

# 000 BENCHMARKING AND ENHANCING LLM AGENTS IN 001 LOCALIZING LINUX KERNEL BUGS 002

003 **Anonymous authors**  
004

005 Paper under double-blind review  
006

## 007 ABSTRACT 008

009 The Linux kernel is a critical system, serving as the foundation for numerous  
010 systems. Bugs in the Linux kernel can cause serious consequences, affecting  
011 billions of users. Fault localization (FL), which aims at identifying the  
012 buggy code elements in software, plays an essential role in software quality  
013 assurance. While recent LLM agents have achieved promising accuracy in  
014 FL on recent benchmarks like SWE-bench, it remains unclear how well these  
015 methods perform in the Linux kernel, where FL is much more challenging  
016 due to the large-scale code base, limited observability, and diverse impact  
017 factors. In this paper, we introduce `LINUXFLBENCH`, a FL benchmark  
018 constructed from real-world Linux kernel bugs. We conduct an empirical  
019 study to assess the performance of state-of-the-art LLM agents on the Linux  
020 kernel. Our initial results reveal that existing agents struggle with this task,  
021 achieving a best top-1 accuracy of only 41.6% at file level. To address this  
022 challenge, we propose `LINUXFL+`, an enhancement framework designed to  
023 improve FL effectiveness of LLM agents for the Linux kernel. `LINUXFL+`  
024 substantially improves the FL accuracy of all studied agents (e.g., 7.2% -  
025 11.2% accuracy increase) with minimal costs.  
026

## 027 1 INTRODUCTION 028

029 The Linux kernel is a critical system which serves as the foundation for numerous operating  
030 systems, servers, and embedded systems, and has evolved over decades with contributions  
031 from thousands of developers and billions of users (The Linux Foundation, 2020). Given  
032 the widespread adoption of the Linux kernel, bugs in the Linux kernel can cause serious  
033 consequences, affecting a vast number of users. Therefore, extensive research has been dedi-  
034 cated to developing automated software quality assurance techniques (e.g., testing (Bligh  
035 & Whitcroft, 2006; Chen et al., 2013; Yang et al., 2025b) and debugging (Bissyandé et al.,  
036 2012; Edge, 2019; Serrano et al., 2020; Jeong et al., 2023)) specifically for the Linux kernel.  
037

038 Fault localization (FL), which aims at identifying the buggy code elements (e.g., files or  
039 functions) in software, plays a critical role in software quality assurance. Given the codebase  
040 of the buggy software and the bug report (e.g., a user-reported bug symptom description),  
041 automated FL techniques return a list of buggy code elements ranked by their suspiciousness  
042 (i.e., the probability of being buggy). In particular, accurate FL is a prerequisite for bug  
043 fixing, as a bug cannot be resolved without correctly identifying the faulty code location.

044 Traditional FL techniques mainly leverage heuristics (Abreu et al., 2006; Wong et al., 2014b)  
045 or information retrieval (IR) (Zhou et al., 2012; Saha et al., 2013) to identify buggy code  
046 elements. More recently, with the advance in large language models (LLMs), LLM agents Liu  
047 et al. (2024) have demonstrated remarkable accuracy in FL. Equipped with tool invocation,  
048 agents can autonomously navigate codebases to identify the buggy location. For example,  
049 the state-of-the-art agents such as SWE-Agent (Yang et al., 2024), AutoCodeRover (Zhang  
050 et al., 2024), Agentless (Xia et al., 2024), achieve around 70% accuracy in localizing buggy  
051 files for Python software in the benchmark SWE-bench (Jimenez et al., 2024).

052 Although achieving promising FL effectiveness, existing agents have been mainly evaluated  
053 on general software at moderate scales. It remains unclear how existing agents perform  
in complex, large-scale software systems like the Linux kernel. In particular, FL in Linux

054 kernel is more challenging than general software due to the following factors. (1) *Large-055 scale Codebase*: the Linux kernel has a massive codebase significantly larger than general056 software. For example, the v5.8 release of Linux kernel includes over 69K files and 28M lines057 of code (The Linux Foundation, 2020), which is over 30 times the scale of even the largest058 project in the most widely-used benchmark SWE-bench. (2) *Limited Observability*: given059 the real-time nature of the Linux kernel with the need to minimize overhead, the kernel060 restricts the use of instrumentation and logging mechanisms during runtime. Additionally,061 the kernel operates in a privileged mode, isolated from user space. As a result, user-reported062 bug descriptions often lack detailed runtime information and debugging hints, creating a063 significant gap between the user description and the actual root causes. (3) *Diverse Impact064 Factors*: kernel bugs are influenced by a wide range of factors, including hardware variability065 (e.g., architectural configurations) and runtime variability (e.g., system load or timing).066 These factors lead to an exponentially large reasoning space to accurately diagnose the root067 causes of errors. Given the unique challenges and the importance of the kernel, this work068 aims at investigating the FL effectiveness of state-of-the-art LLM agents on the Linux kernel.069

**Benchmark.** We first build a new benchmark LINUXFLBENCH of 250 real-world FL tasks070 for the Linux kernel. Each FL task in LINUXFLBENCH includes a user-submitted bug071 report, the buggy Linux kernel codebase, and the ground-truth buggy locations based on072 the associated commit patches. LINUXFLBENCH involves a wide range of Linux kernel073 bugs, spanning over 120 Linux kernel versions and 66 different kernel components. The074 FL tasks are significantly more challenging than those in SWE-bench, as evidenced by the075 substantially larger codebases (10–30× more files and lines of code) and more complex bug076 reports (approximately 1.5× more words).

**Empirical Study.** On LINUXFLBENCH, we make the first attempt to evaluate state-of-077 the-art LLM agents in localizing Linux kernel bugs. Our results reveal the limited FL078 effectiveness (e.g., 36.8% - 41.6% accuracy) of existing agents in the Linux kernel; such a079 FL accuracy is much lower than their performance on general software systems (a 16.7%080 - 31.9% accuracy drop from SWE-bench). We further perform bad case analysis and find081 that existing agents mainly miss the buggy files as they fail to capture the related files or to082 cover complete root causes of kernel bugs. The results indicate that FL in the Linux kernel083 is indeed a more challenging task, highlighting the need for building more advanced agents084 to localize bugs in large and complex software systems like the Linux kernel.

**Technique.** Inspired by our study above, we further propose an enhancing framework085 LINUXFL<sup>+</sup>, which improves the FL effectiveness of existing agents for the Linux kernel.086 LINUXFL<sup>+</sup> incorporates two expansion strategies to refine the prediction results of existing087 agents: directory-aware expansion to include buggy files based on the repository structure,088 and potential cause expansion to identify buggy files based on the additional bug knowledge089 from Linux kernel mailing list (LKML) (Kernel.org, 2025b). Our evaluation results show090 that LINUXFL<sup>+</sup> can substantially improve the FL accuracy of all studied agents (e.g., 7.2%091 - 11.2% accuracy increase) with minimal costs. Moreover, the ablation analysis confirms the092 contribution of each expansion strategies.

## 095 2 BACKGROUND AND RELATED WORK

**097 FL Task Definition.** Given the bug report and codebase, FL techniques identify buggy098 code elements (e.g., files or functions). Formally, let a codebase be represented as a set of099 code elements,  $\mathcal{C} = \{ce_1, ce_2, \dots, ce_N\}$ , where  $N$  denotes the total number of code elements.100 A bug report  $BR$  typically includes a title, a description, and optional metadata (e.g.,101 component and hardware information in the context of Linux kernel), and can be expressed102 as  $BR = (title, desc, meta)$ . A FL task can be modeled as:  $\text{FL} : BR, \mathcal{C} \rightarrow list(\mathcal{C})$ , where103  $list(\mathcal{C})$  denotes a list of code elements that ranked by their probabilities of being buggy.

**104 Existing FL techniques.** FL techniques have been extensively studied in literature:

- 106 Coverage-based FL.** Besides bug reports, some FL techniques leverage test coverage to107 identify buggy locations, such as SBFL (Abreu et al., 2006; Wong et al., 2014b), GNN-based108 FL (Lou et al., 2021), AutoFL (Kang et al., 2024), and AgentFL (Qin et al., 2024).

Table 1: Existing Benchmarks for Software Maintenance

Benchmark	Language	# Repo	# Bugs	Data Source	Linux-Related	User-reported
Defects4J (Just et al., 2014)	Java	17	854	Bug Tracking Systems	✗	✓
Linux-3.16 (Saha et al., 2014)	C	1	1,548	Bug Tracking Systems	✓	✓
SWE-bench (Jimenez et al., 2024)	Python	12	2,294	GitHub Pull Requests	✗	✓
FAUN-Eval-fix (Hu et al., 2024)	Multiple	17	300	GitHub Pull Requests/Issues	✗	✓
KBENCHSYZ (Mathai et al., 2024)	C	113	279	Fuzzing-Detected Crashes	✓	✗
Loc-Bench (Chen et al., 2025)	Python	165	560	GitHub Issues	✗	✓
SWE-lancer (Miserendino et al., 2025)	Python	1	1,488	Upwork Issues	✗	✓
<b>LinuxFLBench</b>	C	120	250	Bug Tracking Systems	✓	✓

However, coverage and executable failure-triggering tests are not always available in practice. Especially for the large systems like Linux kernel, users report bugs by textually describing the error symptoms. Therefore, coverage-based FL cannot be applied to the Linux kernel when only bug reports are available, which thus is not included in this work.

- **Information Retrieval (IR) Based FL.** FL can be formulated as an information retrieval (IR) problem, where a bug report serves as a query to rank code files by relevance. Existing IR-based FL techniques use various similarity measures, such as Vector Space Model (VSM) (Zhou et al., 2012; Saha et al., 2013; 2014; Wang & Lo, 2014; Wong et al., 2014a), Dirichlet Language Model (DLM) (Sisman et al., 2017), or deep learning approaches (Huo et al., 2021; Ciborowska & Damevski, 2022; Mohsen et al., 2023). In this work, we empirically evaluate IR-based FL in the Linux kernel.
- **Agent-based FL.** Recent advances in LLM agents have shown strong performance in software maintenance tasks, including FL. For instance, SWE-Agent (Yang et al., 2024) incorporates a custom-built Agent-Computer Interface to navigate entire repositories; AutoCodeRover (Zhang et al., 2024) equips LLMs with code search capabilities to retrieve relevant code contexts; Agentless (Xia et al., 2024) refines the localization process by restricting the decision-making autonomy of agents. In this work, we not only make the first attempt to empirically evaluate existing agents in the Linux kernel, but also propose a framework to enhance their performance in this challenging domain.

**Benchmarks for Software Maintenance.** As FL is a key sub-task in software maintenance, we revisit existing software maintenance benchmarks in Table 1. The majority of existing benchmarks focus on general software systems in Java or Python. In contrast, our benchmark LINUXFLBENCH specifically targets the large-scale system Linux kernel. Only two prior benchmarks involve the kernel: Linux-3.16 (Saha et al., 2014), which is limited to a single old version, and KBENCHSYZ (Mathai et al., 2024), which collects Syzkaller (Google, 2025)-detected crash bugs. LINUXFLBENCH differs by (1) covering a wider range of kernel versions, (2) including diverse real-world bug types beyond crashes (e.g., functionality and performance bugs), and (3) sourcing all bugs from user reports rather than automated fuzzing. Thus, LINUXFLBENCH complements existing efforts by offering a more comprehensive benchmark for evaluating advanced FL techniques in the Linux kernel.

### 3 LINUXFLBENCH: A FL BENCHMARK FOR LINUX KERNEL

LINUXFLBENCH is a new benchmark of 250 real-world Linux kernel FL tasks.

#### 3.1 CONSTRUCTION OF LINUXFLBENCH

LINUXFLBENCH is constructed through three phases, as described in Appendix B.1.

**Step 1: Bug Report Collection.** We collected Linux kernel bug reports from Kernel.org Bugzilla (Kernel.org, 2025a) up to December 31, 2024. Each report includes a *title*, *description*, and relevant *metadata* (e.g., kernel version, environment). To ensure code availability, we retained only reports linked to kernel versions hosted on the official Linux website (Kernel.org, 2025c). For ground-truth reliability, we required reports marked as “CLOSED” and “CODE\_FIX” in the bug tracking system. Furthermore, we included only bug reports with patches attached, enabling us to identify the buggy locations based on the patch information. In total, we collected 2,138 bug reports during this step.

**Step 2: Buggy Location Identification.** For each collected bug report, we identified the location modified in the developer-committed patch as the ground-truth buggy location. Specifically, we traversed source files with the extensions `.c` or `.h`, skipping other file types

such as `README` or `Makefile`. Following SWE-bench-lite (Jimenez et al., 2024), we kept only unambiguous cases where exactly one file was modified to ensure the reliability of the ground truth. After this step, 635 bug reports with identified buggy files were obtained.

**Step 3: Manual Inspection.** To further ensure quality, we manually reviewed the collected data. Three human annotators checked each bug as follows: (1) bug reports without actual bugs (e.g., those that primarily submit patches) were excluded; (2) bug reports with sufficient information (e.g., clear natural language descriptions or detailed system logs) were retained; (3) bug reports that explicitly mentioned buggy locations or fix solutions were excluded. As a result, the final dataset comprises 250 high-quality FL tasks, and each task includes a bug report, the buggy codebase, and the ground-truth buggy file and method(s). A detailed sample is shown in Appendix B.2.

### 3.2 CHARACTERISTICS OF LINUXFLBENCH

LINUXFLBENCH presents challenging tasks with complex bug reports and large-scale codebase, offering multidimensional diversity across kernel versions, products, and bug types.

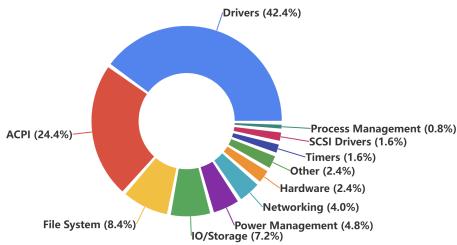


Figure 1: Task Distribution across Products

Table 2: Task Scales of LINUXFLBENCH and SWE-bench.

Benchmark	Bug Description		Codebase	
	# Words	# Files	# Lines	
Mean	LINUXFLBENCH	283.1	28,808	11,492K
	SWE-bench*	195.1	3,010	438K
Max	LINUXFLBENCH	5,139	67,073	28,178K
	SWE-bench*	4,477	5,890	886K

\* Source: SWE-bench (Jimenez et al., 2024).

**Scale.** Table 2 compares the scale of tasks in LINUXFLBENCH and SWE-bench. Our dataset is more challenging, with the codebase tens of times larger and bug reports that are more detailed and complex. We also compare stack trace lengths, the presence of bug location in bug reports, and the sizes of buggy files and golden patches, all of which further underscore the greater complexity of our dataset. More details are provided in Appendix B.3.

**Products.** Fig. 1 shows the distribution of LINUXFLBENCH across kernel products (i.e., high-level categories defined in Bugzilla). In particular, bugs span 16 products, with *Drivers*, *ACPI*, and *File System* being the largest categories. At a finer granularity, the benchmark covers a diverse set of 66 components, with the most frequent being *network-wireless* (6.4%), *Video* (6.0%), *Network* (5.2%), *Power-Battery* (4.8%), and *Sound* (4.4%).

**Versions.** The Linux kernel has evolved over several decades, resulting in the release of numerous versions. LINUXFLBENCH captures this temporal diversity by including bugs from a broad range of kernel versions, covering a total of 120 distinct versions.

**Bug Types.** LINUXFLBENCH encompasses a broad spectrum of bugs by symptoms and causes. Symptomatically, it includes common issues such as system crashes (14.8%), power malfunctions (13.6%), and network failures (10.8%). Causally, frequent sources are hardware configuration faults (19.6%), memory defects (15.6%), and data handling errors (15.2%).

## 4 EVALUATION OF LLM AGENTS ON LINUXFLBENCH

We empirically evaluate SOTA LLM agents on LINUXFLBENCH to investigate their FL effectiveness in the Linux kernel.

### 4.1 STUDY SETUP

**Studied Baselines.** (1) *LLM agents*. We study three SOTA LLM agents, i.e., SWE-Agent (Yang et al., 2024), AutoCodeRover (Zhang et al., 2024), and Agentless (Xia et al.,

216 2024), as they are fully open-sourced and achieve high effectiveness in recent software maintenance leaderboard (SWE-bench, 2025). All agents are equipped with GPT-4o (gpt-4o-2024-08-06) as backbone LLMs (OpenAI, 2024). The detailed implementation of these agents is  
 217 in Appendix C. (2) *IR-based baselines*. For comparison, we also include traditional IR-based  
 218 FL baselines for comparison. Specifically, we include the classic IR-based methods BugLoca-  
 219 tor (Zhou et al., 2012) and BLUiR (Saha et al., 2013), along with widely used IR techniques  
 220 such as BM25 (Robertson et al., 1995) and Sentence-BERT (Reimers & Gurevych, 2019).  
 221

222 **Evaluation Metrics.** In line with previous FL work (Xia & Lo, 2023; Zhou et al., 2012;  
 223 Saha et al., 2014), we include the widely-used metrics like recall at top-k ( $k = 1, 5, 10$ ) and  
 224 the Mean Reciprocal Rank (MRR) to evaluate the FL effectiveness.  
 225

## 227 4.2 QUANTITATIVE ANALYSIS

229 Table 3 shows the overall file-level FL effectiveness of studied techniques on LINUXFLBENCH.

230 **Comparison with IR-based methods.** Overall, existing agents outperform all tra-  
 231 ditional IR methods, indicating the benefits  
 232 from agentic solutions in identifying buggy  
 233 locations for large scale systems. For  
 234 instance, SWE-Agent achieves the best effec-  
 235 tiveness with an MRR of 0.476, significantly  
 236 surpassing other methods. Among IR meth-  
 237 ods, BLUiR performs the best, but only with an MRR of 0.321.  
 238

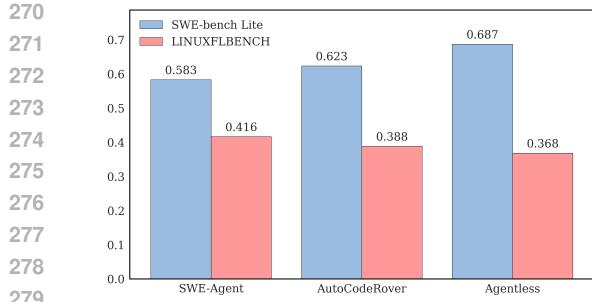
239 **Comparison with general software system.** Although outperforming traditional IR  
 240 methods, existing agents still exhibit limited overall effectiveness on Linux kernel. For  
 241 instance, even the best-performing SWE-Agent only achieves a top-1 recall of only 0.416 on  
 242 LINUXFLBENCH, which is much lower than when it is applied to general software systems  
 243 (i.e., SWE-bench). In particular, Fig.2 compares the FL effectiveness of agents in Linux  
 244 systems (i.e., on LINUXFLBENCH) and in general software systems (i.e., on SWE-bench).  
 245 The reported SWE-bench results are from previous work (Xia et al., 2024). We can observe  
 246 a marked performance decline for all the LLM agents on LINUXFLBENCH compared to  
 247 SWE-bench, with recall values decreasing by more than 0.15. Such an effectiveness drop  
 248 underscores the heightened challenges associated with FL in the larger and more intricate  
 249 Linux kernel codebase than general software systems.  
 250

251 **Uniqueness and Union.** Fig. 3 presents the overlapped/unique bugs that are correctly  
 252 localized at top-1 by studied agents. We could observe complementary strengths of the  
 253 different approaches, as each agent can uniquely resolve 12 - 20 bugs. Nevertheless, even  
 254 when combining the correctly-localized bugs of all agents, only 146 bugs out of 250 total bugs  
 255 can be successfully localized (i.e., 58.4% top-1 recall). It further highlights the considerable  
 256 challenges that agents still face in performing FL within the complex Linux kernel.  
 257

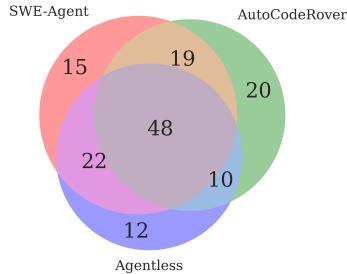
## 258 4.3 QUALITATIVE ANALYSIS

259 To further understand why agents perform poorly in Linux kernel, we manually examine  
 260 bad cases where all studied agents fail to correctly localize the buggy files. Overall, we find  
 261 two main reasons for the limited effectiveness as follows.

262 **Confusion Among Related Files.** As a large-scale software system, bugs in Linux ker-  
 263 nel often propagate along a long chain, where many related files are associated with each  
 264 other via function calls or data dependencies. While agents might be capable of coarse-  
 265 grained FL (e.g., correctly identifying the buggy directories or high-level modules), they  
 266 struggle to further precisely pinpoint the exact faulty file/method among all the related  
 267 files. This challenge is indirectly evidenced by the fact that each Linux directory in LIN-  
 268 UXFLBENCH contains, on average, approximately twice as many files (16 vs. 8) as those in  
 269 SWE-bench, making fine-grained localization within directories more difficult. For example,  
 Appendix D.1 shows a bad case where all agents wrongly localize the files that are in the  
 same directory as the buggy file.



280 Figure 2: Performance of LLM agents on  
281 SWE-bench and LINUXFLBENCH.  
282



280 Figure 3: Venn Diagram for Correctly Lo-  
281 calized Bugs by LLM agents.  
282

283 **Limited Exploration of Potential Causes.** Given the complexity of the Linux kernel, a  
284 bug can arise from diverse and non-obvious root causes. Current agents narrowly focus on  
285 a small set of highly probable causes, failing to explore a broader range of potential causes.  
286 Consequently, this limited exploration leads to missed opportunities for correctly identifying  
287 the buggy file. Appendix D.2 shows a bad case that all agents miss the real cause.  
288

## 289 5 LINUXFL<sup>+</sup>: AN ENHANCING FRAMEWORK 290

291 To address the limitations of existing agent-based methods, we propose a novel enhancing  
292 framework LINUXFL<sup>+</sup>, which improves the FL effectiveness of agents in the Linux kernel.  
293

### 294 5.1 APPROACH 295

296 As discussed in Section 4.3, given the huge space of Linux kernel, existing agents fail to cap-  
297 ture the relationship between files or to cover a complete pool of potential causes. Therefore,  
298 the main insight of LINUXFL<sup>+</sup> is to *expand* the prediction results of existing agents with  
299 both the repository structure and the root causes.  
300

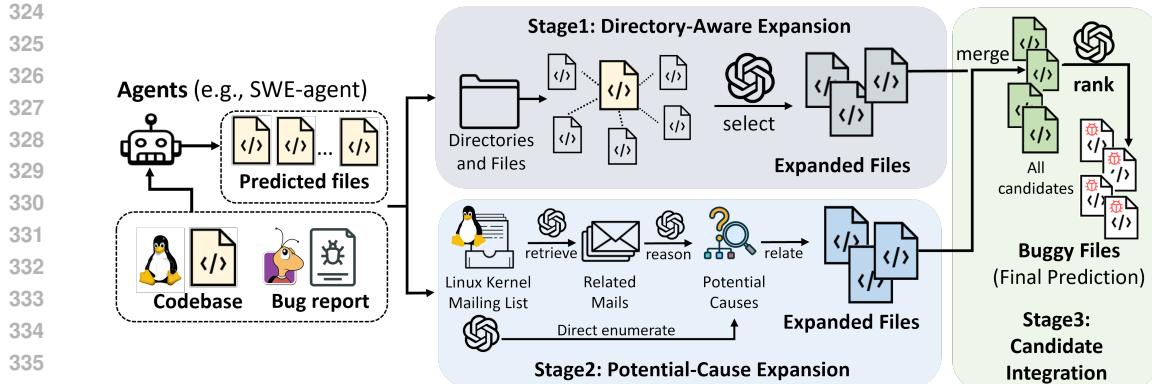
301 Fig. 4 shows the overall workflow of LINUXFL<sup>+</sup>. Given the buggy files predicted by any agent  
302 (e.g., AutoCodeRover), LINUXFL<sup>+</sup> refines the prediction via the following three phases. (1) *Di-*  
303 *rectory-Aware Expansion*: LINUXFL<sup>+</sup> expands the search scope within directories of the  
304 initial predictions generated by LLM agents. LINUXFL<sup>+</sup> then re-selects bug-related files  
305 within these directories, enabling a more thorough exploration of related files; (2) *Potential*  
306 *Cause Expansion*: LINUXFL<sup>+</sup> explores as many potential causes as possible to scale the  
307 related files. LINUXFL<sup>+</sup> includes two hypothesizing strategies to expand the potential  
308 causes for the given bug report, leveraging both the original capabilities of LLMs (i.e.,  
309 direct hypothesis) and the additional knowledge from Linux kernel mailing list (i.e., mail-  
310 augmented hypothesis); (3) *Candidate Integration*: all relevant files are merged as candidates,  
311 followed by a re-ranking process to further refine the results.  
312

#### 313 5.1.1 DIRECTORY-AWARE EXPANSION 314

315 While existing agents can generally identify the correct modules related to a bug, they  
316 often struggle to distinguish relevant files within those modules. To address this limitation,  
317 LINUXFL<sup>+</sup> first expands the search scope to include all files in the directories of the initially  
318 predicted files. Using this expanded candidate set, the LLM re-selects files likely related  
319 to the bug. We retain the top-k ( $k=10$ ) most relevant files as the expanded results. This  
320 approach provides the LLM with an additional opportunity to identify buggy files, enabling  
321 a more comprehensive exploration of related files. Detailed prompts are in Appendix E.1.  
322

#### 323 5.1.2 POTENTIAL CAUSE EXPANSION 324

325 Current agents tend to focus narrowly on few highly probable causes within limited steps.  
326 However, diagnosing complex bugs often requires an iterative “guess-and-check” process (Al-  
327 aboudi & LaToza, 2023; Layman et al., 2013; Liu et al., 2025), where developers form  
328

Figure 4: Overview of LINUXFL<sup>+</sup>.

experience-based hypotheses and progressively refine their understanding to isolate the root cause. Inspired by this, we expand bug-related files by exploring a broader range of potential causes. Specifically, we design two types of hypothesizing strategies to expand probable causes: *Direct Hypothesis*, leveraging models’ inherent knowledge on Linux kernel, and *Mail-Augmented Hypothesis*, integrating historical bug knowledge from mailing list discussions.

**Direct Hypothesis.** As LLMs already possess a foundational understanding of the Linux kernel from extensive pre-training, a straightforward expansion approach is to fully leverage the intrinsic knowledge of models. To this end, we design prompts that instruct the model to generate plausible potential causes, and rank these causes based on their estimated likelihood of being responsible for the bug. To ensure the practicality of each hypothesized cause, the LLM is also required to propose a corresponding fix and identify the specific files that would need modification. We then extract the predicted target files, preserving their original ranking from the associated causes. Detailed prompts are in Appendix E.2.

**Mail-Augmented Hypothesis.** Relying solely on the intrinsic capabilities of LLMs is insufficient, as general-purpose models still lack in-depth and domain-specific knowledge of Linux kernel. To address this limitation, we incorporate historical bug knowledge from the Linux kernel mailing list (LKML) (Kernel.org, 2025b). The LKML is the communication channel among Linux kernel developers, including massive emails discussing bugs, patches, and diverse topics on maintaining Linux kernel. Specifically, we adopt a Retrieval-Augmented Generation (RAG) approach, using mailing list data as an external knowledge base to provide more comprehensive and diverse bug causes in Linux kernel.

**Mail Collection.** To construct the kernel knowledge base, we first collect emails from the LKML. We retain only emails that include patches, as these are more likely to involve discussions of bug fixes or feature implementations, providing useful context for FL. To ensure quality, we discard non-atomic patches modifying over 10 files, as these typically represent merged changes. Additionally, to avoid potential data leakage, we exclude any emails containing external URLs or the keyword “bugzilla”.

**Mail Retrieval.** We adopt a hierarchical retrieval strategy: (1) restrict the search space to only emails linked to code files predicted by agents, and (2) reformulate noisy bug reports(e.g., hexadecimal logs) into four key dimensions—bug behavior, potential causes, expected behavior, and possible solutions. We then apply BM25 (Lù, 2024) to retrieve the top-10 relevant emails restricted to those sent before the bug report for temporal consistency.

**Mail-augmented hypothesis.** Using retrieved mails, we prompt LLMs to generate more diverse and informed causes for the bug, which in turn guide the identification of related buggy files. This step is similar to *Direct Hypothesis* but augmented with mail knowledge. Detailed prompts are in Appendix E.2.

### 5.1.3 CANDIDATE INTEGRATION

In this final phase, we consolidate the files predicted by previous two expansion strategies and rank the aggregated candidate files to produce the final FL results.

We adopt a simple yet effective merging strategy. Specifically, for each candidate file  $f$ , we collect its ranks from the three sources:  $R_{dir}(f)$  (Directory-Aware Expansion),  $R_{direct}(f)$  (Direct Hypothesis), and  $R_{mail}(f)$  (Mail-Augmented Hypothesis). If a file does not appear in the results of a particular method, its rank is set to  $\infty$ . We then compute an aggregated score for  $f$  as follows:  $\text{score}(f) = \frac{1}{R_{dir}(f)} + \frac{1}{R_{direct}(f)} + \frac{1}{R_{mail}(f)}$ . Files that achieve better ranks in any individual method receive higher scores, while those consistently ranked highly across methods are further prioritized. All candidate files are sorted by their aggregated scores to produce the initial merged ranking. To further refine this list, the LLM is prompted to re-rank the files based on the semantic correspondence between their path and bug report.

## 5.2 EXPERIMENTAL SETUP

**Baselines.** To evaluate the effectiveness of  $\text{LINUXFL}^+$  in improving existing agents, we apply  $\text{LINUXFL}^+$  to refine the prediction outputs of recent agents (i.e., SWE-Agent, AutoCodeRover, and Agentless) on  $\text{LINUXFLBENCH}$ .

**Implementation Details.** We leverage GPT-4o (OpenAI, 2024) (gpt-4o-2024-08-06) and the open-source Qwen3-32B (Yang et al., 2025a) as the backbone models for implementing  $\text{LINUXFL}^+$ . We configure the model temperature as 0 to ensure relatively deterministic outputs with other parameters as default settings.

## 5.3 RESULTS AND ANALYSIS

### 5.3.1 OVERALL PERFORMANCE

Table 4: Evaluation results of  $\text{LINUXFL}^+$ .

Methods	Recall@1	Recall@5	Recall@10	MRR
SWE-Agent	0.416	0.552	0.584	0.476
- w/ $\text{LINUXFL}^+$ (GPT-4o)	0.524 (+0.108)	0.720 (+0.168)	0.768 (+0.184)	0.610 (+0.134)
- w/ $\text{LINUXFL}^+$ (Qwen3-32B)	0.476 (+0.060)	0.664 (-0.112)	0.704 (+0.120)	0.558 (+0.082)
AutoCodeRover	0.388	0.496	0.496	0.435
- w/ $\text{LINUXFL}^+$ (GPT-4o)	0.500 (+0.112)	0.712 (+0.216)	0.744 (+0.248)	0.589 (+0.154)
- w/ $\text{LINUXFL}^+$ (Qwen3-32B)	0.440 (+0.052)	0.664 (-0.168)	0.720 (+0.224)	0.539 (+0.105)
Agentless	0.368	0.492	0.504	0.419
- w/ $\text{LINUXFL}^+$ (GPT-4o)	0.440 (+0.072)	0.684 (+0.192)	0.724 (+0.220)	0.549 (+0.130)
- w/ $\text{LINUXFL}^+$ (Qwen3-32B)	0.432 (+0.064)	0.652 (+0.160)	0.688 (+0.184)	0.525 (+0.106)

Table 5: Cost of  $\text{LINUXFL}^+$ .

Methods	# Tokens	\$ Cost
SWE-Agent	72.4 K	0.194
- w/ $\text{LINUXFL}^+$	14.0 K	0.041
AutoCodeRover	206.6 K	0.560
- w/ $\text{LINUXFL}^+$	11.8 K	0.035
Agentless	150.2 K	0.396
- w/ $\text{LINUXFL}^+$	15.3 K	0.044

Table 4 presents the improvements of  $\text{LINUXFL}^+$  on all studied agents.

**Effectiveness.**  $\text{LINUXFL}^+$  exhibits strong performance in enhancing the FL capabilities of agents, as evidenced by substantial improvement across all evaluation metrics. For example, when applied to SWE-Agent with GPT-4o, Recall@10 increases from 0.584 to 0.768, an absolute gain of 18.4 percentage points. Moreover, Recall@1 improves by 10.8 percentage points (from 0.416 to 0.524). The improvement indicates the effectiveness of the expansion strategies of  $\text{LINUXFL}^+$ , which successfully recover the buggy files missed by existing agents.

**Generalizability.**  $\text{LINUXFL}^+$  consistently enhances performance across all state-of-the-art agents and remains effective with different LLMs. Notably, agents with relatively weaker baselines, such as AutoCodeRover and Agentless, achieve performance comparable to SWE-Agent once integrated with  $\text{LINUXFL}^+$ . Furthermore,  $\text{LINUXFL}^+$  yields consistent gains even when applied to smaller open-source models such as Qwen3-32B. These results highlight the strong generalizability of  $\text{LINUXFL}^+$  and its effectiveness across agent-based approaches with diverse baseline strengths and LLM capacities.

**Ablation study.** We perform an ablation study to investigate the contribution of each component in  $\text{LINUXFL}^+$ . In particular, we find all the expansion strategies, i.e., directory-aware expansion and potential causes expansion (with either direct or mail-augmented hypothesis) can improve the FL effectiveness of agents. Detailed results can be found in Appendix F.

**Cost-efficiency.** Table 5 presents the cost of applying  $\text{LINUXFL}^+$  on  $\text{LINUXFLBENCH}$  with GPT-4o. As shown, while  $\text{LINUXFL}^+$  achieves strong performance, it incurs only a modest additional cost. On average, the total number of tokens used per task by  $\text{LINUXFL}^+$  ranges from 11.8k to 15.3k, resulting in an estimated cost of approximately \$0.04. This is roughly one-tenth of the cost incurred by agent-based baselines. The primary cost of  $\text{LINUXFL}^+$

432 stems from its use of email content. These results suggest that  $\text{LINUXFL}^+$  can substantially  
 433 enhance FL for the large-scale system Linux kernel at a affordable cost.  
 434

435 In summary, by enhancing the capabilities of existing agents,  $\text{LINUXFL}^+$  facilitates more  
 436 accurate FL with minimal costs. Our findings underscore the potential of  $\text{LINUXFL}^+$  to  
 437 significantly support software maintenance tasks in Linux kernel.  
 438

### 5.3.2 METHOD-LEVEL FL

440 To further evaluate  $\text{LINUXFL}^+$  at a finer granularity, we extend our evaluation to method-  
 441 level FL. Specifically, given the buggy files predicted by  $\text{LINUXFL}^+$ , we proceed to identify  
 442 buggy methods by prompting LLMs with a skeleton representation of each file, following  
 443 prior work (Xia et al., 2024). This skeleton format preserves only function signatures and  
 444 comments, which reduces input length while retaining essential context. The LLM is then  
 445 prompted to identify the top-k ( $k=10$ ) most relevant functions. Given the characteristics  
 446 of the C language, we define method-level elements as functions, structures, and other code  
 447 blocks. We consider the methods that are modified in the developer-committed patches as  
 448 the ground truth for buggy methods.  
 449

450 Table 6 presents the method-level FL  
 451 results of existing agents and those  
 452 enhanced with  $\text{LINUXFL}^+$  based on  
 453 GPT-4o. Overall,  $\text{LINUXFL}^+$  can  
 454 consistently improve agents in method-  
 455 level FL for Linux kernel. All three  
 456 agent baselines exhibit low Recall@1  
 457 (below 0.1), while  $\text{LINUXFL}^+$  consis-  
 458 tently improves this metric beyond 0.1.  
 459 The improvements are more pronounced in other  
 460 metrics, e.g., for Recall@10,  $\text{LINUXFL}^+$  enhances all baselines by more than 0.09. While  
 461 localizing finer-grained elements is inherently much more challenging specifically for large  
 462 scale systems like Linux kernel, the overall accuracy at method level remains relatively lower  
 463 than at the file level, highlighting the need for further research in this direction.  
 464

## 6 LIMITATIONS

465 **Limited Evaluation on Different LLMs.** To ensure consistency with prior work (Yang  
 466 et al., 2024; Zhang et al., 2024; Xia et al., 2024) and facilitate fair comparison of agent per-  
 467 formance across SWE-bench and  $\text{LINUXFLBENCH}$ , most experiments in this study employed  
 468 GPT-4o as the backbone LLM. To address this limitation, we also validated the effectiveness  
 469 of  $\text{LINUXFL}^+$  with open-source Qwen-32B. While  $\text{LINUXFL}^+$  consistently yields significant  
 470 improvements, its performance with other LLMs was only briefly explored.  
 471

472 **Rough Usage of Mail Data.**  $\text{LINUXFL}^+$  leverages external knowledge from Linux kernel  
 473 mailing list to enhance FL. Given the richness of email content, this resource may also  
 474 contain irrelevant or outdated discussions, though it is valuable. To mitigate this, we employ  
 475 various filtering and querying strategies, such as query reformulation and heuristic filtering,  
 476 to improve the quality of retrieved emails. Despite these efforts, there is still room for further  
 477 enhancement. Future work could explore more sophisticated approaches to effectively utilize  
 478 mailing list knowledge for improved software maintenance tasks on the Linux kernel.  
 479

## 7 CONCLUSION

480 In this work, we introduce  $\text{LINUXFLBENCH}$ , a new and challenging software engineering  
 481 benchmark designed for fault localization in the Linux kernel. To assess the effectiveness of  
 482 existing LLM agents in complex software systems, we conduct an empirical study using  $\text{LIN-}$   
 483  $\text{UXFLBENCH}$ . Initial results reveal that these agents struggle to accurately identify buggy  
 484 files. To address this challenge, we propose  $\text{LINUXFL}^+$ , a fault localization enhancement  
 485 framework that leverages diverse expansion strategies to enrich candidate selection. Our  
 486 approach demonstrates substantial improvements in localization performance.  
 487

Table 6: Method-level FL results.

Methods	Recall@1	Recall@5	Recall@10	MRR
SWE-Agent	0.089	0.178	0.214	0.170
- w/ $\text{LINUXFL}^+$	0.138	0.271	0.326	0.253
AutoCodeRover	0.042	0.088	0.094	0.077
- w/ $\text{LINUXFL}^+$	0.137	0.292	0.349	0.259
Agentless	0.098	0.147	0.179	0.162
- w/ $\text{LINUXFL}^+$	0.111	0.229	0.269	0.217

486  
487

## REPRODUCIBILITY STATEMENT

488  
489  
490  
491

We have taken steps to ensure the reproducibility of our results. All experimental settings are thoroughly described in the main text and appendix, and both the data and source code used in our work are made available at <https://anonymous.4open.science/r/LinuxFLBench-7C0D>.

492

## ETHICS STATEMENT

494  
495  
496  
497  
498

All authors of this work have adhered to the ICLR Code of Ethics. Human involvement in this study was limited to the manual inspection step during the construction of our benchmark, LINUXFLBENCH. This task was reviewed and approved by the Institutional Review Board (IRB) at our institution. All participants were compensated at a rate of \$15 per hour.

499

500  
501

## REFERENCES

502  
503  
504  
505  
506

Rui Abreu, Peter Zoeteweij, and Arjan J. C. van Gemund. An evaluation of similarity coefficients for software fault localization. In *12th IEEE Pacific Rim International Symposium on Dependable Computing (PRDC 2006), 18-20 December, 2006, University of California, Riverside, USA*, pp. 39–46. IEEE Computer Society, 2006. doi: 10.1109/PRDC.2006.18. URL <https://doi.org/10.1109/PRDC.2006.18>.

507  
508  
509  
510  
511  
512

Abdulaziz Alaboudi and Thomas D. LaToza. Hypothesizer: A hypothesis-based debugger to find and test debugging hypotheses. In Sean Follmer, Jeff Han, Jürgen Steimle, and Nathalie Henry Riche (eds.), *Proceedings of the 36th Annual ACM Symposium on User Interface Software and Technology, UIST 2023, San Francisco, CA, USA, 29 October 2023- 1 November 2023*, pp. 73:1–73:14. ACM, 2023. doi: 10.1145/3586183.3606781. URL <https://doi.org/10.1145/3586183.3606781>.

513  
514  
515  
516  
517  
518  
519

Tegawendé F. Bissyandé, Laurent Réveillère, Julia Lawall, and Gilles Muller. Diagnosys: automatic generation of a debugging interface to the linux kernel. In Michael Goedicke, Tim Menzies, and Motoshi Saeki (eds.), *IEEE/ACM International Conference on Automated Software Engineering, ASE'12, Essen, Germany, September 3-7, 2012*, pp. 60–69. ACM, 2012. doi: 10.1145/2351676.2351686. URL <https://doi.org/10.1145/2351676.2351686>.

520  
521

Martin Bligh and Andy P Whitcroft. Fully automated testing of the linux kernel. In *Proceedings of the Linux Symposium*, volume 1, pp. 113–125. Citeseer, 2006.

522  
523  
524  
525  
526  
527  
528

Yu Chen, Fengguang Wu, Kuanlong Yu, Lei Zhang, Yuheng Chen, Yang Yang, and Junjie Mao. Instant bug testing service for linux kernel. In *10th IEEE International Conference on High Performance Computing and Communications & 2013 IEEE International Conference on Embedded and Ubiquitous Computing, HPCC/EUC 2013, Zhangjiajie, China, November 13-15, 2013*, pp. 1860–1865. IEEE, 2013. doi: 10.1109/HPCC.AND.EUC.2013.347. URL <https://doi.org/10.1109/HPCC.and.EUC.2013.347>.

529  
530  
531  
532

Zhaoling Chen, Xiangru Tang, Gangda Deng, Fang Wu, Jialong Wu, Zhiwei Jiang, Viktor K. Prasanna, Arman Cohan, and Xingyao Wang. Locagent: Graph-guided LLM agents for code localization. *CoRR*, abs/2503.09089, 2025. doi: 10.48550/ARXIV.2503.09089. URL <https://doi.org/10.48550/arXiv.2503.09089>.

533  
534  
535  
536  
537

Agnieszka Ciborowska and Kostadin Damevski. Fast changeset-based bug localization with BERT. In *44th IEEE/ACM 44th International Conference on Software Engineering, ICSE 2022, Pittsburgh, PA, USA, May 25-27, 2022*, pp. 946–957. ACM, 2022. doi: 10.1145/3510003.3510042. URL <https://doi.org/10.1145/3510003.3510042>.

538  
539

Jake Edge. A kernel debugger in python: drgn, 2019. <https://lwn.net/Articles/789641/>.

Google. Syzkaller, 2025. <https://syzkaller.appspot.com>.

540 Ruida Hu, Chao Peng, Jingyi Ren, Bo Jiang, Xiangxin Meng, Qinyun Wu, Pengfei Gao,  
 541 Xincheng Wang, and Cuiyun Gao. A real-world benchmark for evaluating fine-grained  
 542 issue solving capabilities of large language models, 2024. URL <https://arxiv.org/abs/2411.18019>.

543

544 Xuan Huo, Ferdinand Thung, Ming Li, David Lo, and Shu-Ting Shi. Deep transfer bug  
 545 localization. *IEEE Trans. Software Eng.*, 47(7):1368–1380, 2021. doi: 10.1109/TSE.2019.  
 546 2920771. URL <https://doi.org/10.1109/TSE.2019.2920771>.

547

548 Dae R. Jeong, Minkyu Jung, Yoochan Lee, Byoungyoung Lee, Insik Shin, and Youngjin  
 549 Kwon. Diagnosing kernel concurrency failures with AITIA. In Giuseppe Antonio Di Luna,  
 550 Leonardo Querzoni, Alexandra Fedorova, and Dushyanth Narayanan (eds.), *Proceedings  
 551 of the Eighteenth European Conference on Computer Systems, EuroSys 2023, Rome, Italy,  
 552 May 8-12, 2023*, pp. 94–110. ACM, 2023. doi: 10.1145/3552326.3567486. URL <https://doi.org/10.1145/3552326.3567486>.

553

554 Carlos E. Jimenez, John Yang, Alexander Wettig, Shunyu Yao, Kexin Pei, Ofir Press, and  
 555 Karthik R. Narasimhan. Swe-bench: Can language models resolve real-world github  
 556 issues? In *The Twelfth International Conference on Learning Representations, ICLR 2024,  
 557 Vienna, Austria, May 7-11, 2024*. OpenReview.net, 2024. URL <https://openreview.net/forum?id=VTF8yNQM66>.

558

559 René Just, Darioush Jalali, and Michael D. Ernst. Defects4j: a database of existing faults  
 560 to enable controlled testing studies for java programs. In Corina S. Pasareanu and Darko  
 561 Marinov (eds.), *International Symposium on Software Testing and Analysis, ISSTA '14,  
 562 San Jose, CA, USA - July 21 - 26, 2014*, pp. 437–440. ACM, 2014. doi: 10.1145/2610384.  
 563 2628055. URL <https://doi.org/10.1145/2610384.2628055>.

564

565 Sungmin Kang, Gabin An, and Shin Yoo. A quantitative and qualitative evaluation of llm-  
 566 based explainable fault localization. *Proc. ACM Softw. Eng.*, 1(FSE):1424–1446, 2024.  
 567 doi: 10.1145/3660771. URL <https://doi.org/10.1145/3660771>.

568

569 Kernel.org. Kernel.org bugzilla, 2025a. <https://bugzilla.kernel.org>.

570

571 Kernel.org. Linux kernel mailing list, 2025b. <https://lore.kernel.org>.

572

573 Kernel.org. Kernel.org rsync, 2025c. <http://rsync.kernel.org/pub/linux/kernel>.

574

575 Lucas Layman, Madeline Diep, Meiyappan Nagappan, Janice Singer, Robert DeLine, and  
 576 Gina Venolia. Debugging revisited: Toward understanding the debugging needs of con-  
 577 temporary software developers. In *2013 ACM / IEEE International Symposium on Empir-  
 578 ical Software Engineering and Measurement, Baltimore, Maryland, USA, October 10-11,  
 2013*, pp. 383–392. IEEE Computer Society, 2013. doi: 10.1109/ESEM.2013.43. URL  
<https://doi.org/10.1109/ESEM.2013.43>.

579

580 Hanzhi Liu, Yanyan Jiang, and Chang Xu. Understanding the linux kernel, visually. In  
 581 *Proceedings of the Twentieth European Conference on Computer Systems, EuroSys 2025,  
 582 Rotterdam, The Netherlands, 30 March 2025 - 3 April 2025*, pp. 1044–1060. ACM, 2025.  
 583 doi: 10.1145/3689031.3696095. URL <https://doi.org/10.1145/3689031.3696095>.

584

585 Junwei Liu, Kaixin Wang, Yixuan Chen, Xin Peng, Zhenpeng Chen, Lingming Zhang, and  
 586 Yiling Lou. Large language model-based agents for software engineering: A survey. *CoRR*,  
 587 abs/2409.02977, 2024. doi: 10.48550/ARXIV.2409.02977. URL <https://doi.org/10.48550/arXiv.2409.02977>.

588

589 Yiling Lou, Qihao Zhu, Jinhao Dong, Xia Li, Zeyu Sun, Dan Hao, Lu Zhang, and Lingming  
 590 Zhang. Boosting coverage-based fault localization via graph-based representation learning.  
 591 In Diomidis Spinellis, Georgios Gousios, Marsha Chechik, and Massimiliano Di Penta  
 592 (eds.), *ESEC/FSE '21: 29th ACM Joint European Software Engineering Conference and  
 593 Symposium on the Foundations of Software Engineering, Athens, Greece, August 23-28,  
 2021*, pp. 664–676. ACM, 2021. doi: 10.1145/3468264.3468580. URL <https://doi.org/10.1145/3468264.3468580>.

594 Xing Han Lù. Bm25s: Orders of magnitude faster lexical search via eager sparse scoring,  
 595 2024. URL <https://arxiv.org/abs/2407.03618>.

596

597 Alex Mathai, Chenxi Huang, Petros Maniatis, Aleksandr Nogikh, Franjo Ivancic, Junfeng  
 598 Yang, and Baishakhi Ray. Kgym: A platform and dataset to benchmark large language  
 599 models on linux kernel crash resolution. *CoRR*, abs/2407.02680, 2024. doi: 10.48550/  
 600 ARXIV.2407.02680. URL <https://doi.org/10.48550/arXiv.2407.02680>.

601

602 Samuel Miserendino, Michele Wang, Tejal Patwardhan, and Johannes Heidecke. Swe-lancer:  
 603 Can frontier llms earn \$1 million from real-world freelance software engineering? *CoRR*,  
 604 abs/2502.12115, 2025. doi: 10.48550/ARXIV.2502.12115. URL <https://doi.org/10.48550/arXiv.2502.12115>.

605

606 Amr Mansour Mohsen, Hesham A. Hassan, Khaled Wassif, Ramadan Moawad, and Soha  
 607 Makady. Enhancing bug localization using phase-based approach. *IEEE Access*, 11:  
 608 35901–35913, 2023. doi: 10.1109/ACCESS.2023.3265731. URL <https://doi.org/10.1109/ACCESS.2023.3265731>.

609

610 OpenAI. Hello gpt-4o, 2024. URL <https://openai.com/index/hello-gpt-4o>.

611

612 Yihao Qin, Shangwen Wang, Yiling Lou, Jinhao Dong, Kaixin Wang, Xiaoling Li, and  
 613 Xiaoguang Mao. Agentfl: Scaling llm-based fault localization to project-level context.  
 614 *CoRR*, abs/2403.16362, 2024. doi: 10.48550/ARXIV.2403.16362. URL <https://doi.org/10.48550/arXiv.2403.16362>.

615

616 Nils Reimers and Iryna Gurevych. Sentence-bert: Sentence embeddings using siamese bert-  
 617 networks. In Kentaro Inui, Jing Jiang, Vincent Ng, and Xiaojun Wan (eds.), *Proceedings*  
 618 *of the 2019 Conference on Empirical Methods in Natural Language Processing and the 9th*  
 619 *International Joint Conference on Natural Language Processing, EMNLP-IJCNLP 2019,*  
 620 *Hong Kong, China, November 3-7, 2019*, pp. 3980–3990. Association for Computational  
 621 Linguistics, 2019. doi: 10.18653/V1/D19-1410. URL <https://doi.org/10.18653/v1/D19-1410>.

622

623 Stephen E. Robertson, Steve Walker, and Micheline Hancock-Beaulieu. Large test collection  
 624 experiments on an operational, interactive system: Okapi at TREC. *Inf. Process. Manag.*,  
 625 31(3):345–360, 1995. doi: 10.1016/0306-4573(94)00051-4. URL [https://doi.org/10.1016/0306-4573\(94\)00051-4](https://doi.org/10.1016/0306-4573(94)00051-4).

626

627 Ripon K. Saha, Matthew Lease, Sarfraz Khurshid, and Dewayne E. Perry. Improving bug  
 628 localization using structured information retrieval. In Ewen Denney, Tevfik Bultan, and  
 629 Andreas Zeller (eds.), *2013 28th IEEE/ACM International Conference on Automated*  
 630 *Software Engineering, ASE 2013, Silicon Valley, CA, USA, November 11-15, 2013*, pp.  
 631 345–355. IEEE, 2013. doi: 10.1109/ASE.2013.6693093. URL <https://doi.org/10.1109/ASE.2013.6693093>.

632

633 Ripon K. Saha, Julia Lawall, Sarfraz Khurshid, and Dewayne E. Perry. On the effectiveness  
 634 of information retrieval based bug localization for C programs. In *30th IEEE International*  
 635 *Conference on Software Maintenance and Evolution, Victoria, BC, Canada, September*  
 636 *29 - October 3, 2014*, pp. 161–170. IEEE Computer Society, 2014. doi: 10.1109/ICSME.  
 637 2014.38. URL <https://doi.org/10.1109/ICSME.2014.38>.

638

639 Lucas Serrano, Van-Anh Nguyen, Ferdian Thung, Lingxiao Jiang, David Lo, Julia Lawall,  
 640 and Gilles Muller. SPINFER: inferring semantic patches for the linux kernel. In Ada  
 641 Gavrilovska and Erez Zadok (eds.), *Proceedings of the 2020 USENIX Annual Technical*  
 642 *Conference, USENIX ATC 2020, July 15-17, 2020*, pp. 235–248. USENIX Association,  
 643 2020. URL <https://www.usenix.org/conference/atc20/presentation/serrano>.

644

645 Bunyamin Sisman, Shayan A. Akbar, and Avinash C. Kak. Exploiting spatial code proximity  
 646 and order for improved source code retrieval for bug localization. *J. Softw. Evol. Process.*,  
 647 29(1), 2017. doi: 10.1002/SMR.1805. URL <https://doi.org/10.1002/smj.1805>.

648

649 SWE-bench. Swe-bench leaderboards, 2025. <https://www.swebench.com/>.

648 The Linux Foundation. Linux kernel history report 2020, 2020.  
 649 URL [https://www.linuxfoundation.org/resources/publications/  
 650 linux-kernel-history-report-2020](https://www.linuxfoundation.org/resources/publications/linux-kernel-history-report-2020).

651 Shaowei Wang and David Lo. Version history, similar report, and structure: putting them  
 652 together for improved bug localization. In Chanchal K. Roy, Andrew Begel, and Leon  
 653 Moonen (eds.), *22nd International Conference on Program Comprehension, ICPC 2014,  
 654 Hyderabad, India, June 2-3, 2014*, pp. 53–63. ACM, 2014. doi: 10.1145/2597008.2597148.  
 655 URL <https://doi.org/10.1145/2597008.2597148>.

656 Chu-Pan Wong, Yingfei Xiong, Hongyu Zhang, Dan Hao, Lu Zhang, and Hong Mei. Boosting  
 657 bug-report-oriented fault localization with segmentation and stack-trace analysis. In *30th  
 658 IEEE International Conference on Software Maintenance and Evolution, Victoria, BC,  
 659 Canada, September 29 - October 3, 2014*, pp. 181–190. IEEE Computer Society, 2014a.  
 660 doi: 10.1109/ICSME.2014.40. URL <https://doi.org/10.1109/ICSME.2014.40>.

661 W. Eric Wong, Vidroha Debroy, Ruizhi Gao, and Yihao Li. The dstar method for effective  
 662 software fault localization. *IEEE Trans. Reliab.*, 63(1):290–308, 2014b. doi: 10.1109/TR.  
 663 2013.2285319. URL <https://doi.org/10.1109/TR.2013.2285319>.

664 Chunqiu Steven Xia, Yinlin Deng, Soren Dunn, and Lingming Zhang. Agentless: De-  
 665 mystifying llm-based software engineering agents. *CoRR*, abs/2407.01489, 2024. doi:  
 666 10.48550/ARXIV.2407.01489. URL <https://doi.org/10.48550/arXiv.2407.01489>.

667 Xin Xia and David Lo. *Information Retrieval-Based Techniques for Software Fault Local-  
 668 ization*, pp. 365–391. 2023. doi: 10.1002/9781119880929.ch8.

669 An Yang, Anfeng Li, Baosong Yang, Beichen Zhang, Binyuan Hui, Bo Zheng, Bowen Yu,  
 670 Chang Gao, Chengan Huang, Chenxu Lv, Chujie Zheng, Dayiheng Liu, Fan Zhou, Fei  
 671 Huang, Feng Hu, Hao Ge, Haoran Wei, Huan Lin, Jialong Tang, Jian Yang, Jianhong  
 672 Tu, Jianwei Zhang, Jian Yang, Jiaxi Yang, Jingren Zhou, Junyang Lin, Kai Dang, Keqin  
 673 Bao, Kexin Yang, Le Yu, Lianghao Deng, Mei Li, Mingfeng Xue, Mingze Li, Pei Zhang,  
 674 Peng Wang, Qin Zhu, Rui Men, Ruize Gao, Shixuan Liu, Shuang Luo, Tianhao Li, Tianyi  
 675 Tang, Wenbiao Yin, Xingzhang Ren, Xinyu Wang, Xinyu Zhang, Xuancheng Ren, Yang  
 676 Fan, Yang Su, Yichang Zhang, Yinger Zhang, Yu Wan, Yuqiong Liu, Zekun Wang, Zeyu  
 677 Cui, Zhenru Zhang, Zhipeng Zhou, and Zihan Qiu. Qwen3 technical report. *CoRR*,  
 678 abs/2505.09388, 2025a. doi: 10.48550/ARXIV.2505.09388. URL <https://doi.org/10.48550/arXiv.2505.09388>.

679 Chenyuan Yang, Zijie Zhao, and Lingming Zhang. Kernelgpt: Enhanced kernel fuzzing  
 680 via large language models. In Lieven Eeckhout, Georgios Smaragdakis, Katai Liang,  
 681 Adrian Sampson, Martha A. Kim, and Christopher J. Rossbach (eds.), *Proceedings of the  
 682 30th ACM International Conference on Architectural Support for Programming Languages  
 683 and Operating Systems, Volume 2, ASPLOS 2025, Rotterdam, Netherlands, 30 March  
 684 - 3 April 2025*, pp. 560–573. ACM, 2025b. doi: 10.1145/3676641.3716022. URL  
 685 <https://doi.org/10.1145/3676641.3716022>.

686 John Yang, Carlos E. Jimenez, Alexander Wettig, Kilian Lieret, Shunyu Yao, Karthik  
 687 Narasimhan, and Ofir Press. Swe-agent: Agent-computer interfaces enable automated  
 688 software engineering. *CoRR*, abs/2405.15793, 2024. doi: 10.48550/ARXIV.2405.15793.  
 689 URL <https://doi.org/10.48550/arXiv.2405.15793>.

690 Yunlong Zhang, Haifeng Ruan, Zhiyu Fan, and Abhik Roychoudhury. Autocoderover: Au-  
 691 tonomous program improvement. In Maria Christakis and Michael Pradel (eds.), *Proceed-  
 692 ings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Anal-  
 693 ysis, ISSTA 2024, Vienna, Austria, September 16-20, 2024*, pp. 1592–1604. ACM, 2024.  
 694 doi: 10.1145/3650212.3680384. URL <https://doi.org/10.1145/3650212.3680384>.

695 Jian Zhou, Hongyu Zhang, and David Lo. Where should the bugs be fixed? more accurate  
 696 information retrieval-based bug localization based on bug reports. In Martin Glinz, Gail C.  
 697 Murphy, and Mauro Pezzè (eds.), *34th International Conference on Software Engineering,  
 698 ICSE 2012, June 2-9, 2012, Zurich, Switzerland*, pp. 14–24. IEEE Computer Society,

702 2012. doi: 10.1109/ICSE.2012.6227210. URL <https://doi.org/10.1109/ICSE.2012.6227210>.

704

705

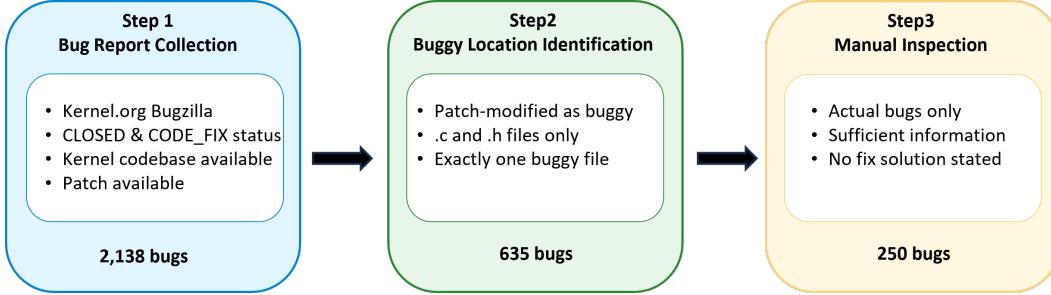
## 706 A APPENDIX

707

## 708 B ADDITIONAL DETAILS OF LINUXFLBENCH

### 710 B.1 CONSTRUCTION PIPELINE OF LINUXFLBENCH

712



722 Figure 5: Construction pipeline of LINUXFLBENCH.

724

725 Fig. 5 illustrates the pipeline for constructing LINUXFLBENCH, which consists of three  
726 main phases: *Bug Report Collection*, *Buggy Location Identification*, and *Manual Inspection*.  
727 Through this process, we curate a total of 250 high-quality fault localization tasks.

728

729

### B.2 A SAMPLE KERNEL BUG FROM LINUXFLBENCH

730

731

We collect Linux kernel bugs (as shown in Fig. 6) from the reported and fixed bugs on  
Kernel.org Bugzilla. For each bug, the key information includes:

732

1. **Title:** The summary title of the bug report.
2. **Description:** A human-written description of the bug, which may include various types of information such as observed buggy behavior, reproduction steps in natural language, system logs, or call traces.
3. **Product:** The product category to which the bug is assigned.
4. **Component:** The specific component within the product affected by the bug.
5. **Hardware:** The hardware configuration on which the bug was observed.
6. **Kernel Version:** The version of the Linux kernel in which the bug occurred (e.g., 5.6.7).
7. **Paths:** The paths of the buggy files, extracted from the golden patch that fixes the bug.
8. **Modified Functions:** The method-level code elements modified by the patch.

748

749

### B.3 COMPARISON WITH SWE-BENCH LITE

750

751

752

753

754

755

To investigate the complexity and challenge of our benchmark, we conducted a series of quantitative analyses in comparison with SWE-bench Lite. The results are presented in Table 7 and summarized as follows.

**File length.** We compared the sizes of buggy files and patches (measured in lines of code). As shown in the table, LINUXFLBENCH contains substantially larger buggy files and patches, indicating higher complexity and greater difficulty for fault localization.

			Description
756	<b>Bug 207835 - ACPI video backlight</b>	<b>Paul Menzel 2020-05-21 21:01:40 UTC</b>	
757	<b>brightness control does not work on</b>		On the Intel GM45/GMA 4500MHD laptop with Debian Sid/unstable and Linux 5.6.7 and GNOME Shell 3.36.2, trying to change the brightness of the internal display using the function keys, GNOME shows the OSD, which seems to have five levels (from acer_wmi?), but the actual brightness does *not* change. There is only `/sys/class/backlight/intel_backlight/brightness` though, and writing values to it seems to work.
758	<b>Acer TravelMate 5735Z by default</b>		
759	<b>Status: CLOSED CODE_FIX</b>		
760	<b>Alias: None</b>		Booting the system with `acpi_backlight=vendor`, exposes `/sys/devices/platform/acer-wmi/backlight/acer-wmi/brightness`, but the behavior is the same.
761	<b>Product: ACPI</b>		
762	<b>Component: Other (show other bugs)</b>		Booting the system with `acpi_backlight=native` or `acpi_backlight=auto`, changing the brightness with the function keys works, and there now seem to be 15 (or 16) levels.
763	<b>Hardware: All Linux</b>		
764	<b>Importance: P1 normal</b>		
765	<b>Assignee: acpi_other</b>		
766	<b>Reported: 2020-05-21 21:01 UTC by Paul Menzel</b>		
767	<b>Modified: 2020-06-30 06:49 UTC (History)</b>		
768	<b>CC List: 3 users (show)</b>		
769	<b>Kernel Version: 5.6.7</b>		
770	<b>Regression: No</b>		
771	<b>Paths: drivers/acpi/video_detect.c</b>		
772	<b>Modified Functions: video_detect_dmi_table</b>		
773			
774			
775			
776			

Figure 6: A sample kernel bug from LINUXFLBENCH.

Table 7: Comparison between LINUXFLBENCH and SWE-bench Lite

Dataset	File Statistics (Lines)				Location Information	Type	Stack Trace	Directory
	Mean	Buggy	Mean	Patch				
LinuxFLBench	2050.08	22.32	20142	572	0.568	0.008	14.33	16 files
SWE-bench Lite	1211.13	10.13	8237	76	0.487	0.160	5.73	8 files

**Directory size.** To indirectly capture the scope of potentially relevant files, we analyze the average number of files per directory. LINUXFLBENCH has an average of 16 files per directory, while SWE-bench Lite has only 8, suggesting that fault localization in our benchmark requires reasoning over larger and more interconnected contexts.

**Stack trace length.** Some bug reports in our benchmark include stack traces, which reflect the propagation paths of underlying bugs. On average, LINUXFLBENCH reports contain 14.33 functions per stack trace, compared to 5.73 in SWE-bench Lite. This suggests that bugs in our dataset involve longer propagation chains and more complex interactions.

**Location information.** Following the methodology of (Xia et al., 2024), we analyze the overlap between issue descriptions and file location information. Specifically, we distinguish between (i) straightforward bugs, where the full file path is explicitly mentioned in the description, and (ii) challenging bugs, where no related keywords appear. The results show that location information in LINUXFLBENCH is significantly sparser than in SWE-bench Lite, further increasing the difficulty of fault localization.

## C DETAILS OF BASELINES USED IN THIS PAPER

### C.1 STUDIED LLM AGENTS.

This paper evaluates three SOTA LLM agents: SWE-Agent (Yang et al., 2024), AutoCodeRover (Zhang et al., 2024), and Agentless (Xia et al., 2024).

- **SWE-Agent.** SWE-Agent navigates the entire repository to identify the bug’s location. To adapt this system to our benchmark, we modified the task description in the system prompt, specifying the objective as identifying suspicious files, while keeping the rest of the framework unchanged.
- **AutoCodeRover.** AutoCodeRover locates suspicious Python files based on the give GitHub issues through advanced code search techniques. We extended its functionality to support C/C++ projects by replacing its parser with *ctags*, enabling it to perform code search within Linux kernel codebases. Moreover, we also manually

810  
811  
812  
813  
814  
815  
816  
817  
818  
819  
820  
821  
822  
823  
824  
825  
826  
827  
828  
829  
830  
831  
832  
833  
834  
835  
836  
837  
838  
839  
840  
841  
842  
843  
844  
845  
846  
847  
848  
849  
850  
851  
852  
853  
854  
855  
856  
857  
858  
859  
860  
861  
862  
863

sampled and inspected the trajectories of the agent to ensure proper handling of C language features.

- **Agentless.** Agentless identifies the suspicious files based on a concise representation of the repository structure. Given the vast number of files in the Linux kernel, we partition the repository structure into manageable portions by folder and feed them to the LLM in multiple iterations.

## C.2 IR-BASED BASELINES.

To further investigate the effectiveness of agent-based methods, we also selected traditional IR-based baselines for comparison. Specifically, we included the classic methods BugLocator (Zhou et al., 2012) and BLUiR (Saha et al., 2013), along with widely used IR techniques such as BM25 (Robertson et al., 1995) and Sentence-BERT (Reimers & Gurevych, 2019).

- **BM25.** BM25 is one of the most widely used IR methods, and we include it as one of our baselines. BM25 is a bag-of-words retrieval function that ranks a set of documents based on term frequency and inverse document frequency of each document.
- **BugLocator.** BugLocator retrieves buggy files from a codebase by treating a bug report as a query and ranking files based on similarity using a revised vector space model (rVSM). The rVSM method prioritizes longer documents, assuming these files are more likely to contain bugs. Additionally, BugLocator incorporates historical bug fixes to further assess the likelihood of defects in a given file. In this work, we do not leverage this historical bug fix feature of BugLocator due to the unavailability of the necessary data.
- **BLUiR.** BLUiR enhances bug localization by extracting code entities, such as classes, methods, and variable names, from source code files. It calculates the relevance of these entities to both the title and description of a bug report respectively, aiding in the identification of buggy files.
- **SentenceBERT.** SentenceBERT enhances the traditional BERT model by incorporating siamese and triplet network architectures, enabling more efficient semantic search with reduced computational overhead. For our implementation, we utilize the sentence-transformer model *all-MiniLM-L6-v2*.

## D FAILURE CASES OF LLM AGENTS ON LINUXFLBENCH

The suboptimal performance of agent-based methods can be attributed to several limitations, including confusion among related files and insufficient exploration of potential root causes. This section presents representative failure cases to illustrate these limitations.

### D.1 CONFUSION AMONG RELATED FILES

An illustrative case is shown in Fig. 7. In this example, the update of the computer’s battery and AC status involves interactions among the ACPI battery, AC adapter, and the embedded controller (EC). The corresponding drivers for these components all reside in the *drivers/acpi* directory. While different agent baselines identify the files related to the ACPI battery and adapter, they confuse and overlook the deeper component in the bug propagation chain—the EC driver—resulting in incorrect FL.

### D.2 LIMITED EXPLORATION OF POTENTIAL CAUSES

A representative case is provided in Fig. 8. the bug behavior “hangs on shutdown” could stem from various causes, since system shutdown involves a sequence of operations across

864	<b>Bug 6455 - battery and AC status</b>	<b>Jure Repinc 2006-04-28 14:23:51 UTC</b>	<a href="#">Description</a>
865	<b>stops updating - HP Compaq</b>		
866	<b>nx6125</b>		
867			
868	<b>Status: CLOSED CODE_FIX</b>		
869	<b>Alias: None</b>		
870	<b>Product: ACPI</b>		
871	<b>Component: Power-Battery (show other bugs)</b>		
872	<b>Hardware: i386 Linux</b>		
873	<b>Importance: P2 normal</b>		
874	<b>Assignee: Vladimir Lebedev</b>		
875	<b>Reported: 2006-04-28 14:23 UTC by Jure Repinc</b>		
876	<b>Modified: 2006-06-30 17:01 UTC (History)</b>		
877	<b>CC List: 1 user (show)</b>		
878	<b>Kernel Version: 2.6.16.11</b>		
879	<b>Regression: ---</b>		
880	<b>Paths: drivers/acpi/ec.c</b>		
881			

Figure 7: An illustrative case for "Confusion Among Related Files".

882	<b>Bug 3024 - Tyan Thunder K7 hangs</b>	<b>Marcel Weber 2004-07-06 09:23:13 UTC</b>	<a href="#">Description</a>
883	<b>on shutdown / reboot</b>		
884			
885			
886	<b>Status: CLOSED CODE_FIX</b>		
887	<b>Alias: None</b>		
888	<b>Product: ACPI</b>		
889	<b>Component: Power-Off (show other bugs)</b>		
890	<b>Hardware: i386 Linux</b>		
891	<b>Importance: P2 normal</b>		
892	<b>Assignee: acpi_power-off</b>		
893	<b>Reported: 2004-07-06 09:23 UTC by Marcel Weber</b>		
894	<b>Modified: 2004-08-29 10:01 UTC (History)</b>		
895	<b>CC List: 2 users (show)</b>		
896	<b>Kernel Version: 2.6.7</b>		
897	<b>Regression: ---</b>		
898	<b>Paths: drivers/acpi/hardware/hwsleep.c</b>		
899			
900	<b>Top-1 Prediction of baselines</b>		
901	<b>SWE-agent: kernel/power/poweroff.c</b>		
902	<b>AutoCodeRover: drivers/char/watchdog/wdt_pci.c</b>		
903	<b>Agentless: drivers/acpi/power.c</b>		

Figure 8: An illustrative case for "Limited Exploration of Potential Causes".

multiple components. Agents with limited exploration may employ a "depth-first search"-like strategy, focusing on superficially obvious reasons—such as failures in general power-off routines—while overlooking less apparent causes rooted in hardware state handling.

## E PROMPT DESIGN OF LINUXFL<sup>+</sup>

### E.1 PROMPT TEMPLATES IN DIRECTORY-AWARE EXPANSION

LINUXFL<sup>+</sup> re-selects related files within the same directories as the originally predicted files. Given the bug report("bug information") and the list of files("candidate files") in these directories, the LLM is instructed to select the relevant files using the following prompt.

918  
 919     **Prompt for Directory-Aware Expansion:** Please look through the following Linux  
 920     kernel bug report and candidate files, and select a list of files that one would need to edit to  
 921     fix the bug.  
 922  
 923     Here is the information about the bug:  
 924     #### Linux kernel bug report ####  
 925     {*bug information*}  
 926     ####  
 927     Based on the bug provided above, I will present a list of candidate files that may be relevant  
 928     to the bug.  
 929     #### Candidate files ####  
 930     {*candidate files*}  
 931     ####  
 932  
 933     Please select files that are most likely to need modification to fix this bug.  
 934  
 935     Your response should be in the format of a list of file paths, and should be ordered by relevance  
 936     in descending order. Please return at most 10 files.  
 937     #### output example ####  
 938     ['net/ipv6/proc.c', 'net/ipv6/netfilter/ip6\_tables.c']  
 939     ####  
 940  
 941     Please format your response strictly according to the format provided above without com-  
 942     mentary.

## 943     E.2 PROMPT TEMPLATES IN POTENTIAL CAUSE EXPANSION

944  
 945     In the phase of Potential Cause Expansion, LINUXFL<sup>+</sup> instructs the LLM to enumerate  
 946     as many potential causes as possible using two approaches: direct hypothesis and mail-  
 947     augmented hypothesis. The prompt for Mail-Augmented Hypothesis is presented below.  
 948     Given the bug report ("*bug information*") and the retrieved emails ("*mail content*"), the  
 949     LLM is prompted to generate potential causes along with corresponding fix suggestions and  
 950     the affected code files in a specified JSON format. The prompt for Direct Hypothesis is  
 951     similar, but without including the retrieved email content.

952  
 953     **Prompt for Mail-Augmented Hypothesis:** Please review the following Linux kernel  
 954     bug report, and then deduce the possible causes of the bug and provide corresponding code  
 955     files and a potential fix. The bug is known to be related to the kernel code, and the fix should  
 956     involve modifications to kernel code files.

957     Here is the information about the bug:

958     #### Linux kernel bug report ####

959     {*bug information*}  
 960     ####

961  
 962     To assist in your analysis, here are some emails retrieved using BM25 that may be relevant  
 963     to the bug. Use them to inspire and identify additional possible causes:

964     #### Mails ####

965     {*mail content*}  
 966     ####

967  
 968     Based on the bug provided above, please output the possible causes, relevant code files, and  
 969     solutions. Your response should follow the format below.

970     #### Output example ####

```

972 [ { 'cause': 'A description of the potential cause of the bug.', 'code_file': 'Path of the code file  

973 that is most likely related to the bug.', 'fix_solution': 'A short description of the fix solution  

974 to apply in the code file.' }, ... ]
975 #####
976 Please ensure the following:
977
978 - List as many causes as possible, ordered by relevance in descending order, with the most  

979 likely cause first.
980
981 - For each cause, list all relevant code files and their corresponding fixes, but only provide one  

982 code file and one fix per entry.
983
984 - The relevant code file is not necessarily the one causing the bug but should be a file where  

985 the bug can be fixed.
986
987 - The code file should be in the format of "net/ipv6/proc.c".
988
989 - Format your response strictly according to the format provided above without commentary.
990
991
992
993
994
995
996
997
998
999
1000
1001
1002
1003
1004
1005
1006
1007
1008
1009
1010
1011
1012
1013
1014
1015
1016
1017
1018
1019
1020
1021
1022
1023
1024
1025

```

## F ABLATION STUDY

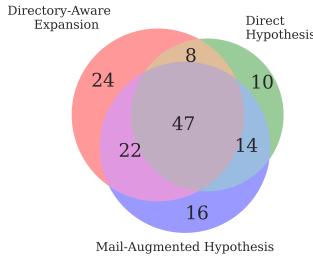


Figure 9: Venn Diagram for Correctly Localized Bugs by Agentless with Different Strategies.

Table 8: Evaluation results of  $\text{LINUXFL}^+$  in different steps.

Methods	Recall@1	Recall@5	Recall@10	MRR
<i>Direct LLM Hypothesis</i>	0.316	0.424	0.424	0.362
<b>SWE-Agent</b>				
- w/ <i>Directory-Aware Expansion</i>	0.448	0.640	0.680	0.527
- w/ <i>Direct Hypothesis</i>	0.440	0.672	0.684	0.537
- w/ <i>Mail-Augmented Hypothesis</i>	0.488	0.632	0.632	0.549
- w/ <i>Merge all</i>	0.516	0.712	0.772	0.601
<b>AutoCodeRover</b>				
- w/ <i>Directory-Aware Expansion</i>	0.424	0.592	0.608	0.492
- w/ <i>Direct Hypothesis</i>	0.444	0.636	0.636	0.528
- w/ <i>Mail-Augmented Hypothesis</i>	0.468	0.576	0.576	0.515
- w/ <i>Merge all</i>	0.476	0.692	0.74	0.570
<b>Agentless</b>				
- w/ <i>Directory-Aware Expansion</i>	0.404	0.584	0.632	0.484
- w/ <i>Direct Hypothesis</i>	0.412	0.596	0.612	0.484
- w/ <i>Mail-Augmented Hypothesis</i>	0.396	0.520	0.520	0.447
- w/ <i>Merge all</i>	0.440	0.672	0.720	0.548

Table 8 presents the results of integrating individual components of  $\text{LINUXFL}^+$  into the baselines based on GPT-4o, examining how each localization phase contributes to the final performance.

**Complementarity of Scaling Strategies.** As shown in Table 8, agent baselines augmented with the three scaling strategies—Directory-Aware Expansion, Direct Hypothesis, and Mail-Augmented Hypothesis—exhibit varying FL performance on  $\text{LINUXFLBENCH}$ . Merging the results from these strategies leads to improved performance, suggesting their complementary nature. To further investigate this characteristic, we present a Venn diagram in Fig. 9, illustrating the top-1 successfully localized bugs achieved by each strategy beyond Agentless. Each strategy independently identifies a substantial number of bugs that the others fail to locate. This highlights the rationale behind our merging approach. The three strategies emphasize different aspects: directory-level structural information from the codebase, intrinsic knowledge from the LLM, and external expertise from historical mailing lists. Integrating these perspectives allows for more effective and robust fault localization.

**Effectiveness of Direct Hypothesis.** The Direct Hypothesis strategy asks LLMs to directly infer buggy files from bug reports, independent of the outputs from agent-based methods. The results of this standalone approach, denoted as Direct LLM Hypothesis, are reported in the first row of Table 8. To further assess its effectiveness, we also evaluate its combination with agent baselines. Specifically, we integrate the predicted files from Direct Hypothesis with the original predictions of each agent, followed by a reranking step. As the results demonstrate, this strategy consistently improves localization performance across various agents. Although the standalone performance of Direct LLM Hypothesis is lower than that of the original agents, it provides complementary information that enriches both

1026 (1) the original agent predictions (as shown in Table 8) and (2) other expansion strategies  
 1027 (as shown in Fig. 9). The primary goal of this strategy is to distill the internal knowledge  
 1028 of LLMs for understanding Linux kernel bugs. By integrating Direct Hypothesis with these  
 1029 agents and expansion strategies, we achieve a more robust and effective fault localization  
 1030 approach.

1031 **Utility of Mail Retrieval.** As discussed in Section 6, LKML may contain irrelevant or outdated discussions.  
 1032 To evaluate our mail retrieval strategy, we first measure the proportion of retrieved emails that contain the correct  
 1033 buggy files. As shown in Table 9, our strategy significantly outperforms direct BM25 retrieval (recall 0.332) on all agents, demonstrating its effectiveness. We further  
 1034 examine the impact on top-10 predictions under the Mail-Augmented Hypothesis by tracking  
 1035 two types of changes: (i) *None* → *Found*, where previously missing buggy files appear, and  
 1036 (ii) *Found* → *Lost*, where files drop out. The results indicate that expansion consistently adds correct files (e.g., 0.136 for AutoCodeRover) while rarely displacing existing  
 1037 ones, confirming that mail retrieval effectively enhances baseline  
 1038

1039 **Benefit of Mail Knowledge.** To investigate the benefits of incorporating knowledge from  
 1040 LKML, we compare baseline methods augmented with the Mail-Augmented Hypothesis  
 1041 against those only using the Direct Hypothesis. As shown in the Table 8, Mail-Augmented  
 1042 Hypothesis consistently outperforms Direct Hypothesis. The latter relies solely on the  
 1043 intrinsic knowledge of LLMs, without utilizing predictions from agent methods, and achieves a  
 1044 recall@1 of only 0.316. In contrast, with the assistance of mail knowledge, Mail-Augmented  
 1045 Hypothesis achieves a recall@1 as high as 0.488, with even more significant improvements  
 1046 observed in recall@10. These results demonstrate that mailing list data can effectively bridge  
 1047 the knowledge gap LLMs face in localizing bugs within the Linux kernel. It is worth noting  
 1048 that the effectiveness of Mail-Augmented Hypothesis varies across different agent methods.  
 1049 For instance, in the case of SWE-Agent, the predicted files facilitate the retrieval of more  
 1050 relevant emails, which provide stronger guidance during cause exploration.  
 1051

1052 **Impact of Re-Ranking.** LINUXFL<sup>+</sup> performs a re-ranking step on the candidates obtained  
 1053 from previous phase. The results in Table 8 demonstrate the effectiveness of this  
 1054 re-ranking process. By comparing the results after merging with the final performance of  
 1055 LINUXFL<sup>+</sup>, further improvement in localization accuracy could be observed. It may stem  
 1056 from the high-quality candidate files provided by the different expansion strategies. With  
 1057 these enriched candidates, even a simple re-ranking allows the model to more easily identify  
 1058 the correct buggy files.  
 1059

## 1060 G HUMAN PARTICIPATION

1061 In this work, human involvement is limited to the Manual Inspection step during the  
 1062 construction of our benchmark, LINUXFLBENCH. This task was approved by the Institutional  
 1063 Review Board (IRB) at our institution. All participants were compensated at a rate of \$15  
 1064 per hour.

1065 During Manual Inspection, each annotator was provided with the following instruction:  
 1066 “*Given the title and description of the bug report, please label the report as ‘yes,’ ‘no,’ or  
 1067 ‘unsure’ for each of the following three questions: (1) Does the report describe an actual bug  
 1068 (e.g., not merely submitting a patch)? (2) Does the report contain sufficient information,  
 1069 such as clear natural language descriptions of the buggy behavior, reproduction steps, or  
 1070 detailed system logs? (3) Does the report avoid including solutions, such as identifying the  
 1071 buggy location or attaching patches? If unsure, please select the label ‘unsure.’*” A report was  
 1072 assigned a final label of “yes” only if all three questions received a “yes” from an annotator.  
 1073 Each bug report was independently labeled by three participants. Reports that received at  
 1074 least two “yes” labels across annotators were retained in the final dataset.

Table 9: Mailing list retrieval analysis.

Agent	Recall of Retrieved Mails	None → Found	Found → Lost
SWE-Agent	0.536	0.128	0.080
AutoCodeRover	0.488	0.136	0.056
Agentless	0.460	0.132	0.116

1080 **H EXPERIMENT STATISTICAL SIGNIFICANCE**  
10811082 Table 10: Experiment Statistical Significance of LINUXFL<sup>+</sup>  
1083

Method	Enhanced (Mean $\pm$ Std)	Original (Mean $\pm$ Std)	Mean Diff	t-stat	p-value	CI (Enhanced)	CI (Original)
Agentless	0.549 $\pm$ 0.431	0.419 $\pm$ 0.463	0.129	6.126	0.000	[0.493, 0.600]	[0.361, 0.491]
AutoCodeRover	0.589 $\pm$ 0.437	0.435 $\pm$ 0.469	0.154	5.825	0.000	[0.537, 0.643]	[0.374, 0.493]
SWE-Agent	0.610 $\pm$ 0.433	0.476 $\pm$ 0.463	0.134	5.679	0.000	[0.561, 0.663]	[0.416, 0.533]

1088 To evaluate the effectiveness of the proposed LINUXFL<sup>+</sup>, we performed statistical significance  
1089 tests comparing the MRR scores of LLM agents enhanced with LINUXFL<sup>+</sup> to those  
1090 of their original counterparts. As presented in Table 10, all enhancements introduced by  
1091 LINUXFL<sup>+</sup> yield statistically significant improvements, with paired t-tests producing p-  
1092 values below 0.0001, estimated using 1,000 resamples. Importantly, the confidence intervals  
1093 for the enhanced models do not overlap with those of the original models, providing ad-  
1094 dditional evidence for the significance of the observed improvements. These consistent and  
1095 statistically significant gains across multiple LLM agents underscore the robustness and  
1096 effectiveness of our FL-enhancing framework.

1097 **I LLM USAGE**  
1098

1100 In preparing this work, we used LLMs as an assistive tool. Specifically, LLMs (e.g., Chat-  
1101 GPT) were employed to refine the clarity and readability of manuscript drafts through  
1102 language polishing. Importantly, all research ideas, methodology design, experimental im-  
1103 plementation, and analysis were conceived and conducted by the authors. The LLMs were  
1104 not used for generating research hypotheses, designing experiments, or interpreting results.  
1105 The authors take full responsibility for the content of this paper.

1106  
1107  
1108  
1109  
1110  
1111  
1112  
1113  
1114  
1115  
1116  
1117  
1118  
1119  
1120  
1121  
1122  
1123  
1124  
1125  
1126  
1127  
1128  
1129  
1130  
1131  
1132  
1133