses and prompts.

Legend

🛠️ Final/Best Prompt Used

✅ Successful Prompt

⚠️ Moderately Successful Prompt

⛔ Unsuccessful Prompt

Task 1.1: Generate Mutations

🛠️ **Prompt 1: For the java function with comments given in the variable 'code' below, generate two versions of the function. First, change the data flow of the function. Second, change the control flow of the function. It is fine to change the semantics of the function. Follow the instructions carefully. Instructions: Be as creative as possible. Return the generated functions.**

code: <code>

Response 1: <response>

Prompt 2: Write the data flow mutated function in <data> </data> tags and control flow mutated function in <control> </control> tag

1️⃣ **"For the <LANG> function with comments given in the variable 'code' below,"**

This part of the prompt defines the task at hand along with the programming language used. For our use case, the languages will be java/python. The reason behind adding "with comments" is that we think this helps the model consider the comments to write the explanation for any given input.

2️⃣ “**generate two versions of the function. First, change the data flow of the function.**”

This part of the prompt defines the task at hand along with the programming language used. For our use case, the languages will be java/python. The reason behind adding “with comments” is that we think this helps the model consider the comments to write the explanation for any given input.

3️⃣ “**Second, change the control flow of the function.”**

This part of the prompt defines the task at hand along with the programming language used. For our use case, the languages will be java/python. The reason behind adding “with comments” is that we think this helps the model consider the comments to write the explanation for any given input.

4️⃣ “**Follow the instructions carefully. Instructions: Be as creative as possible.”**

This part of the prompt helps us list the things we **need** the response to follow in its entirety. Here, “Be as creative as possible” instructs the model to make the changes without semantic consideration. Without this prompt, the model generates semantic equivalent data and control mutations only.

Instead of coming up with manual ways to do this, we decided to test the limits of ChatGPT going one step beyond for the tasks it can or cannot perform. prima facie it looks like it can create mutations quite seamlessly.

5️⃣ “**List out all of the changes done to the original function.”**

This part of the prompt helps us evaluate the extent of the modifications done by ChatGPT and can be used as a metric describing how extensive the modification was from the original function.

6️⃣ **Show calculations and reason about its execution for each of the three input-output pairs.**

We’ve decided to send this prompt after we get a response from GPT giving us input-output pairs because according to the research we’ve seen during the course there were multiple instances where breaking down the prompt into smaller prompts helped the researchers to get the expected answer. This also helps the model as the model has a single deliverable after each major prompt. So, in the first case, it returns input-output pairs and after the second prompt, it reasons about its execution.

7️⃣ **It is fine to change the semantics of the function.**

We added this prompt to give provision to ChatGPT to make significant changes in function to generate different variants. We also observed that ChatGPT tries to preserve semantics with this prompt as well.

⛔ **Explain how the data flow and control flow of generated functions differs from the original function.**

Adding this line at the end of the prompt gave us the required explanation regarding the difference in data flow and control flow of generated functions compared to the original but it generated some non-required data in the response which resulted in a timeout of ChatGPT.

Task 1.2: Code Explanations of Data Flow and Control Flow Mutated Functions.

🛠️ **Prompt 1: For the <LANG> function with comments given in the variable 'code' below, write three inputs and outputs in a table and reason about the execution of the function. Follow the instructions carefully. Instructions: Assume the variable's contents as required, The generated response must have the following qualities: concise, stepwise, and specific.**

code: <code>

Response 1: <response>

Prompt 2: Show calculations and reasons for each of the three input-output pairs.

1️⃣ **"For the java function with comments given in the variable 'code' below…"**

This part of the prompt defines the task at hand along with the programming language used. For our use case, the languages will be java/python. The reason behind adding "with comments" is that we think this helps the model consider the comments to write the explanation for any given input.

2️⃣ "**write three inputs and outputs in a table and reason about the execution of the function."**

Instead of creating the input ourselves and asking the model for answers and then evaluating it's reasoning, we get the input from GPT this allows us to check for edge cases where the function might break. This also is a good way for us to evaluate GPT's effectiveness and understanding of the code.

One specific case we've noticed is that even if we specify "three" input-output pairs in the prompt sometimes the model generates two or even four such pairs. We know for a fact that GPT 3.5 i.e. ChatGPT is not that great with large numbers but seeing it ignore smaller numbers in the prompt is an interesting observation.

3️⃣ "**Follow the instructions carefully. Instructions: Assume the variable's contents as required, …"**

This part of the prompt helps us list the things we **need** the response to follow in its entirety. Here, "Assume variable's content as required" instructs the model to mock variables as required or instantiate it while coming up with an output for the model as class-level or system-level variables are not present at the function-level. Instead of coming up with manual mocks or stubs. We ask GPT to do it for us.

4️⃣ **The generated response must have the following qualities: concise, stepwise, and specific.**

This part of the prompt helps us tune the response to our required qualities like conciseness (instead of verbose), stepwise, and being as specific as possible (includes details). This is added to ensure the quality of the generated response.

5️⃣ **code: <code>**

This variable contains the method under test as it is from the repository. Without modifications.

Task 2: Bug Localization for PIT mutated functions.

🛠️ **Prompt 1: For the <LANG> function tell if it is buggy. Follow the instructions carefully. Instructions: The first word of your response must be `Yes` if the function is buggy, else `No`. Give a coherent explanation for your analysis. Also, if it is buggy then localize the bug in the function.**

code: <code>

Response 1: <response>

Prompt 2: Now consider this information about the function provided to you in the previous message. The function is buggy, now your task is to find the exact location of the bug.

1️⃣ "**For the <LANG> function tell if it is buggy. "**

This language-specific prompt helps to lay out the task at hand in a simple and precise manner.

We tried verbose prompts which worked no better than simpler versions.

2️⃣ "**Follow the instructions carefully. Instructions: The first word of your response must be `Yes` if the function is buggy, else `No`. "**

This prompt helps the model follow a few basic rules which outline the task at hand.

We noticed the more verbose the prompt more is the chance of that part getting skipped or missed during execution.

Thus in our conclusion precise and crisp is the name of the game.

3️⃣ "**Give a coherent explanation for your analysis. Also, if it is buggy then localize the bug in the function. "**

This prompt does two things, once tries to explain new/unseen generated code and asks the model to localize the error in the code. It's currently hit-and-miss. Explanation helps us understand the change and also GPT try to do the same,

⛔ **For the <LANG> function tell if it is buggy or correct. Follow the instructions carefully. Instructions: The first word of your response must be Yes if the function is buggy, else No**

We first tried this variant of the prompt to determine if the ChatGPT was able to determine if the function is buggy or not. And based on the reply `Yes` or `No` we wanted to decide if we want to ask it the second prompt, which explicitly told ChatGPT that it was buggy and then asked it to find the bug.
But the output was not as expected ChatGPT, we wanted the first word of the response to be `Yes` or `No` if it was buggy. ChatGPT would reply with `Yes` or `No` but at times it would say `Yes` it is correct and at times `Yes` it is buggy. So to get one type of response, `Yes` or `No` with respect to buggy function, we had to modify the prompt a little. Also, the idea behind prompting ChatGPT with the second prompt only if it wasn't able to detect the bug using the first one is a design choice. We think that if it is able to find the bug using the init prompt then there should not be a need to explicitly ask it to find the bug.

⛔ **For the <LANG> function tell if it is buggy. Follow the instructions carefully. Instructions: The first word of your response must be `Yes` if the function is buggy, else `No`.**

To avoid ambiguity in `Yes` or `No` for the correct buggy, we tried removing `or correct` from the prompt and then we were able to see that ChatGPT provides the response only in terms of whether the method under test is buggy or not. Thus, we were able to automate the second part where we only provided the second prompt only if it replied `No` in the init prompt. Moreover, we wanted ChatGPT to provide an explanation for why it thinks it is buggy or not and if it is buggy, then we wanted it to also localize it. So, we modified it in the following way to be used for our automated script.

❓ **Note: Prompt 2 is executed only if the output of Response 1 is `No` (Below is Prompt 2)**

4️⃣ **"Now consider this information about the function provided to you in the previous message. The function is buggy, now your task is to find the exact location of the bug."**

If the model thinks that the code is buggy then we run another prompt to understand how the model thinks when it comes to code execution. This prompt helped us understand what works and if it does not then understand why!

Formatting

🪜 **"Step by Step" / "Stepwise"**

When you provide a prompt to GPT and ask it to generate text "step by step," the model will use its language generation capabilities to provide a sequence of steps or instructions related to the prompt you provided. This ensures that the generated text is formatted nicely in the form of bullet points or numbered lists which makes it easier to read and concise. Instead of a verbose explanation in huge paragraphs. Our intuition is that it also forces the model to think about the solution in a stepwise manner.

📄 **"In a tabular manner"**

Similar to the prompt above, this part asks GPT to generate the output and format it into a markdown-compatible table format so that it's easier to read and concise. For instance, we use this part of the prompt when we ask the model to generate three varied inputs for the method under test.

⚠️ **"display the results using comma separated values"**

This prompt works sometimes and sometimes the results are not well formatted, so we choose to not use this prompt to format the response from GPT.

Results

Interesting Observations

1️⃣ **ChatGPT Hallucinates**

This is one of the most interesting observations we've made from the last milestone and we observed the same in this milestone. We've observed that ChatGPT hallucinates.

This in our opinion is worse than not knowing as it shows that ChatGPT has no idea if the data it is using is from the given location or not. So, we think ChatGPT is not able to visit links but instead tokenizes the link and finds something similar to the given link but not the actual info.

2️⃣ **Failing to give correct reasoning for the functions with changed Control Flow**

In some cases, the ChatGPT fails to give correct reasoning for the functions with changed Data Flow. But for the original function, the given reasoning is correct.

3️⃣ **Gives priority to preserving the semantics when changing Data or Control Flow**

While generating the variants of the original function with changed Data Flow and Control Flow, the ChatGPT gives priority to preserving the semantics. We have explicitly mentioned in the prompt that - it is fine to change semantics. Still, wherever possible, ChatGPT tends to preserve the semantics.

4️⃣ **ChatGPT succeeds in finding a bug for the NON_VOID_METHOD_CALLS type mutation**

In the NON_VOID_METHOD_CALLS type of mutation, the return type of a condition is replaced by its default value. For most of the cases where this mutation is applied, ChatGPT manages to localize and detect the exact place of mutation and possible bugs.

5️⃣ **Gives Import Error if unable to find any bug in Python**

When asked to localize the bug, ChatGPT gives import errors if it cannot find anything wrong with the code after the second prompt

7️⃣ **ChatGPT often asks for more context to localize and detect a bug**

If there is a call to another function then ChatGPT will ask for its implementation and mentions that there could be something wrong in the function but it cannot localize it until we give its implementation and the rest of the code is okay.

8️⃣ **ChatGPT is good at determining arithmetic bugs with meaningful identifiers**

For the functions with meaningful identifiers, the ChatGPT performs well in localizing and determining the bugs. It has performed well in suggesting solutions to such bugs.

Conclusion

This milestone aimed to perform Prompt Engineering using ChatGPT on more code-related tasks such as code reasoning on functions with changed data flow and control flow, bug localization, and detection. It used Python and Java functions from the

CodeSearchNet dataset, totaling around 50 data points for each language for task-1 and 25 functions each for task-2. The difficulty level of each data point was labeled based on its input parameters, i.e., primitive or non-primitive, and call to other methods.

For task-1, which involves the reasoning of functions with changed data flow and control flow, we used ChatGPT to determine the variants of functions with changed data flow and control flow. The generated variants are given to ChatGPT to do the reasoning with the same prompts which we finalized in milestone-2. Prompt engineering is done to finalize the prompt of generating the code variants. The results of these tasks showed promising results. The ChatGPT performed well though the control flow and data flow of functions are changed. As already observed, it sometimes fails to do the mathematical calculation correctly. We observed the same in this task.

For task-2, we generated the mutated variants of 25 functions for both Java and Python and passed them to ChatGPT with a prompt to localize and detect bugs in the mutated function. We followed the PIT rules while generating the mutants of these functions. We did prompt engineering to finalize prompts to localize and detect bugs. From the results of this task, we observed that for most of the time, ChatGPT fails to localize or detect bugs. But if we explicitly mention about the presence of a bug, then it has improved in localizing the bugs. Still in detecting bugs, it is not performing that well. It also uses identifiers as a context to localize the bugs. For a few mutants, ChatGPT even suggested the correct resolution to the bug.

Overall, the results showed the capability of ChatGPT in understanding code irrespective of its data flow and control flow. The result also highlights the need for improvement in the dataset with mathematical calculations. For bug localization and detection tasks, the ChatGPT still needs a lot of improvement.