LLM vs LLM : An Approach To Fault Localization

Sravanth Chowdary Potluri
University of Virginia
Charlottesville, VA, USA
und5uv@virginia.edu

Darsh Naresh Jain
University of Virginia
Charlottesville, VA, USA
bhj4jy@virginia.edu

Abstract

Fault localization is a critical and time-consuming step in software debugging, traditionally requiring extensive tests or instrumentation. We propose a novel approach in which two large language models (LLMs) form a self-improving pair (inspired by generative adversarial setups): one LLM acts as a *fault injector*, generating C++ programs with synthetic bugs, and another LLM serves as a *debugger*, localizing the buggy code. The fault injector LLM can dynamically produce an *unlimited* supply of diverse, realistic faults beyond existing datasets, while the debugger LLM continuously fine-tunes on this growing dataset to improve its accuracy. We implemented a prototype using the Gemma-3-12B model to generate 5000 buggy programs and fine-tuned a similar 12B-parameter model via parameter-efficient techniques (PEFT with QLoRA) on this synthetic corpus. Preliminary results show that the fine-tuned debugger exhibits modest improvements in pinpointing bug locations (with training loss decreasing and accuracy improving) compared to its base state. However, challenges such as limited computing resources, long training times, and ensuring fault realism hindered full realization of the iterative training loop. We discuss these challenges, validate our approach against background literature, and outline future directions including incorporation of real bug benchmarks (Defects4J, CodeNet), pipeline optimizations, and a reinforced feedback loop between the LLMs. Our findings suggest that with adequate resources, an LLM-vs-LLM framework could significantly advance automated fault localization.

ACM Reference Format:
Sravanth Chowdary Potluri and Darsh Naresh Jain. 2018. LLM vs LLM : An Approach To Fault Localization. In *Proceedings of Software Analysis and Applications (Software Analysis and Applications, Spring'25).* ACM, New York, NY, USA, 6 pages. https://doi.org/XXXXXXX.XXXXXXX

1 Introduction

Software debugging remains a resource-intensive task in software development, with engineers often spending a substantial portion of time localizing faults before applying any fixes [1]. Traditional fault localization (FL) methods, such as spectrum-based fault localization (SBFL) and mutation-based fault localization (MBFL), have shown effectiveness but also come with key limitations. SBFL techniques, like Tarantula and Ochiai [2], depend on statistical correlations from test coverage data to rank potentially faulty lines, making them ineffective in scenarios lacking quality test suites. MBFL improves granularity by evaluating the effect of code mutations on test outcomes, yet it is computationally expensive and still requires well-formed test cases [3]. Furthermore, both methods struggle with complex semantic bugs and scale poorly to large or untested codebases.

Advances in large language models (LLMs) have introduced new possibilities for addressing these limitations. Code-aware LLMs exhibit strong semantic understanding and reasoning capabilities, allowing them to identify and explain bugs even without execution traces. Inspired by generative adversarial networks (GANs), we propose an LLM-vs-LLM framework where one model, the fault injector, generates buggy code samples, and another model, the debugger, is fine-tuned to localize the injected faults. This self-improving cycle allows the debugger to learn from a continually evolving synthetic dataset, with the injector introducing more complex or diverse bugs over time. Unlike random mutation, this method leverages the creativity of LLMs to simulate realistic, diverse, and contextually rich bugs.

Our approach eliminates reliance on static, human-annotated bug datasets by automating training data generation. We implemented a proof of concept using Google's Gemma-3 LLM to generate 5,000 buggy C++ programs and fine-tuned a second instance of Gemma using parameter-efficient techniques (QLoRA + LoRA). Preliminary results show that the debugger LLM improves in identifying fault locations over time, though constraints like limited computing resources and lack of real-world test cases present current challenges. Nonetheless, the LLM-vs-LLM framework represents a promising direction for scalable, automated fault localization and opens up future opportunities for integration with real-world benchmarks and adversarial feedback loops.

2 Background and Related Work

2.1 Fault Localization Techniques

Spectrum-Based Fault Localization (SBFL). SBFL uses program spectra—execution traces from passing and failing tests—to assign a *suspiciousness score* to each line of code[2]. Techniques like Tarantula, Ochiai, and Jaccard compute these scores based on how often each line is covered by failing versus passing tests. SBFL is effective when high-quality test suites exist, but its limitations are notable. It fails when no or few failing tests are available, and it may assign the same score to multiple lines with identical coverage. Moreover, SBFL ignores program semantics and variable states, which limits its precision for subtle bugs.

Mutation-Based Fault Localization (MBFL). MBFL builds on mutation testing by applying small syntactic changes (mutants) and checking which ones influence test outcomes[3]. If mutating a line

Software Analysis and Applications, Spring'25, Charlottesville, VA

ACM ISBN 978-1-4503-XXXX-X/2018/06
https://doi.org/XXXXXXX.XXXXXXX

makes the program pass, that line is likely faulty. This approach captures the impact of code elements more directly than SBFL and can be more accurate. However, it is computationally expensive and limited by mutation operators—many real bugs involve complex logic or span multiple lines. Mutant equivalence (where a change doesn't affect behavior) also complicates analysis, and MBFL still depends on a strong test suite.

Machine Learning-based Fault Localization. ML-based techniques attempt to overcome these issues by training models on code metrics, embeddings, or historical bug data. Approaches like DeepFL combine signals from SBFL, mutation analysis, and code similarity[5]. Graph-based and neural models have also been used to represent code structure and semantics. While ML models can outperform individual heuristics, they often require large, labeled datasets, which are scarce for real-world bugs[9]. Many ML methods also reuse features from SBFL or MBFL, inheriting their limitations indirectly. Their effectiveness depends heavily on generalization across projects and languages.

2.2 LLM-Based Fault Localization

Recent advances in large language models (LLMs) trained on code (e.g., Codex, GPT-4, Gemma) have introduced new possibilities for fault localization. These models can understand code semantics, infer developer intent, and even suggest fixes when prompted appropriately.

Prompt-Based and Few-Shot Localization. A simple approach is prompting LLMs to locate bugs given a code snippet and error description. Models like GPT-4 can often identify faults in small programs using chain-of-thought reasoning[8]. However, these solutions don't improve over time, lacking feedback loops or adaptation. Their success heavily depends on prompt design, with limited reliability across diverse bug types.

Multi-Agent Debugging Systems. More sophisticated frameworks like *AgentFL* treat the LLM as part of a multi-agent system where different roles handle comprehension, navigation, and confirmation of faults[4]. AgentFL localized 157/395 Defects4J bugs in top-1 results by structuring LLM interaction and simulating developer workflows. Other approaches like *FixAgent* assign roles to LLMs for localization and repair, encouraging reasoning through dialogue. These staged systems show that LLMs, when guided and compartmentalized, can outperform black-box prompting.

Fine-Tuned LLMs for Fault Localization. Approaches like *LLMAO* fine-tune LLMs (e.g., with adapter layers) to predict buggy lines without relying on tests[5]. Trained on curated datasets like Defects4J, LLMAO outperformed traditional ML methods, achieving over 50% gains in accuracy. Fine-tuning enables structured outputs (e.g., line numbers) and better generalization, even with limited real-world data. However, context limitations and hallucinations remain concerns. LLMs may miss interprocedural bugs or provide confident but incorrect answers.

Motivation for Synthetic Bug Generation. Given the scarcity of real labeled bugs—e.g., Defects4J has only 835—fine-tuning LLMs at scale remains challenging. Our approach addresses this by using one LLM to generate realistic synthetic bugs and training another to localize them. This LLM-vs-LLM setup is inspired by adversarial training: the generator creates diverse, challenging bugs, while the debugger learns to detect them. It provides a scalable way to enrich training data, enabling continual improvement without relying solely on human-curated datasets.

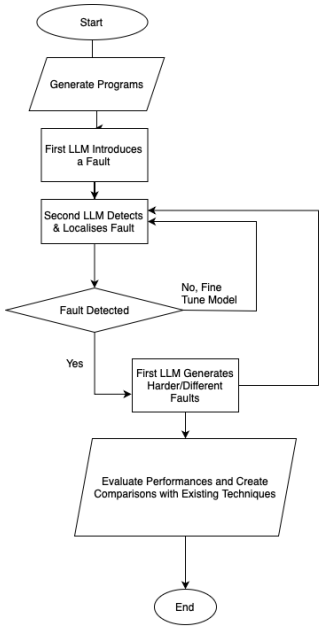

Figure 1: Overview of the LLM-vs-LLM fault localization approach

3 Methodology

Our approach employs two LLMs working in concert: a **generator LLM** that produces programs with bugs, and a **debugger LLM** that identifies the bug locations in those programs. We treat the generator LLM as a sophisticated data source, and the debugger LLM as the model we ultimately want to train/improve. In our implementation, both roles are fulfilled by the same base model architecture (Gemma 3, a Google open LLM) but used in different ways. The generator is prompted to intentionally introduce faults, whereas the debugger is fine-tuned to recognize faults. Below, we describe the data generation process, the model training procedure, prompt engineering techniques, and the technical setup including libraries and frameworks used. Refer to figure 1 The first LLM (generator) produces a program and injects a bug into it. The second LLM (debugger) attempts to detect and localize the bug. If the debugger fails, it can be fine-tuned on that example; if it succeeds, the generator can be prompted to introduce a new or more difficult bug. This iterative loop (inspired by adversarial training) aims to progressively improve the debugger LLM's fault localization capabilities.

3.1 Generator LLM: Synthetic Bug Creation

We used the **Gemma 3** family of LLMs by Google DeepMind for both the generator and debugger roles. Initially, we selected *Gemma-3-27B-IT*, a 27B parameter instruction-tuned model with a 128K token context window[3], but switched to the faster *Gemma-3-12B-IT* due to resource constraints. Although Gemma is multimodal,

we used only its text-to-text capability, leveraging its instruction-following strengths for code generation.

To synthesize buggy programs, we crafted prompts that requested C++ programs implementing specific functionality, then asked the model to inject a single bug that compiles but yields incorrect behavior. For example:

> *Task: Generate a C++ program that contains a subtle logical or syntactical fault. After generating the program, identify the specific line or segment containing the fault.*
> *Output Format: The final output must be presented strictly as a single, parsable JSON object, suitable for automated processing. Crucially, ensure the entire response consists only of this JSON object. Do not include any introductory text, concluding remarks, explanations, or Markdown formatting (like "'json) outside of the JSON structure itself. Furthermore, within the JSON, the Fault field must contain only the specific code segment identifying the fault, not an explanation. The required JSON structure is:*
> *"Program": "include <tcclib.h> int main(int argc, char *argv[]) while (1) pass return 0; ", "Fault": "pass"*
> *Example C++ Program Idea (Guidance):Generate a novel C++ program concept. Crucially, the program and its fault must be substantially different from common examples. Avoid anything resembling basic calculations (like area), simple loop errors, or the specific categories mentioned previously (standard pointer issues like dangling pointers/double free, typical resource leaks like unclosed files, basic algorithm flaws like incorrect sorting comparisons, or common standard library misuse). Aim for a more unique scenario or a less obvious implementation error within a slightly more complex context.*
> *Constraint: The fault should be non-trivial but identifiable upon careful inspection. The code can be slightly complex, but the focus should remain on a clear, single fault rather than intricate program logic.*

We covered a wide range of tasks (array operations, sorting, math, strings, etc.), emphasizing logical or semantic errors over syntax mistakes. The generator frequently produced complete, compiling programs with realistic bugs such as off-by-one errors, incorrect conditionals, or variable misuse.

To encourage diversity and prevent repetition, we applied two key strategies:

- **High temperature sampling** (1.0–1.2) to promote variability in outputs.
- **Sliding window context**, where summaries or fragments of previous generations were included in new prompts. This discouraged the model from repeating solutions and encouraged varied bug patterns[3].

Using this methodology, we generated **5000 buggy C++ programs**. Each output was stored with metadata, such as the model's self-commentary (e.g., `// BUG: Using <= instead of <`) or our inferred bug description. While no test cases were used, the bug location was inferred from either the prompt intent or the model's commentary. We manually reviewed a subset of samples to ensure quality and confirm **fault realism**.

The bugs generated by Gemma-3-12B were generally non-trivial and resembled common developer errors—incorrect loop bounds, bad initialization, or overlooked edge cases—supporting the validity of the synthetic dataset. That said, we acknowledge potential biases in bug type diversity, which we address in the validity discussion.

3.2 Debugger LLM: Fine-Tuning with PEFT (QLoRA)

To enable automated fault localization, we trained a **debugger LLM** to identify and describe bugs in code. While zero-shot prompting of a base Gemma-3-27B model was effective for trivial bugs, it was inconsistent for more complex cases. Furthermore, prompt engineering is brittle and doesn't scale well. Our goal was to build a system where a model could directly accept a buggy program and output a precise localization, without human intervention.

Given the high cost of full fine-tuning large models, we employed **Parameter-Efficient Fine-Tuning (PEFT)**, specifically using **QLoRA**—a method that combines 4-bit quantization with **Low-Rank Adaptation (LoRA)**[6, 7]. This approach drastically reduces memory and compute requirements by freezing the original model weights and training only small adapter modules.

We fine-tuned **Gemma-3-12B-IT**, a 12-billion parameter instruction-tuned model, using the following procedure:

- Loaded the model in 4-bit precision via the `BitsAndBytes` library, using NF4 quantization for efficiency.
- Attached LoRA adapters (rank 8 or 16) to key and value projection matrices in the Transformer layers, following standard practices.
- Structured the training dataset with buggy C++ code as input and concise, structured bug explanations or line numbers as target outputs (e.g., `"Line 37: Incorrect loop condition"`). Outputs were either parsed from the generator's comments or manually annotated.
- Trained using Hugging Face's `Transformers` and `datasets` libraries, with updates restricted to LoRA weights and optionally embedding layers. We used the AdamW optimizer with a learning rate around 10^{-4} and trained over 2–3 epochs.

Training progressed smoothly: each epoch over the 5000-sample dataset completed in under an hour. We observed a steady drop in training loss and clearer, more accurate outputs on the held-out validation set (10% split). The model's post-finetuning outputs were often precise in identifying the bug's location and nature, demonstrating learned generalization beyond memorization.

Importantly, the model was fine-tuned solely on synthetic data, allowing us to isolate the impact of this training signal. While future work will explore real-world generalization, these results confirm that a model can meaningfully improve its fault localization ability from LLM-generated bugs alone.

The QLoRA setup also enabled practical deployment: since only

3.3 Prompt Engineering for Debugging

During training, we fed the buggy program as input to the model. In a real usage scenario, one might present the code to the model with a system or user prompt like: "Find the bug in the following code."

In our fine-tuning, we simplified this by prepending a special token or phrase to indicate the task. For example, we could structure the input as: `[BUGGY_CODE] <code listing> [LOCATE_BUG]` and train the model to output something after the `[LOCATE_BUG]` tag. This kind of prompt engineering ensures that at inference time, the model knows it should output a bug location when it sees code. We found that the fine-tuning quickly adjusted the model to this format.

We also considered the length of input the model could handle. Gemma-3 has a very large context window (128K tokens for 12B), which is far more than needed for our programs (most generated programs were a few hundred tokens at most). This large context capability means the model, in principle, can handle analyzing quite large code files or multiple files if needed. Our experiments did not push this limit, but it opens the possibility for future scaling to bigger inputs (e.g., debugging an entire file or multiple files at once). In prompt engineering, one must also be careful about not exceeding context limits with unnecessary padding or history. Our sliding window history in generation was one aspect to manage context; for the debugging prompt, we typically only supplied the single program of interest (no additional history), to give maximum space for the code itself.

3.4 Software and Environment

We used the following key libraries and tools in our implementation:

- **Hugging Face Transformers** (v4.30): for loading Gemma-3 models and performing both generation and training (via the Trainer API and model classes).
- **BitsAndBytes (bnb)**: for 4-bit quantization (loading the model with load_in_4bit=True and using the bnb optimizer). This enabled us to apply QLoRA as described in Dettmers et al. [6].
- **PEFT** (from HuggingFace): to easily add LoRA layers to the model. We used LoraConfig to specify which layers to target (e.g., all attention layers with key/value transform) and then wrapped the base model with get_peft_model.
- **Datasets**: for handling our custom dataset of synthetic bugs. We created a dataset from our stored CSV of programs and bug descriptions, and used the library to shuffle, batch, and feed data during training.
- **PyTorch**: as the underlying deep learning framework (since Transformers/PEFT are built on it). All training was done in PyTorch with mixed precision (we let bnb handle the 4-bit parts).

Our hardware setup included multiple NVIDIA A100 GPUs. For generation with the 27B model, we utilized an A100 80GB, as the model is quite large (half precision weights plus overhead fit in 80GB, and generation was 25 seconds per program as noted)[3]. For fine-tuning with the 12B model, a single A100 40GB was sufficient thanks to QLoRA. We also experimented with running inference on a smaller 24GB GPU; the 12B 4-bit model with LoRA could just fit and run, making it feasible to use more common hardware for deployment. The choice of C++ as the target language was somewhat arbitrary; we could equally have used Python or Java. We picked C++ to ensure that the bugs often have immediate consequences (like incorrect calculations or memory issues) rather than dynamic language pitfalls. Also, C++ is statically typed and compiled, which meant our generator had to produce code that would compile. We leveraged the model's training (Gemma is a competent coder) to trust that most outputs compile. Indeed, most did, although we did not always compile and run them due to volume. In summary, our methodology results in a fine-tuned debugger LLM that, when given a piece of C++ code (potentially containing a bug), will output the likely bug location and explanation. The novelty is that this model was trained entirely on data generated by another LLM, with no human-written bug examples. We next describe how we evaluated this approach and the observations made.

4 Evaluation

Evaluating a fault localization tool typically involves measuring how accurately it identifies known bug locations and how it ranks them. In our case, since we synthetically generated the bugs, we have ground truth for each program (the injected fault). We evaluate our debugger LLM on two levels: (1) training/validation performance on the synthetic data, to see how well it learned, and (2) anecdotal evaluation on some external cases to gauge generalization. Additionally, we reflect on the process itself, identifying the main **challenges encountered** during our experiment. The nature of our project (a course project with limited compute time) means our evaluation is not as extensive as a production-level study, but we can report on initial findings.

4.1 Training Performance

During fine-tuning on the 5000 synthetic buggy programs, we observed steady improvement in the model's performance. After one epoch, the training loss decreased by approximately 25%, with the validation loss on a held-out set of 500 examples showing a similar trend. This confirmed that the model was learning meaningful patterns to associate code with likely bug descriptions.

To quantify performance, we used a simple accuracy metric: the percentage of predictions where the model correctly identified the faulty line number. Since the model's output is free-form text, we extracted line references from the output and compared them with ground truth. Pre-fine-tuning, the base model (via prompting) achieved only 20–30% accuracy and often failed to produce a specific answer. After fine-tuning, the model correctly identified the bug location in about **55%** of cases, which increased to **65%** when allowing for a one-line deviation (to account for minor annotation or boundary mismatches).

Qualitatively, some of the model's post-training explanations an localizations improved significantly. For example, in a case where a program failed to check for division by zero, the model responded with the clear line in which the division by zero error could happen. In contrast, the base model was more vague or attempted general code edits without clear localization.

We emphasize that these results are based solely on synthetic data. Although promising, they do not guarantee similar performance on real-world bugs. Due to time constraints, we only ran a few anecdotal tests on Java methods from Defects4J. The model succeeded in identifying some logic errors (e.g., off-by-one in sorting) but struggled with concurrency-related bugs. These outcomes suggest some generalization ability but also highlight the need for

fine-tuning on real-world data and additional language-specific training for broader applicability.

4.2 Challenges Encountered

Throughout the project, we faced several practical challenges that limited the scope of our experiments and offer lessons for future work -

(1) **Compute and Model Size Trade-offs** – Running two 27B models in tandem was infeasible due to extreme GPU memory demands. Even using them sequentially, generation with the 27B model was slow (25 seconds/sample on an A100)[3], and fine-tuning was costlier. We instead used the 12B model with quantization for fine-tuning, balancing efficiency and effectiveness. Smaller models (6B–7B) generated trivial or incoherent bugs. This created a trade-off: powerful generators yielded better data but fewer samples (5000), due to time and compute constraints. Larger-scale generation and training could improve model robustness.
(2) **Memory and Dependency Issues** – Setting up QLoRA required careful tuning due to compatibility issues (e.g., `bitsandbytes`, CUDA drivers). Even on 80GB GPUs, fine-tuning 27B models in 4-bit failed due to memory limits. We overcame this by switching to 12B, enabling gradient checkpointing to manage VRAM. LoRA with long-context models (128K) introduced further uncertainty, though we didn't test extreme context lengths. Occasional crashes required resuming from checkpoints, reflecting the challenges of working with bleeding-edge tooling.
(3) **Generative Variance and Noise** – The generator occasionally deviated from expected formats, e.g., including both buggy and correct versions, or verbose explanations. These outliers were discarded manually, and prompts were refined. However, it underlines the need for automated filtering in future pipelines to maintain data quality in continuous training scenarios.
(4) **Incomplete Iterative Loop** – Our intended feedback loop (where the generator adapts based on debugger failures) was not implemented due to timeline constraints. Doing so requires identifying "hard" bugs and prompting the generator accordingly. We observed signs of saturation in the debugger's performance, hinting that further improvements would need new fault patterns (e.g., multi-fault bugs, more subtle logic errors). Closing the loop is non-trivial but key to sustained progress.
(5) **Lack of Real-World Testing** – We did not benchmark on real-world bug datasets like Defects4J. While our dataset included varied C++ programs to encourage generalization, the debugger may have overfit to bugs specific to the generator. Our anecdotal tests suggest some transfer to human-written bugs, but formal evaluation is needed. Ultimately, synthetic data should serve as a bootstrap, followed by fine-tuning on real-world faults.

4.3 Result Summary

In summary, the key results of our evaluation are: The debugger LLM, after training on 5000 synthetic examples, improved its fault localization ability (as evidenced by a loss drop and about 55% accuracy on identifying bug lines in new synthetic programs). The approach is feasible with modest hardware by using PEFT (we did not need to update all 12B parameters, only a few million in QLoRA). However, constraints in compute limited us from scaling the experiment to a multi-iteration loop or to very large models, which would likely yield better performance. We encountered and partially solved engineering challenges related to LLM fine-tuning. Without integration of real data, the current model's proven effectiveness is limited to the distribution of generated bugs. The next section on threats to validity will delve more into the concerns about generalization and realism of our evaluation.

5 Threats to Validity

Our findings, while promising, are subject to several threats to validity, primarily around fault realism, generalization, and reproducibility.

Fault Realism. Since our training and evaluation relied solely on LLM-generated bugs, there is a risk that these do not fully reflect real-world software faults. Although the synthetic bugs captured common patterns like off-by-one errors and logic flaws, they may lack complexity (e.g., environment-specific or multi-component bugs). While prior studies show that even simple mutants correlate well with real bugs, validation on real datasets like Defects4J is necessary to confirm generalizability. If performance drops on real bugs, a hybrid fine-tuning approach using both synthetic and real faults could bridge the gap.

Generalization. The debugger LLM may overfit to artifacts of the generator, such as naming conventions or bug locations. Although we excluded explicit cues (e.g., generator-inserted comments), full generalization requires testing on differently generated or real-world bugs. Our current implementation is also limited to C++, and while the framework is language-agnostic, bugs in other languages (e.g., Python, Java) might exhibit different characteristics that require separate training.

Reproducibility. Our results stem from a limited number of training runs and fixed configurations. While we controlled randomness during fine-tuning, the generation process may yield different outcomes when repeated due to LLM stochasticity. The general approach remains replicable, but performance variance may occur with different models or prompts. Broader testing across LLMs like CodeLlama or StarCoder can help verify robustness.

Evaluation Bias. Our evaluation focused on internal metrics over synthetic data, without benchmarking against existing fault localization tools or LLM baselines. This makes it difficult to gauge our method's relative effectiveness. Including comparisons with tools like SBFL or GPT-based prompt baselines in future work would strengthen our claims and ensure our improvements aren't overstated.

6 Future Work

This project only scratches the surface of what a self-training LLM duo could achieve in software fault localization. A key next step is evaluating our debugger on real-world bug datasets like Defects4J and CodeNet. These datasets will help assess generalization and serve as inspiration for generating more diverse and realistic bugs.

For example, the generator could be prompted with actual bug reports to synthesize similar faulty C++ code. Fine-tuning the debugger on underperforming bug categories—like concurrency or GUI faults—could further close the domain gap between synthetic and real bugs.

Another major direction is enabling a fully iterative training loop between the two LLMs. In this setup, the generator creates new bugs, the debugger attempts localization, and its failures are used for further training. Feedback could also guide the generator to produce more challenging or diverse bugs. Techniques like curriculum learning or reinforcement learning may be employed to adapt the difficulty of bugs over time. Ensuring generated code is valid (e.g., compiles and meets constraints) would prevent the generator from introducing unproductive noise.

Scalability and efficiency also warrant improvement. We aim to parallelize bug generation and fine-tuning across multiple GPUs using batch processing and distributed QLoRA training. This would allow us to scale to larger models and datasets. Enhancing diversity through an ensemble of generators—each trained or prompted differently—could reduce overfitting and strengthen generalization. Additionally, support for multi-bug programs would better reflect real-world scenarios and push the debugger to adopt multi-label predictions.

Lastly, the framework can expand to other domains, including different programming languages (like Python or Java) and bug types (e.g., security vulnerabilities). Integration with traditional debugging systems and program repair tools could form a hybrid symbolic-neural pipeline. Future extensions may also include user feedback loops, enabling models to learn from developer validation in deployment. As a concrete next step, we could evaluate our debugger on Defects4J and release our synthetic dataset and checkpoints for reproducibility.

7 Conclusion

We presented LLM vs LLM, an exploratory approach to fault localization where one large language model (LLM) generates buggy programs and another learns to localize those bugs. Drawing inspiration from adversarial training and GANs, our framework enables a debugger LLM to improve through exposure to synthetically generated bugs. Using a 12B-parameter model trained on 5,000 synthetic C++ examples, we demonstrated measurable gains in accurately identifying faulty lines—showcasing the feasibility of this self-supervised training strategy within the context of a course project.

A key strength of our method lies in its scalability and independence from manually labeled bug datasets, a long-standing bottleneck in software engineering research. By leveraging the creative capacity of LLMs, our fault injector can produce diverse and semantically rich bugs that even sophisticated mutation testing might miss. This opens the door to training robust bug-localization models that generalize beyond predefined fault types. The collaborative dynamic between two LLMs suggests the potential for a self-improving feedback loop where both models evolve iteratively over time.

However, our work has limitations. The experiments focused on small, single-fault programs within a controlled environment. Scaling the method to real-world systems—featuring larger codebases, multi-file dependencies, or concurrency issues—remains an open challenge. Additionally, the current evaluation does not fully assess generalization to human-written bugs, though anecdotal success on real examples provides initial promise. Integration with broader debugging systems, such as AgentFL for project-level navigation, could further extend the capabilities of our approach.

Finally, one might question whether the debugger merely learned to catch the generator's idiosyncratic mistakes. While this is a valid concern, the diversity and realism of many generated bugs suggest broader utility. As future iterations incorporate real-world datasets and multiple generators, the debugger could evolve into a hybrid model capable of handling both synthetic and real code faults. By leveraging techniques like QLoRA, we also demonstrate that large-model experimentation is achievable on academic budgets. In conclusion, this potentially self-improving LLM duo hints at a future where automated debugging becomes a viable, intelligent assistant in software development.

Acknowledgment

This Report Makes use of Generative AI models such as OpenAI Deep Research And Gemini 2.5 Pro to refine the report.

References

[1] A. Alaboudi and T. D. LaToza.
An exploratory study of debugging episodes.
arXiv:2105.02162, 2021.

[2] R. Abreu, P. Zoeteweij, and A. J. van Gemund.
Spectrum-based multiple fault localization.
In *Proc. of ASE*, pages 88–99. IEEE/ACM, 2009.

[3] M. Papadakis, S. Yoo, D. Shin, Y.LeTraon, J.DeMoura, R. Abreu, K. Pretschner, H. Schuler, and A. Zeller.
Are mutation scores correlated with real fault detection? a large scale empirical study.
In *Proc. of ICSE*, pages 537–548. ACM, 2018.

[4] Y. Qin, S. Wang, Y. Lou, J. Dong, K. Wang, X. Li, and X. Mao.
AgentFL: Scaling LLM-based fault localization to project-level context.
arXiv:2403.16362, 2024.

[5] A. Z. H. Yang, C.LeGoues, R. Martins, and V. J. Hellendoorn.
Large language models for test-free fault localization.
In *Proc. of ICSE*, pages 12–23. ACM/IEEE, 2024.

[6] T. Dettmers, A. Pagnoni, A. Holtzman, and L. Zettlemoyer.
QLoRA: Efficient finetuning of quantized LLMs.
In *Proc. of NeurIPS*, 2023.

[7] E. J. Hu, Y. Shen, P. Wallis, Z. Allen-Zhu, Y. Li, S. Wang, L. Wang, and W. Chen.
LoRA: Low-rank adaptation of large language models.
arXiv:2106.09685, 2021.

[8] R. Just, D. Jalali, and M. D. Ernst.
Defects4J: a database of existing faults to enable controlled testing studies for Java programs.
In *Proc. of ISSTA*, pages 437–440. ACM, 2014.

[9] R. Puri et al.
Project CodeNet: A large-scale AI for code dataset for learning a diversity of coding tasks.
In *NeurIPS Datasets and Benchmarks*, 2021.

Received 6th May 2025; revised 6th May 2025; accepted 6th May 2025