A Appendices

A.1 MULocBench Statistics

Project and Issue Statistics. Figure 1 shows the issue counts of the 46 projects included in MULocBench. The dataset covers a wide range of project types and domains. For example, pandas and scikit-learn represent popular data processing and machine learning libraries, transformers and keras correspond to deep learning frameworks, while ansible and yt-dlp reflect automation and media processing tools. Smaller projects such as DeepSpeeck-V3 and grok-1 illustrate that MULocBench also includes niche and emerging repositories. This diversity highlights the benchmark's coverage of different programming scenarios, user communities, and issue characteristics.

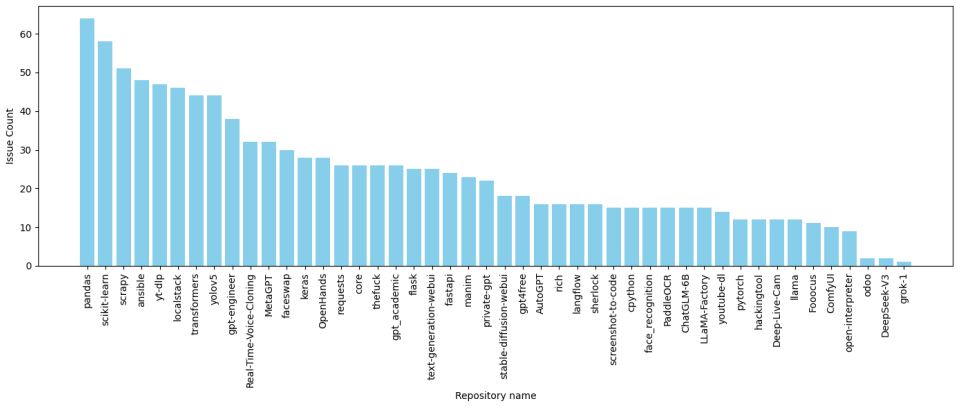

Figure 1: Statistics of issue counts across each projects.

File Type	Issue Number	File Number
Python	842	2494
OtherPL (JavaScript/C/Java)	16	53
Test	273	513
Documentation	259	443
Asset	48	59
Configuration	167	308

Table 1: Statistics of file types in the MULocBench.

File Type Distribution in MULocBench. Table 1 summarizes the distribution of file types in MULocBench. Among the 1,100 issues, although the projects are primarily Python-based, not all issue localizations involve Python code. This distribution indicates that issue localization tasks are not limited to Python source code but also span a diverse set of file types that play important roles in modern software projects.

Approximately 23.5% of issues correspond to locations that do not include any Python files. Specifically, 1.5% of issues involve files written in other programming languages (e.g., JavaScript, C, or Java), 24.8% involve test files, 23.5% involve documentation files, 4.4% involve asset files, and 15.2% involve configuration files.

From the file perspective, Python source files account for over 2,000 files. Each of the test, documentation, and configuration categories contains approximately 300–500 files. Files in other programming languages and asset files are less common, with each category containing slightly over 50 files.

A.2 Issue Characteristics

To understand how issue characteristics influence localization performance, we consider nine issue characteristics to analyze their relationship with localization performance [1–3]. Table 2 presents the descriptive statistics of the issue characteristics, including their mean, variance, minimum, and maximum values. The values of issue characteristics exhibit high variability, reflecting the diverse nature of issues and their potential impact on localization difficulty.

- **iss_des_length**: The number of words in the issue description. Longer descriptions generally provide more detailed information for developers to understand the problem context, interpret the symptoms, and identify potential root causes.
- **has_traceback**: Indicates whether the issue description contains an execution traceback. Tracebacks provide precise failure points, including file paths, line numbers, and call chains, which usually allow developers more quickly identify the root cause.

Table 2: Descriptive Statistics of Issue Features

Feature	Mean	Variance	Min	Max
iss_descr_length	615.45	2,692,121.36	4	37,442
has_traceback	0.21	0.17	0	1
has_code_block	0.50	0.25	0	1
has_inline_code	0.64	0.23	0	1
file_num	2.85	16.93	1	30
cls_num	2.11	35.48	0	68
func_num	2.80	56.46	0	114
line_num	22.09	8,511.70	0	1,291
diff_file_type_num	1.49	0.58	1	5

- **has_code_block**: Indicates whether the issue description contains a Markdown code block (“‘). Code blocks often include reproduction scripts, configuration snippets, error logs, or minimal examples. Such explicit technical content provides structured clues about the execution context, helping narrow down the related components or functions in the system.
- **has_inline_code**: Indicates whether the issue description contains inline code segments (enclosed in backticks, e.g., function name). Inline code is typically used to reference specific identifiers, such as variables, function names, file paths, or configuration keys. These references add specificity to the description, improving the alignment between the issue text and concrete code elements involved in localization.
- **file_num**: The number of files involved in issue localization. More files indicate a wider scope and greater structural complexity, making localization more challenging.
- **cls_num**: The number of classes involved in issue localization. More classes suggest cross-cutting changes and higher design complexity, increasing the difficulty of understanding and localizing the issue.
- **func_num**: The number of functions involved in issue localization. A larger number of functions implies multiple logical units are affected, making it harder to identify the root cause.
- **line_num**: The total number of lines changed involved in issue localization. Larger changes reflect greater issues complexity and increase the search space for locating the source of the issue.
- **diff_file_type_num**: The number of different file types modified (e.g., source code, tests, documentation). A higher number indicates that cross-cutting changes require diverse expertise, making the issue harder to locate.

Approach		iss_des_length	has_traceback	has_code_block	has_inline_code	file_num	cls_num	func_num	line_num	diff_file_num
BM25		0.0001*	-0.0423*	1.1124*	1.5827*	-0.4134*	-0.1683*	-0.239*	-0.0014*	-0.884*
Agentless	GPT-4o-mini	0.0001*	-0.1558*	-0.1712*	-0.0277*	-0.4276*	-0.1159*	-0.1033*	-0.0028*	-0.9445*
	Claude3.5	0.0002*	-0.008*	0.1011*	0.2063*	-0.803*	-0.2868*	-0.2503*	-0.0014*	-1.1176*
LocAgent	GPT-4o-mini	0.0001*	0.8571*	1.226*	2.2032*	0.0077*	-0.2445*	-0.2074*	-0.0135*	-0.9089*
	Claude3.5	-0.0002*	-0.0128*	0.5467*	0.2628*	-0.8365*	-0.2712*	-0.1331*	-0.0315*	-1.9413*
OpenHands	GPT-4o-mini	0.0001*	0.2253*	0.2338*	0.4942*	-0.6249*	-0.2258*	-0.2517*	-0.0225*	-0.8046*
	Claude3.5	-0.0002*	-0.0009*	-0.0752*	0.0771*	-1.0655*	-0.4096*	-0.3868*	-0.0142*	-1.6713*

Table 3: Correlation between issue complexity and issue localization in SOTA approaches(P-values < 0.05 are marked with *).

We assess the relationship between the issue characteristics and the issue localization by calculating their correlation coefficient. Given that the distribution of these characteristics exhibits skew- ness, and the localization result is binary (correct or not), logistic regression is employed for the analysis across three LLMs. Table 3 and Table 4 shows the correlation between issue characteristics and issue localization in SOTA approaches and LLM-prompting approaches.

Location-related features, especially the number of files and different file types, consistently show strong negative correlations across all LLMs, indicating that **issues involving more extensive locations are generally harder to localize**.

Textual features show weaker and more variable correlations. Issue description length has only a weak correlation (<0.01), while richer textual information (e.g., tracebacks) provides modest correlation overall. However, the effect of richer textual features varies across LLMs. For example, LocAgent with GPT-4o-mini benefits from tracebacks, whereas Claude 3.5 is negatively affected by them, showing negative correlations. Similarly, Location-Hint with Claude 3.5 benefits from code blocks, while GPT-4o-mini is negatively affected by code blocks.

Approach		iss_des_length	has_traceback	has_code_block	has_inline_code	file_num	cls_num	func_num	line_num	diff_file_num
ClosedBook	GPT-4o-mini	0.0001	0.3269	0.2364	0.2182	-2.1504*	-0.2016*	-0.2477*	-0.0244*	-4.4434*
	Claude3.5	0.0003*	0.3471*	0.3561*	0.3028*	-1.9109*	-0.269*	-0.2916*	-0.0316*	-3.6438*
ProStructure	GPT-4o-mini	0.0002*	0.2757*	-0.1109*	0.003*	-1.7122*	-0.1987*	-0.2417*	-0.0208*	-3.537*
	Claude3.5	0.0002*	0.2401*	0.0478*	0.0671*	-2.0467*	-0.3434*	-0.3431*	-0.0332*	-3.2742*
LocHint	GPT-4o-mini	0.0*	0.2775*	-0.2031*	-0.1733*	-1.8291*	-0.3247*	-0.297*	-0.0231*	-2.9825*
	Claude3.5	0.0001*	0.2551*	0.1672*	0.0854*	-1.3183*	-0.2462*	-0.2148*	-0.0219*	-1.7195*
Pipeline-ProStructure	GPT-4o-mini	0.0001*	0.2478*	-0.1216*	-0.0484*	-1.7954*	-0.1189*	-0.1578*	-0.0204*	-2.9983*
	Claude3.5	0.0002*	0.1686*	0.1394*	0.187*	-2.1249*	-0.2483*	-0.2624*	-0.0247*	-3.5681*
Pipeline-LocHint	GPT-4o-mini	0.0001*	0.2315*	-0.2248*	-0.0471*	-1.7547*	-0.224*	-0.2239*	-0.0188*	-2.9143*
	Claude3.5	0.0003*	0.1821*	0.2549*	0.286*	-2.1216*	-0.2834*	-0.2907*	-0.026*	-4.3645*

Table 4: Correlation between issue complexity and issue localization in LLM-prompting approaches(P-values < 0.05 are marked with *).

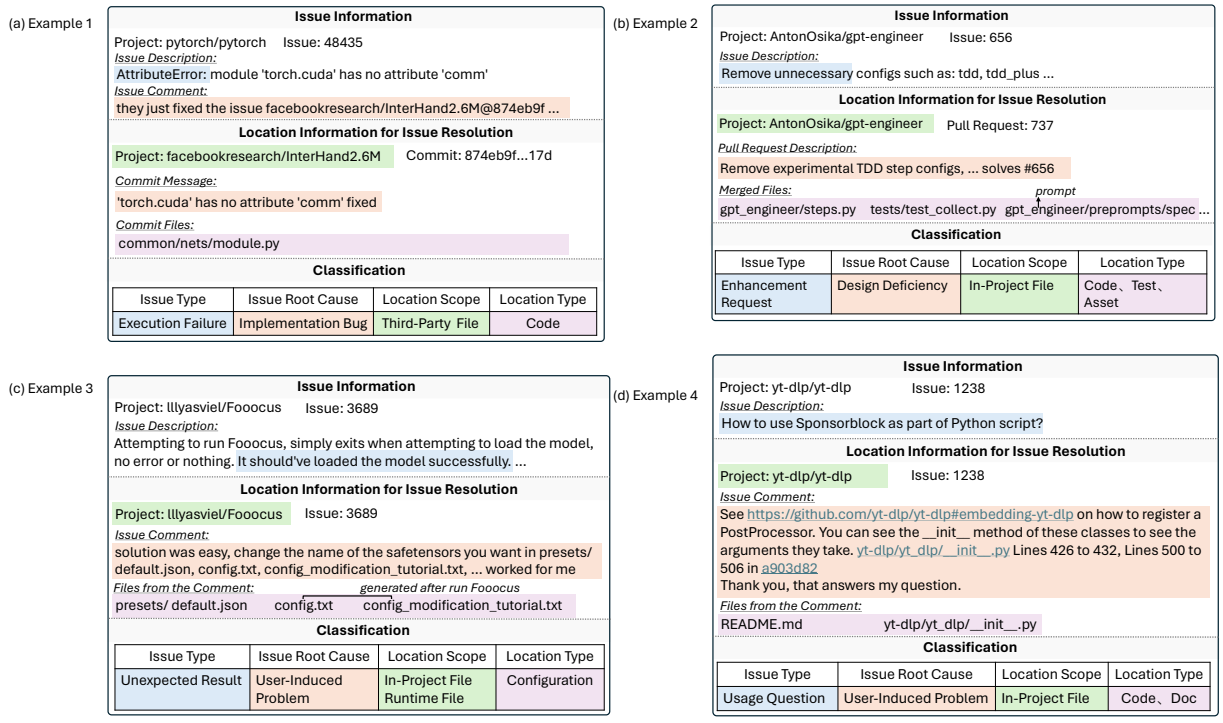

Figure 2: Four examples from MULocBench illustrating diverse issue types, causes, location scopes and types.

These variable patterns show that, **although richer textual information can aid localization, current LLMs cannot fully leverage these signals across complex issue contexts, limiting their effectiveness**.

A.3 Empirical Analysis

Issue Type captures the kinds of problem developers raise in issue reports, such as errors during code execution. It reflects what challenges developers encounter or what support they seek during project use and maintenance. We categorize issues into four types.

(1) Execution Failure: Issues where the project fails to run properly, such as installation errors, startup crashes, or runtime errors. This category accounts for 39.9% (439 issues). For example, in panel (a) of Figure 2, a developer reports an AttributeError: module 'torch.cuda' has no attribute 'comm' when executing code with the `pytorch` project.

(2) Unexpected Result: Issues where the project runs successfully, but its behavior or output deviates from expectations, such as producing incorrect results. It accounts for 23.7% (261 issues). For example, in panel (c) of Figure 2, a developer attempts to run `Fooocus` to load the model, but the program simply exits without loading it or showing any errors.

(3) **Enhancement Request**: Issues where users request new functionality, improvements to existing features, or extensions to the project's capabilities. It accounts for 25.1% (276 issues). For example, in panel (b) of Figure 2, a developer requests remove unnecessary configurations, such as tdd.
(4) **Usage Question**: Issues where users ask about project usage, such as how to accomplish specific tasks, locating files, or understanding code structure. It accounts for 11.3% (124 issues). For example, in panel (d) of Figure 2, a developer asks how to use `Sponsorblock` as part of Python script.

Root Causes of Issues capture the underlying reasons behind issue reports, such as code bugs. We categorize issue causes into three types:

(1) **Implementation Bug**: Problems arising due to errors in the project's implementation, where the functionality does not perform as expected, such as logic errors and incorrect outputs. It accounts for 34.5% (380 issues). For example, in panel (a) of Figure 2, a developer fixes an AttributeError raised by calling a non-existent API `torch.cuda.comm.broadcast`.
(2) **Design Deficiency**: Situations where a project is incomplete or poorly implemented, such as missing functionality and need for refactoring. It accounts for 36.3% (399 issues). For example, in panel (b) of Figure 2, a developer removes unnecessary configurations to improve design clarity and maintainability.
(3) **User-Induced Problem**: Problems arising from incorrect usage, misunderstandings, or limited knowledge of the project on the user's side. It accounts for 29.2% (321 issues). For example, in panel (c) of Figure 2, developers fail to load a model in `Fooocus` until they correct their own configuration settings.

Location Scope for Issue Resolution refers to the range of files involved in issue resolution, which can also include locations beyond the project repository. To use a software project, developers depends not only on files within the project itself, but also on external dependencies, runtime-generated files, and user-authored code. We classify location scopes into four categories:

(1) **In-Project File**: Files that are physically present within the project repository. Typical examples include source code files (`main.py`), configuration files (`settings.yaml`), or documentation (`README.md`). It accounts for 94.1% (1035 issues). For example, in panel (b) of Figure 2, an issue from the `AntonOsika/gpt-engineer` project was resolved by modifying three files within the repository, which shows a case of in-project file localization.
(2) **Runtime File**: Files that are outside the project but are generated during execution or required at runtime. It accounts for 2.2% (24 issues). For example, a generated file created after running a source script, or an environment-specific configuration file such as `.env`. In panel (c) of Figure 2, an issue from the `lllyasviel/Fooocus` project falls into this category: its resolution required modifying two files, `config.txt` and `config_modification_tutorial.txt`, which are not part of the repository but are created after execution.
(3) **Third-Party File**: Files from external projects, such as third-party libraries that the current project depends on. It accounts for 2.1% (23 issues). For example, in panel (a) of Figure 2, an issue reported in the `pytorch/pytorch` project is ultimately traced to an implementation bug in the external dependency `facebookresearch/InterHand2.6M`, rather than in PyTorch itself.
(4) **User-Authored File**: Files written by users outside the original project, typically referenced in issue descriptions when reporting problems. It accounts for 3.0% (33 issues). For example, in issue 78759 of the `ansible/ansible` project, a user reports an error when passing a variable to a loop and provides the corresponding configuration. Another user suggests to replace `loop:{{sysctl.values}}` with `loop:{{sysctl['values']}}`, which resolves the issue. In this case, the location of the issue resolution is the user's own configuration file, specifically the line `loop:{{sysctl.values}}`.

Location Type for Issue Resolution refers to the file types affected during issue resolution. A software project is not solely code; it also depends on configurations, documentation, and other artifacts that together to function as a complete project. We categorize issue locations into five types based on their functional roles.

(1) **Code**: Source files that implement the core logic and functionality of the project, typically written in programming languages such as `.py` or `.cpp`. It accounts for 80.8% (889 issues). For example, in panel (a) of Figure 2, the resolution was located in `common/nets/module.py`, which is a Python source file.
(2) **Test**: Test files that verify the correctness of the project through test cases. This category includes unit, integration, and system test files, e.g., `test_*.py` and `*.spec.js`. It accounts for 23.5% (258 issues). For example, in panel (b) of Figure 2, the issue resolution location involves the file `tests/test_collect.py`, which is a Python test file.
(3) **Configuration**: Configuration files used for build processes, dependency management, or runtime settings, such as `requirements.txt` and `pom.xml`. It accounts for 15.2% (167 issues). For example, in panel (c) of Figure 2, the issue resolution location contains the file `presets/default.json`, which is a configuration file.
(4) **Document**: Documentation files intended to describe and support the usage and maintenance of the project, typically found in files such as `README` or within the `docs` directory. It accounts for 23.5% (259 issues). For example, in panel (d) of Figure 2, the issue resolution location involves `README.md`, which provides guidance for developers on using the project.
(5) **Asset**: Non-code resources that support system operation or enhance functionality, such as data files (e.g., `.csv`, `.json`), static resources (e.g., images, fonts), or auxiliary scripts. It accounts for 4.4% (48 issues). For example, in panel (b) of Figure 2, the issue resolution location involves `gpt_engineer/preprompts/spec`. As this file contains only prompt text consumed as input, rather than executable code, it is classified as an asset.

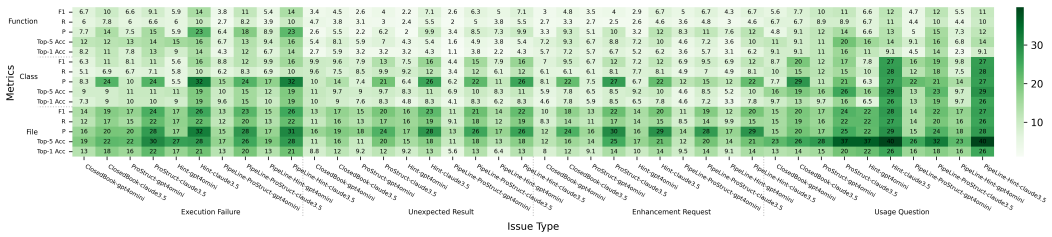

Figure 3: Performance comparison of different issue types on full MULocBench.

Figure 4: Performance comparison of different issue reasons on full MULocBench.

A.4 Figures of Performance comparison of different issue types, reasons, location scopes and types on Full MULocBench

Peformance Comparison across Issue Types. Figure 3 presents the performance comparison across different issue types on the full MULocBench. The performance differences observed on Python code closely mirror those on the full benchmark. **Execution Failures** achieve moderate results, with Location-Hint reaching 28% Acc@5 and 26% F1 at the file level. Their performance is relatively balanced across location levels (file, class, function) and metrics (Acc@1, Acc@5, precision, recall, F1), typically yielding mid-range scores: stronger than Enhancement Requests and Unexpected Results but weaker than Usage Questions, indicating a moderate level of localization difficulty. **Unexpected Results and Enhancement Requests** yield the weakest performance, with Acc@5 capped at around 20% at the file level. At the class level, Acc@5 and F1 are around 10% and 15%, while at the function level both fall below 10%, reflecting the challenge of pinpointing subtle or underspecified behavioral issues. **Usage Questions** achieve the strongest overall performance, with Location-Hint reaching 40% Acc@5 and 28% F1 at the file level. At the class level, Acc@5 and F1 are 29% and 27%, and at the function level both remain around 10%. Their relatively higher performance likely due to clearer textual cues in issue descriptions that make localization easier.

Peformance Comparison across Issue Reasons. Figure 4 shows the performance comparison of different issue reasons on the full MULocBench. **Implementation Bug** achieve the most stable performance, with Location-Hint reaching 23% Acc@5 and 26% F1 at the file level, and 12–15% at finer levels. Although bug fixing is generally considered highly challenging, the relatively balanced performance across levels, together with the long-standing emphasis of researchers on bug-fix scenarios, explains why this issue reason performs more consistently than others. **Design Deficiencies** perform weakest, with best file-level results of 20% Acc@5 and 21% F1, but Acc@5 and F1 at the class and function levels drop to 5.4%–16%. Notably, precision is obviously higher than recall across all three levels. Across all levels, precision is higher than recall, as design-related issues are abstract and cross-cutting: they are harder to locate, but once identified, predictions tend to be precise. **User-Induced Problems** show intermediate performance, with 33% Acc@5 and 20% F1 at the file level, but only 7%–13% at the class and function levels. This suggests that user mistakes often provide clear textual cues that aid file-level localization. However, such cues are less informative for finer-grained levels, which results in weaker class- and function-level performance.

A.5 A Qualitative Failure Analysis of Issue Localization on MULocBench

We provide five qualitative analyses of wrong issue localization from state-of-the-art approaches or LLMs.

A.6 Prompt Templates of LLM-based Approaches

Figure 10, Figure 11, Figure 12, Figure 13, Figure 14, Figure 15, Figure 16 shows the prompt templates of LLM-based approaches.

Issue Description *pytorch/pytorch/issues/48435*

AttributeError: **module 'torch.cuda' has no attribute 'comm**

I'm using torch 1.7.0, and get this kind of error

my torch is installed via pip install torch==1.7.0+cu101 torchvision==0.8.1+cu101 torchaudio===0.7.0
…

LLM's Response

GroundTruth
Project: InterHand2.6M
File: common/nets/module.py
Function: soft_argmax_1d
Line: 43

Closed-Book GPT-4omini ✖
Project: pytorch
File: **torch/cuda**/__init__.py
Line: 1
Line: 2
...

File: **torch/cuda/comm**.py
Line: 10
...

Pro-Structure GPT-4omini ✖
Project: pytorch
File: **torch/cuda**/__init__.py
Line: 1

Project: pytorch
File: **torch/cuda**/comm.py
Line: 1
...

Hint Claude3.5 ✖
Project: pytorch
File: **torch**/csrc/**cuda**/comm.cpp
File:
torch/csrc/**cuda**/Module.cpp
...
File:
torch/cuda/__init__.py
..

Pipeline-ProStructure Claude3.5 ✖
Project: pytorch
File: **torch/cuda**/__init__.py
Line: 1

Project: pytorch
File: **torch/cuda**/comm.py
Line: 1
…

Pipeline-Hint GPT-4o-mini ✖
Project: pytorch
File: **torch**/csrc/**cuda**/comm.cpp
Line: 1
Line: 2
..
project: pytorch
file: **torch**/csrc/**cuda**/comm.h

Explanation

Word-Matched Misleading: LLMs rely on keywords such as torch, cuda, or comm in the issue description to search for files containing these terms as candidate locations, even when these files are unrelated to the actual root cause.

The part analysis of Claude3.5 is as follows:
The error indicates that `torch.cuda.comm` is missing, which suggests issues with CUDA communication operations. The most important files to check are:
1. Core CUDA communication implementation files (`torch/cuda/comm.py`, `torch/csrc/cuda/comm.cpp`)
2. Module initialization files (`torch/cuda/__init__.py`, `torch/csrc/cuda/Module.cpp`) …

We also interacted with LLMs, such as GPT-4o-mini, to clarify that the issue originates from a third-party project. However, the model again suggested incorrect projects such as torch or torchvision, purely based on keyword matching.

Figure 5: In this example, LLMs rely on keywords such as torch, cuda, or comm in the issue description to search for files containing these terms as candidate locations, even when these files are unrelated to the actual root cause.

References

[1] Hiroki Kuramoto, Dong Wang, Masanari Kondo, Yutaro Kashiwa, Yasutaka Kamei, and Naoyasu Ubayashi. 2024. Understanding the characteristics and the role of visual issue reports. *Empirical Software Engineering* 29, 4 (2024), 89.

[2] Sebastiano Panichella, Gerardo Canfora, and Andrea Di Sorbo. 2021. "Won't We Fix this Issue?" Qualitative characterization and automated identification of wontfix issues on GitHub. *Information and Software Technology* 139 (2021), 106665.

[3] Wei Tao, Yucheng Zhou, Yanlin Wang, Wenqiang Zhang, Hongyu Zhang, and Yu Cheng. 2024. Magis: Llm-based multi-agent framework for github issue resolution. *Advances in Neural Information Processing Systems* 37 (2024), 51963–51993.

Issue Description *lllyasviel/Fooocus/issues/3689*

Exits upon attempting to load a model on Windows
What happened?
Attempting to run Fooocus on Windows 11 simply exits when attempting to load the default model, no error or nothing.
Steps to reproduce the problem
1.Install Fooocus on Windows 11 with a NVIDIA GPU ...
What should have happened?
It should've loaded the model successfully. ....

LLM's Response

GroundTruth
Project Name: Fooocus
File: presets/default**.json**
Line: 2

File: config**.txt**
File: config_modification_tutorial**.txt**

Closed-Book GPT-4o-mini ✖
Project: Fooocus
File: src/main**.py**
Class: MainApp
Function: load_model
Line: 45
...
File: src/utils/model_loader**.py**
Class: ModelLoader
Function: load_default_model
Line: 30
...

Pro-Structure Claude3.5 ✖
Project: Fooocus
File: entry_with_update**.py**
Function: main
...
File: shared**.py**
Function: initialize_environment
...

Hint Claude3.5 ✖
Project Name: Fooocus
File: modules/model_loader**.py**
Class: ModelLoader
Function: load_file_from_url
...
File: backend/fcbh/diffusers_load**.py**
File: modules/core**.py**
Class: StableDiffusionMode
...

Pipeline-ProStructure GPT-4o-mini ✖
Project: Fooocus
File: entry_with_update**.py**
Line: 1
...
File: backend/modules/model_loading**.py**
Line: 1
...

Pipeline-Hint GPT-4o-mini ✖
project: Fooocus
File: modules/model_loader**.py**
Function: load_file_from_url
...
File: modules/core**.py**
Class: StableDiffusionModel
...

Explanation

Code-Centric Bias: LLMs incorrectly assume that the issue is code-related, resulting in all predicted locations pointing to Python source files.
LLMs then follow a series of reasoning steps based on this initial misjudgment, resulting in a list of Python files unrelated to the actual root cause. For example, part of the analysis produced by Claude 3.5 is as follows:
Key areas to investigate:
1. Model loading logic → load_model function in entry_with_update.py
2. System compatibility checks → check_system_compatibility in modules/config.py
3. Path resolution → get_model_path function in modules/system_specific.py
4. Initialization procedures → initialize_model function in modules/core.py

Figure 6: In this example, LLMs mistakenly assume that the issue is code-related. Based on this incorrect assumption, they decompose the model-loading process step by step, generating a list of Python code locations that are unrelated to the actual failure.

Issue Description *pallets/flask/issues/5160*

Switch to importlib breaks scripts with app.run()
Switch to importlib breaks scripts with `app.run()`With a trivial script:
if __name__ == "__main__":
app.run(debug=True)
…

LLM's Response

GroundTruth
Project: flask
File: src/flask/scaffold.py
Function: _find_package_path
Line: 825
Line: 826
…
File: **src/flask/helpers.py**
Function: get_root_path
..

Hint Claude3.5 ✖
Project: flask
File: src/flask/scaffold.py
Function: _find_package_path
Line: 858
…
File: src/flask/app.py
Line: 630
…

Closed-Book GPT-4o-mini ✖
Project: flask
File: src/flask/app.py
Class: Flask
Function: __init__
...
project: flask
file: src/flask/scaffold.py
Function: find_package
Line: 898
Line: 858

Pipeline-ProStructure GPT-4o-mini ✖
Project: flask
File: src/flask/app.py
Line: 376
Line: 630
…
File: src/flask/scaffold.py
…

Pro-Structure Claude3.5 ✖
Project: flask
File: src/flask/scaffold.py
Function: _find_package_path
Line: 858
...
File: src/flask/app.py

Pipeline-Hint Claude3.5
Project: pytorch ✖
Project: flask
File: src/flask/scaffold.py
Function: _find_package_path
Line: 858
…
File: src/flask/app.py
…

Explanation

Responsibility Chain Tracking Limitation: LLMs can localize scaffold.py, where importlib is used to find package specs, but fail to identify **helpers.py**, which implements the fallback logic triggered by scaffold.py. Instead, they over-predict unrelated files like app.py or test modules.

The part analysis of Claude3.5 is as follows:
here are the key locations that need to be edited to fix the issue with importlib breaking scripts using `app.run()`:
Explanation:
1. The core fix needs to be in `scaffold.py` where importlib is used to find package specs
2. `app.py` needs updates since it calls the affected package finding functions
3. Tests must be added/updated to verify the fix works for `__main__` scripts
4. Documentation should be updated to reflect any changes
5. Example projects using `app.run()` should be verified to work correctly

Figure 7: In this example, LLMs correctly identify scaffold.py, where importlib is used to find package specs, but fail to identify helpers.py, which implements the fallback logic triggered by scaffold.py. Instead, LLMs over-predict unrelated files such as app.py or test modules, illustrating a failure to trace the narrow responsibility chain.

Issue Description *geekan/MetaGPT/issues/1174*

New provider: Amazon Bedrock (AWS)
Feature description
Please include support for Amazon Bedrock models. These models can be from Amazon, Anthropic, AI21, Cohere, Mistral, or Meta Llama 2.
Your Feature
Create a new LLM Provides under metagpt/provider for Amazon Bedrock
Include it in the LLMType available

LLM's Response

GroundTruth
Project: MetaGPT
File: metagpt/configs/llm_config.py
Line: 19
Line: 23

File: metagpt/configs/llm_config.py
Class: LLMType
Line: 34

Class: LLMConfig
Line: 77
…
File: metagpt/utils/**token_counter.py**
Line: 212
File: metagpt/provider/__init__.py
File: tests/metagpt/provider/**mock_llm_config.py**
..

Agentless Claude3.5 ✖
File: metagpt/**provider/bedrock/bedrock_provider.py**
Class: PROVIDERS
Function: get_provider
..
File: **metagpt/provider/bedrock**_api.py
Class: BedrockLLM

File: **metagpt/provider/bedrock**/__init__.py
Line: 21
Line: 32
…

BM25 ✖
File: …provider/bedrock/bedrock_provider.py
Class: MistralProvider
Class: MetaProvider
…

LocAgent Claude3.5 ✖
File: metagpt/configs/llm_config.py # Add new enum for Amazon Bedrock
Class: LLMType
Line: 17
..
File: metagpt/**provider/bedrock.py** #Create new Amazon Bedrock LLM Provider implementation
Line: 1
…
File: metagpt/provider/__init__.py #Import and register the new Bedrock provider
File: metagpt/**llm**.py
…

OpenHands GPT-4o-mini ✖
File: metagpt/configs/llm_config.py # Add
Class: LLMType

Function: api_type
File: metagpt/provider/__init__.py.
File: **metagpt/provider**/llm_provider_registry.py
Function: **LLMProvider**Registry
…

Explanation

Word-Matched Misleading: SOTA approaches are misled by file paths containing keywords such as ``provider'' or ``bedrock'', even though these files are unrelated to the issue localization.
Responsibility Chain Tracking Limitation: Although SOTA approaches like LocAgent and OpenHands can identify obvious components such as configs/llm_config.py, they fail to capture dependency files that are critical for the feature's functionality and verification. For example, **token_counter.py** handles token counting for the new provider, and **mock_llm_config.py** provides the test scaffolding needed to validate its behavior.

Figure 8: In this example from the MetaGPT feature request, SOTA approaches such as LocAgent and OpenHands exhibit keyword matched pseudo localization by being misled by files containing terms like "provider" or "bedrock". They also show responsibility chain tracking failures by missing critical dependency files such as token_counter.py and mock_llm_config.py that are required for correct functionality and verification.

Issue Description *ansible/ansible/issues/27729*

Removed restricted key from module data: ansible_lxc_bridge
COMPONENT NAME: **Gathering Facts**
ANSIBLE VERSION: ansible 2.3.1.0 config file = /etc/ansible/ansible.cfg
configured module search path = Default w/o overrides
OS / ENVIRONMENT Archlinux, but probably not platform specific
SUMMARY
During **gathering facts** I get following warning

TASK [**Gathering Facts**] **
[WARNING]: **Removed restricted key from module data: ansible_lxc_bridge** = {u'macaddress': u'70:85:c2:0b:a3:4a', u'features': {}, u'interfaces': [u'vethG18OR8', u'enp0s31f6', u'vethWYJVBN'], u'mtu': 1500, u'active': True, u'promisc': False, u'stp': False, u'ipv4': {u'broadcast': u'192.168.0.255', u'netmask': u'255.255.255.0', u'network': u'192.168.0.0', u'address': u'192.168.0.110'}, u'ipv6': [{u'scope': u'link', u'prefix': u'64', u'address': u'fe80::7285:c2ff:fe0b:a34a'}], u'device': u'lxc_bridge', u'type': u'bridge', u'id':
STEPS TO REPRODUCE: Try to run **playbook** against host with **lxc** and bridge configured.
EXPECTED RESULTS: No warning.

LLM's Response

GroundTruth
Project: ansible
File: lib/ansible/playbook/task.py
Class: Task
Function: preprocess_data
Line: 208
Line: 214
…
File: lib/ansible/plugins/action/__init__.py
Class: ActionBase
Function: _clean_returned_data
Line: 783

Agentless Claude3.5 ✖
File: setup.py
Function: setup
Line: 28
Line: 208
Line: 209
Line: 214
Line: 215
..
Line: 223

File: v1/tests/TestVaultEditor.py
Line: 783

Note: The two files are retrieved using **embedding-based similarity** to the issue description.

BM25 ✖
File: v1/ansible/inventory/__init__.py
Class: Inventory
Function: get_**restriction**
Function: **restrict**_to
…

LocAgent Claude3.5 ✖
File: lib/ansible/module_utils/**facts**/network/linux.py # otential location for network interface fact gathering
File: lib/ansible/module_utils/**facts/collector**.py # potential location for fact collection logic
File: lib/ansible/modules/system/setup.py # core module for gathering facts
File: lib/ansible/module_utils/**facts**/namespace.py
…

OpenHands GPT-4o-mini ✖
File: lib/ansible/executor/**play_iterator.py**
Class: PlayIterator Note: File instantiating the
Function: __init__ "Gathering Facts" task
… setup_task.name = 'Gathering Facts'
File: contrib/inventory/**lxc**_inventory.py
Class: **lxc**
File: contrib/inventory/libvirt_**lxc**.py
Class: libvirt_**lxc**
File: contrib/inventory/proxmox.py
…

Explanation

Word-Matched Misleading: SOTA approaches overfit to **keyword similarity** (e.g., "lxc", "bridge", "fact"), retrieving unrelated inventory or network modules. They miss the true root cause in regex-based sanitization logic that contains no matching keywords.
Responsibility Chain Tracking Limitation: SOTA approaches fail to trace the responsibility chain across the multi-stage execution pipeline of fact gathering.
Although the issue arises during **"Gathering Facts,"** the relevant control flow spans:
task preprocessing (playbook/task.py: preprocess_data),
action result sanitization (action/__init__.py: _clean_returned_data), and
regex-based filtering of ansible_<conn>_* keys (re.compile('^ansible_%s_' % conn_name)).
Current systems cannot follow this chain and instead **attribute the failure to modules that merely *mention* LXC or network interface**s.

Figure 9: In this example, SOTA methods are misled by surface keyword matches (e.g., lxc, fact) and fail to trace the multi-stage responsibility chain behind fact gathering, causing them to localize the issue to irrelevant inventory or network modules instead of the true root cause in regex-based sanitization.

Prompt Template of Closed-Book LLM

Please look through the following GitHub problem description and provide a complete list of locations with full path that one would need to edit to fix the problem.

Each location should include:
Project name
Full file path (absolute within the repository, e.g., src/utils/helpers.py)
Class names (only for source code files)
function names (only for source code files)
Line numbers

Notes:
1. Only include the full file paths in your response.
2. Include all necessary files to resolve the issue, and do not include any unrelated files.
3. Your answer no more than 30 files.

The returned files should be separated by new lines ordered by most to least important and wrapped with
``````

### **GitHub Problem Description** ###
{issue report consisting of project name, issue title and issue description}

###

for example, Your response format should like that: (each block only has one file)
```
project: XXX
file: file1.py
Class: XXX
Function: XXX
Line: XXX
Line: XXX

project: XXX
file: b/file1.js
Class: XXX
Class: XXX
Function: XXX
Function: XXX
Line: XXX
Line: XXX
Line: XXX

project: XXX
file: a/xx/yy/README.md
Line: XXX
Line: XXX
...
```

**Figure 10: Prompt template of closed-book LLM in project localization for issue resolution.**
```

Prompt Template of Project-Structure LLM

Based on the provided **Project Structure**, please look through the following GitHub problem description and provide a complete list of locations with full path that one would need to edit to fix the problem.

Each location should include:
Project name
Full file path (absolute within the repository, e.g., src/utils/helpers.py)
Class names (only for source code files)
function names (only for source code files)
Line numbers

Notes:
1. Only include the full file paths in your response.
2. Include all necessary files to resolve the issue, and do not include any unrelated files.
3. Your answer no more than 30 files.

The returned files should be separated by new lines ordered by most to least important and wrapped with
``` ``` ```

### **GitHub Problem Description** ###
{issue report consisting of project name, issue title and issue description}

###

### **Project Structure**###
{project structure is a hierarchical representation of the directory tree, along with relative paths, file names and file extensions}

###

for example, Your response format should like that: (each block only has one file)
```
project: XXX
file: file1.py
Class: XXX
Function: XXX
Line: XXX
Line: XXX

project: XXX
file: b/file1.js
Class: XXX
Class: XXX
Function: XXX
Function: XXX
Line: XXX
Line: XXX
Line: XXX

project: XXX
file: a/xx/yy/README.md
Line: XXX
Line: XXX
...
```

**Figure 11: Prompt template of Project-Structure LLM in project localization for issue resolution.**
```

Prompt Template of Location-Hint LLM

Based on the provided **Location Hint and Project Structure**, please look through the following GitHub problem description and provide a complete list of locations with full path that one would need to edit to fix the problem.

Each location should include:
Project name
Full file path (absolute within the repository, e.g., src/utils/helpers.py)
Class names (only for source code files)
function names (only for source code files)
Line numbers

Notes:
1. Only include the full file paths in your response.
2. Include all necessary files to resolve the issue, and do not include any unrelated files.
3. Your answer no more than 30 files.

The returned files should be separated by new lines ordered by most to least important and wrapped with
``````

GitHub Problem Description
{issue report consisting of project name, issue title and issue description}

###

Location Hint###
Your found Locations should be { in-project file/runtime file/third-party file/user-authored file }
Your found file types should be code/test/documentation/configuration/asset }
###

Project Structure###
{project structure is a hierarchical representation of the directory tree, along with relative paths, file names and file extensions. Note: According to the given location hint, only the corresponding file types are shown in the structure.
}

###

for example, Your response format should like that: (each block only has one file)
```
project: XXX
file: file1.py
Class: XXX
Function: XXX
Line: XXX
Line: XXX

project: XXX
file: b/file1.js
Class: XXX
Class: XXX
Function: XXX
Function: XXX
Line: XXX
Line: XXX
Line: XXX

project: XXX
file: a/xx/yy/README.md
Line: XXX
Line: XXX
...
```

Figure 12: Prompt template of Location-Hint LLM in project localization for issue resolution.
```

Step 1 Prompt Template of Pipeline (Project-Structure Guided) LLM

Based on the provided **Project Structure**, please look through the following GitHub problem description and provide a complete location list with full path that one would need to edit to fix the problem.

Each location should include:
Project name
Full file path (absolute within the repository, e.g., src/utils/helpers.py)

Notes:
1. Only include the full file paths in your response.
2. Include all necessary files to resolve the issue, and do not include any unrelated files.
3. Your answer no more than 30 files.

The returned files should be separated by new lines ordered by most to least important and wrapped with
``````

### **GitHub Problem Description** ###
{issue report consisting of project name, issue title and issue description}
###

### **Project Structure**###
{project structure is a hierarchical representation of the directory tree, along with relative paths, file names and file extensions.
}
###

for example, Your response format should like that: (each block only has one file)
```
project: XXX
file: file1.py

project: XXX
file: b/file1.js

project: XXX
file: a/xx/yy/README.md
...
```

**Figure 13: Step 1 Prompt template of PipeLine (Project-Structure Guided) LLM in project localization for issue resolution.**
```

Step 2 Prompt Template of Pipeline (Project-Structure Guided) LLM

Based on the **file list**, please look through the following GitHub problem description and provide a complete location list with full path that one would need to edit to fix the problem.

Each location should include:
Full file path (absolute within the repository, e.g., src/utils/helpers.py)
Class names (only for source code files)
function names (only for source code files)
Line numbers

Notes:
1. Only include the full file paths in your response.
2. Include all necessary files to resolve the issue, and do not include any unrelated files.
3. Your answer no more than 30 files.

Returned files should be separated by new lines ordered by most to least important and wrapped with ``````

### **GitHub Problem Description** ###
{issue report consisting of project name, issue title and issue description}
###

### **File List** ###
{{Files predicted by LLMs in Step 1. For valid source files, we also include their classes and functions extracted by parsing.}
###

for example, Your response format should like that: (each block only has one file)
```
project: XXX
file: file1.py
Class: XXX
Function: XXX
Line: XXX
Line: XXX

project: XXX
file: b/file1.js
Class: XXX
Class: XXX
Function: XXX
Function: XXX
Line: XXX
Line: XXX
Line: XXX

project: XXX
file: a/xx/yy/README.md
Line: XXX
Line: XXX
...
```

**Figure 14: Step 2 Prompt template of PipeLine (Project-Structure Guided) LLM in project localization for issue resolution.**

Step 1 Prompt Template of Pipeline (Location-Hint Guided) LLM

Based on the provided **Location Hint and Project Structure**, please look through the following GitHub problem description and provide a complete location list with full path that one would need to edit to fix the problem.

Each location should include:
Project name
Full file path (absolute within the repository, e.g., src/utils/helpers.py)

Notes:
1. Only include the full file paths in your response.
2. Include all necessary files to resolve the issue, and do not include any unrelated files.
3. Your answer no more than 30 files.

The returned files should be separated by new lines ordered by most to least important and wrapped with
``````

GitHub Problem Description
{issue report consisting of project name, issue title and issue description}
###

Location Hint###
Your found Locations should be { in-project file/runtime file/third-party file/user-authored file }
Your found file types should be code/test/documentation/configuration/asset }
###

Project Structure###
{project structure is a hierarchical representation of the directory tree, along with relative paths, file names and file extensions. Note: According to the given location hint, only the corresponding file types are shown in the structure.
}
###

for example, Your response format should like that: (each block only has one file)
```
project: XXX
file: file1.py

project: XXX
file: b/file1.js

project: XXX
file: a/xx/yy/README.md
...
```

Figure 15: Step 1 Prompt template of PipeLine (Location-Hint Guided) LLM in project localization for issue resolution.
```

Step 2 Prompt Template of Pipeline (Location-Hint Guided) LLM

Based on the **file list**, please look through the following GitHub problem description and provide a complete location list with full path that one would need to edit to fix the problem.

Each location should include:
Full file path (absolute within the repository, e.g., src/utils/helpers.py)
Class names (only for source code files)
function names (only for source code files)
Line numbers

Notes:
1. Only include the full file paths in your response.
2. Include all necessary files to resolve the issue, and do not include any unrelated files.
3. Your answer no more than 30 files.

Returned files should be separated by new lines ordered by most to least important and wrapped with ``````

GitHub Problem Description
{issue report consisting of project name, issue title and issue description}
###

File List
{{Files predicted by LLMs in Step 1. For valid source files, we also include their classes and functions extracted by parsing.}
###

for example, Your response format should like that: (each block only has one file)
```
project: XXX
file: file1.py
Class: XXX
Function: XXX
Line: XXX
Line: XXX

project: XXX
file: b/file1.js
Class: XXX
Class: XXX
Function: XXX
Function: XXX
Line: XXX
Line: XXX
Line: XXX

project: XXX
file: a/xx/yy/README.md
Line: XXX
Line: XXX
...
```

Figure 16: Step 2 Prompt template of PipeLine (Location-Hint Guided) LLM in project localization for issue resolution.