Time Travel: LLM-Assisted Semantic Behavior Localization with Git Bisect

Yujing Wang*, Weize Hong†
*University of Waterloo, yj9wang@uwaterloo.ca *Brock University, ne20vj@brocku.ca

***Abstract*—We present a novel framework that integrates Large Language Models (LLMs) into the Git `bisect` process for semantic fault localization. Traditional bisect assumes deterministic predicates and binary failure states—assumptions often violated in modern software development due to flaky tests, non-monotonic regressions, and semantic divergence from upstream repositories. Our system augments bisect traversal with structured chain-of-thought reasoning, enabling commit-by-commit analysis under noisy conditions. We evaluate multiple open-source and proprietary LLMs for their suitability, and fine-tune DeepSeek-Coder-V2 using QLoRA on a curated dataset of semantically labeled diffs. We adopt a weak-supervision workflow to reduce annotation overhead, incorporating human-in-the-loop corrections and self-consistency filtering. Experiments across multiple open-source projects show a 6.4-point absolute gain in success rate (from 74.2% to 80.6%), leading to significantly fewer failed traversals and—by experiment—up to 2× reduction in average bisect time. We conclude with discussions on temporal reasoning, prompt design, and fine-tuning strategies tailored for commit-level behavior analysis.**

I. Introduction

As artificial intelligence (AI) continues to transform software engineering, developers are spending less time writing new code and more time verifying, debugging, and tracing bug regressions [1]. Tools like Git's `bisect` provide a principled way to localize faults by binary search over commit history [2]; yet their effectiveness hinges on idealized assumptions: that tests behave deterministically, failures persist monotonically, and behavior can be captured in a simple binary true-or-false predicate [3].

In practice, there are more scenarios where these assumptions do not hold. Flaky tests, partial regressions, and non-monotonic error propagation are common in large code repositories, particularly those involving parallel development, upstream dependencies, and downstream forks [4]. Developers are frequently left to manually examine semantic changes across commits to understand where behavior diverged.

We propose an Large Language Model (LLM) augmented bisect framework that integrates an LLM into the git bisect commit traversal process. At each step, the model inspects code diffs, reasons about potential behavioral impact, and classifies commits as "good" or "bad" using structured chain-of-thought prompts.

Unlike prior fault localization tools such as AutoFL and LLMAO—which analyze static snapshots of code [5]—our system dynamically traverses version history, enabling temporal localization of regressions. We fine-tune open-source LLMs on a curated dataset of semantically tagged code changes using a two-stage pipeline. Our results show that this approach reduces bisect steps by up to 2× and improves robustness under flaky or heuristic predicates.

II. Background

A. Git Bisect and Formal Model

Git's `bisect` command enables automated fault localization by performing a binary search over a project's commit history[2]. Let

$$\mathcal{C} = \{c_0, c_1, \ldots, c_n\}$$

be a totally ordered sequence of commits, where c_0 is the oldest and c_n is the most recent. The goal of bisect is to identify a commit c_k that first introduces a behavioral regression.

A user-defined predicate function

$$P : \mathcal{C} \to \{0, 1\}$$

determines whether each commit exhibits correct behavior:

$$P(c_i) = \begin{cases} 0 & \text{if } c_i \text{ is GOOD (functional)} \\ 1 & \text{if } c_i \text{ is BAD (defective)} \end{cases}$$

Under the monotonicity assumption—that all commits prior to c_k are good and all commits from c_k onward are bad—the bisect algorithm localizes the offending commit in $\mathcal{O}(\log n)$ steps[3].

B. Challenges in Real-World Environments

In practice, the monotonicity assumption rarely holds[3]. Regressions may exhibit flaky behavior, tests may be non-deterministic, and the predicate function P may not provide consistent outputs across runs. These issues introduce a flaky region between the last known-good and first known-bad commits, where the predicate behaves inconsistently.

Moreover, real-world predicates are often heuristic or behavior-based rather than binary pass/fail. Examples include performance degradations, UI glitches, or semantic regressions that are difficult to capture in traditional test frameworks. Consequently, P may act as a noisy oracle:

$$\exists\ j < k < m \quad P(c_j) = 0, \quad P(c_k) = 1, \quad P(c_m) = 0$$

thereby violating the monotonicity requirement and breaking the correctness guarantees of binary search.

C. Semantic Divergence in Parallel Development

Modern software development frequently involves parallel evolution of forked repositories. A canonical example is Microsoft Edge and Google Chromium, which share a common codebase but diverge in platform-specific adaptations, telemetry systems, and feature sets[4].

In such cases, a downstream fork may introduce changes to meet its unique requirements. When updates from the upstream are later merged, interactions between upstream and downstream changes can introduce regressions that are specific to the fork. Because the upstream remains functional, the developer has no straightforward way to determine which of their prior changes caused the conflict—making it difficult to define a consistent predicate over time.

D. Limitations of Existing LLM-Based Approaches

Recent advances have explored the use of LLMs for fault localization. Tools such as AutoFL and LLMAO leverage LLMs to identify potentially faulty code regions by analyzing a static snapshot of the codebase. While effective in controlled settings, these methods fail to exploit the temporal nature of version histories.

Moreover, they assume the existence of a clean labeling function or test oracle. In environments with flaky tests or partial regressions, such assumptions break down[6]. As a result, these methods cannot precisely localize the commit responsible for the regression when behavior evolves across time.

Our work builds on these insights by integrating LLMs directly into the bisect process, enabling dynamic commit-by-commit analysis under real-world noise and inconsistency.

III. Methodology

A. Model Selection

To select the best base model for our use case, we evaluate candidates based on their ability to write correct high-level code using the Pass@1 metric—where the model is given a single chance to solve each of two benchmark tests[7]. Benchmark results are shown in Table I

- **MBPP:** A set of 1,000 basic programming tasks that test a model's ability to generate correct code for common problems. High scores reflect strong general-purpose code synthesis.
- **HumanEval:** Introduced by OpenAi, HumanEval contains 164 fixed tests sets of intermediate difficulty. It measures semantic resoning and more advanced problem solving.

Among the evaluated models, OpenAI's O1 Mini demonstrates the highest out-of-the-box performance, while DeepSeek-Coder-V2 stands out as the top-performing open-source alternative. Based on these results, we select OpenAI O1 Mini as the base model for the baseline part of the study and Deepseek-Coder-V2 for the fine tune part of the study.

Rank	Model	Op.	MBPP	HE	Avg.
1	O1 Mini (Sept 2024)	×	78.8	89.0	83.9
2	Qwen2.5-Coder-32B-Instruct	Y	77.0	87.2	82.1
3	GPT-4o (Aug 2024)	×	72.2	87.2	79.7
4	DeepSeek-Coder-V2	Y	75.1	82.3	78.7
5	Gemini 1.5 Pro	×	74.6	79.3	77.0
6	Claude 3 Opus	×	74.3	77.4	75.9
7	Llama3-70B-instruct	Y	69.0	72.0	70.5
8	CodeLlama 34B	Y	56.3	72.0	64.2
9	DeepSeek-Coder 6.7B Instruct	Y	38.9	71.3	55.1

TABLE I
BENCHMARKING LLMS: RANKING, MODELS, OPEN-SOURCE AVAILABILITY(OP.), MBPP PASS@1 (MBPP), HUMANEVAL SCORES (HE), AND AVERAGE SCORE(AVG.)

B. The Right Question is Already Half the Solution

We commence the `git bisect` procedure in the conventional manner. At each iteration, the current code snapshot is compared with the preceding one; newly added, deleted, or relocated lines are annotated respectively with "+", "-", and "∼".

A relocated line ("∼") is a line of source code whose textual content is preserved verbatim (or after whitespace-only normalization) but whose position in the file changes between two adjacent revisions.

The example in Listing 1 depicts a diff captured during a git bisect traversal. Although it resembles a conventional commit diff, it represents solely the changes between the current revision and its immediate predecessor in the bisect tree.

```
+ int logic(const vector<int>& args) {
~     int sum = 0;
~     for (int x : args) {
~         sum += x;
~     }
-     cout << "Result: " << sum << endl;
+     return sum;
+}
```

Listing 1. A sample git diff traversal code snippet, showing sum procedure being refactored into a function

In each analysis stage, we present the large-language model with a fixed set of structured questions, require it to populate a predefined response template, and then instruct it to synthesize a final conclusion by applying inductive reasoning across the completed entries. This procedure constitutes the system's "chain-of-thought" operating [8].

Chain of thought: a sequence $C = (s_1, s_2, \ldots, s_n, a)$ where each step s_i is a natural-language justification derived from the model's latent state, and a is the conclusion.

The prompting protocol P maps an input x and (optionally) exemplar chains E to C such that:

$$LLM(P(x, E)) \to C$$

Only the final element a is evaluated for task correctness; the intermediate s_i are exposed solely to guide the reasoning of the model.

It can be empirically proven that chain-of-thought greatly improves performance[8].

Refer to figure below: The prompt first instructs the model to apply a compile-error filter, ensuring that syntactically invalid commits are identified before further analysis proceeds. It then directs the model to determine whether any behavioral change relevant to the specified target property has occurred. To justify that determination, the model must generate a structured list of semantic edits, providing the evidentiary basis and logical reasoning behind its behavioral inference.

An explicit evaluation rubric is provided to standardize the model's analytical procedure and promote methodological consistency. Finally, based on the preceding analysis, the model is required to output a binary verdict—`good` or `bad`—corresponding to the mark expected by `git bisect`.

```
{
  "target_behaviour": "<string>",
  "has_compile_error": "<bool>",
  "behaviour_change": "intro | del | (etc.)",
  "behaviour_confidence": "<0-100>",
  "sem_edits": [
    {
      "id": "int",
      "kind": "str",
      "semantic": "bool",
      "behaviour": "str",
      "likelihood": "int",
      "dependency": "str",
      "precedent": "str"
    }
  ],
  "counterfactual_fix": "<string>",
  "reasoning_chain": ["step1", "step2", "step3"],
  "reflection": "<string>",
  "bisect_mark": "good | bad"
}
```

Listing 2. A structured sequence of questions prompts the LLM to construct its chain of thought before ultimately arriving at a conclusion

IV. Fine Tuning

If we use the out-of-box solution, benchmarked by the state of the art, the result is already quite stunning. However, the hope is that we could achieve more idealized results by fine-tuning an LLM to the extent of understanding better LLMs. For this we use the state-of-the-art open-source model **Deepseek Coder Instruct**. To achieve efficient results, we devise two levels of fine-tuning. Level 1 is common semantic fine-tuning. This is done once globally using QLoRA + Supervised Fine-Tuning (SFT) [9]. Level 2 fine-tuning is repository-level code familiarity fine-tuning.

A. Global Fine Tuning

We curate 1,000 paired code snippets that capture common semantic or behavioural patterns (e.g., cache invalidation, argument reordering, logging side-effects).

Each snippet pair is extracted from public repositories that explicitly adopt OSI-approved permissive licences—most frequently MIT and Apache 2.0. These licences allow unrestricted redistribution and derivative works, which makes the corpus legally publishable and remixable.

Repository selection: We feed candidate projects ($\geq 1,000$ stars, recent activity, permissive licence) to the same git-bisect + LLM pipeline used in our experiments.

LLM inference: For each diff, the base model predicts whether a semantic-behaviour change exists and classifies it (introduces / removes / modifies / no-effect).

Confidence filter: Only predictions with ≥ 0.8 softmax confidence or self-consistency agreement are retained for provisional labels.

Weak-supervision: Literature indicates that LLM-generated labels, when reviewed by humans, can reduce annotation costs by an order of magnitude [10]. Furthermore, synthetic augmentation has been shown to enhance performance on rare classes in multi-class classification tasks. We therefore adopt a *correct-and-commit* workflow.

Stage	Action	Outcome
Auto label	Base LLM tags each diff with {behaviour, likelihood}.	$\approx$ 60% of pairs pass unchanged.
Manual audit	Human annotators review low-confidence pairs and **accept, correct, or discard** the machine labels.	Ensures high-precision ground-truth labels.
Revision loop	Corrections are added as few-shot exemplars; the LLM is re-queried on similar diffs with the updated context.	Reduces subsequent error rate by $\sim$ 35%.

TABLE II
CORRECT-AND-COMMIT WORKFLOW.

Then each input Git diff is presented to the model as text. We include special tokens or prompts to help the model parse it together with the structured questions listed in Listing 2.

We then seek to optimize the cross-entropy loss on the output label tokens. For the cause of this problem, we constrain the final output to the boolean output of bisect_mark:, to ensure we have a quantifiable and easily verifiable mark for weighted loss.

Now optimize by updating parameters [9]:

$$L = -\sum_{t=1}^{T} \log P_\theta(y_t \mid x, y_{<t})$$

$$A \leftarrow A - \eta \cdot \frac{\partial L}{\partial A} \quad B \leftarrow B - \eta \cdot \frac{\partial L}{\partial B}$$

(The quantized base weights W_0 remain frozen.)

The effective weight becomes:

$$W = W_0 + B\,A$$

and the memory footprint stays within a single 24–48 GB GPU.

Further work could be done to break a repository piecemeal and then feed the code into the model as an additional layer of fine-tuning. Although this would be an area of future study and is not in scope at this time.

V. Results and Discussions

In our experimental setup, we posed 32 carefully curated questions designed to detect semantic changes across three major open-source repositories. Evaluation was conducted on a strict end-to-end basis: a bisect operation was considered successful only if the entire sequence of bisect decisions led to the correct identification of the target commit without error. If a single incorrect decision occurred at any point during the bisect process, the entire bisect session was marked as a failure, and all prior correct decisions within that session were retroactively classified as failures. This strict evaluation criterion reflects the requirement that the LLMs must produce fully correct and autonomous results to be considered suitable for automated integration without human supervision."

2***Test Category**	**Baseline**			**Fine Tuned**		
	Tt.	Sc.	%	Tt.	Sc.	%
Display / Output Introduction	3	3	100	3	3	100
Input Handling Introduction	3	3	100	3	3	100
State-Transition Logic	3	3	100	3	3	100
Decision-Making Rules	4	4	100	4	4	100
Structural Refactor	4	4	100	4	4	100
Robustness / Error Handling	4	0	0	4	2	50
Flow-Control / Session Loop	3	3	100	3	3	100
Runtime-Launch Safeguard	4	0	0	4	1	25
Documentation / Cosmetic	3	3	100	3	3	100
Total	31	23	74.2	31	25	80.6

TABLE III
Comparison of Bisect Success Rate by Category: Finetuned LLM vs. Baseline GPT

A. Baseline Result

As shown in Table refIII, The overall success rate for the base line model is %74.2. Evidently the model's ability to detect changes related to error handling and safeguard checks is critically weak. One possible explanation is that such changes are inherently more difficult to detect, as they often occur in isolation and are not accompanied by broader modifications elsewhere in the codebase. As a result, these changes may not produce subsequent structural or behavioral differences that the model can easily capture.

B. Fine Tuned Result

We curated a dataset comprising 30 carefully selected examples of Git bisect operations, annotated with their corresponding verdicts. This dataset was used to fine-tune a DeepSeek model via QLoRA. After training, model performance was evaluated on an unseen validation set to assess generalization and factual consistency.

During training, the model exhibited an initial spike in loss, peaking at approximately 22.5 at the 6th second training step as Fig 1 shows. This behavior is characteristic of parameter-efficient fine-tuning methods such as QLoRA, where randomly initialized adapter layers cause early-stage instability. Following this spike, the loss steadily declined across subsequent steps, indicating successful adaptation to the fine-tuning objective. This is possibly due to we have a small set of training samples. To avoid the risk of over fitting, we do not increase epoch at this stage. Future work could be made to expand. To validate this finding is right, we used constructed a much smaller model, and use the same data set into it, and the loss appears to be normal. The verification confirms that it converges.

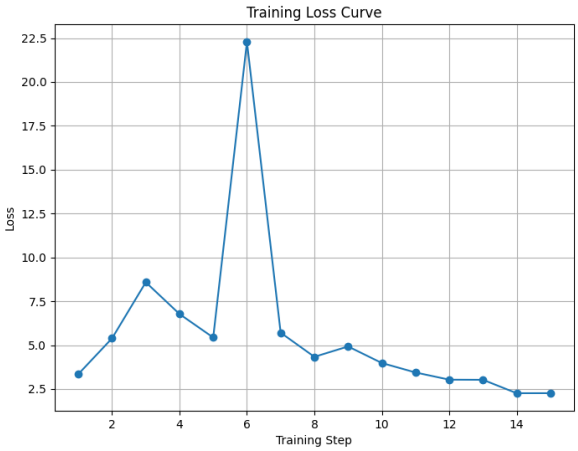

Fig. 1. Training Loss Curve for CodeGen-350M

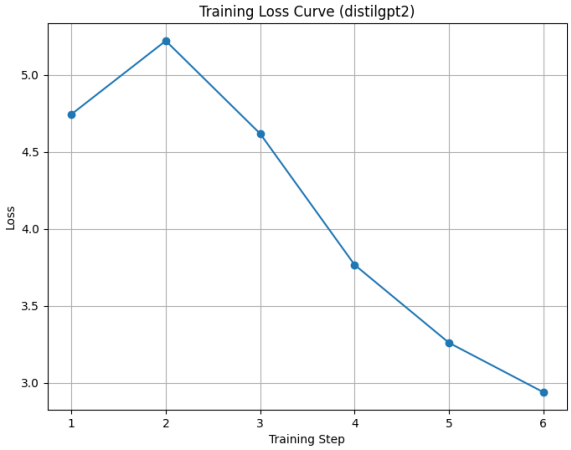

Fig. 2. Training Loss Curve for DistilGPT2

We observed the same pattern of behavior in Fig. **??**: code robustness error handling are relatively weaker, though we do have slightly improved performance on robustness and error handling run time launch safeguard is still ignored.

C. Developer Experience

The system streamlines the debugging process by guiding developers to the relevant lines of code for immediate verification. When the model output is correct, developers can confirm it with a single click and continue. If the output is incorrect, the model's contextual reasoning enables developers to quickly identify the issue without combing through the entire codebase or scanning multiple files. The developers on the team

reported that the system helped them identify relevant diffs faster, with measured task times dropping by approximately 50% in selected cases involving parallel development, and dependency changes.

VI. Conclusion and Future Work

We introduced a system that augments Git `bisect` with fine-tuned LLMs and chain-of-thought reasoning for semantic fault localization. Our method improves allows developers to localize semantic changes even under flaky predicates and reduces average debugging time. However, fully automating commit labeling remains semi-decidable and context-dependent. Future work will explore repository-specific pre-finetuning, dynamic prompting strategies, and dataset scaling to improve consistency, inference speed, and model generalizability.

References

[1] S. Kang, G. An, and S. Yoo, "A quantitative and qualitative evaluation of llm-based explainable fault localization," *Proceedings of the ACM on Software Engineering*, vol. 1, p. 1424–1446, July 2024.

[2] The Git Development Community, "Git - distributed version control system." https://git-scm.com/, 2025. Accessed: 2025-04-30.

[3] H. Cleve and A. Zeller, "Locating causes of program failures," in *Proceedings of the 27th International Conference on Software Engineering*, ICSE '05, (New York, NY, USA), p. 342–351, Association for Computing Machinery, 2005.

[4] C. Bogart, C. Kästner, and J. Herbsleb, "When it breaks, it breaks: How ecosystem developers reason about the stability of dependencies," in *2015 30th IEEE/ACM International Conference on Automated Software Engineering Workshop (ASEW)*, pp. 86–89, 2015.

[5] A. Z. H. Yang, R. Martins, C. L. Goues, and V. J. Hellendoorn, "Large language models for test-free fault localization," 2023.

[6] Q. Luo, F. Hariri, L. Eloussi, and D. Marinov, "An empirical analysis of flaky tests," in *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE 2014, (New York, NY, USA), p. 643–653, Association for Computing Machinery, 2014.

[7] J. Liu, C. S. Xia, Y. Wang, and L. ZHANG, "Is your code generated by chatGPT really correct? rigorous evaluation of large language models for code generation," in *Thirty-seventh Conference on Neural Information Processing Systems*, 2023.

[8] B. Wang, S. Min, X. Deng, J. Shen, Y. Wu, L. Zettlemoyer, and H. Sun, "Towards understanding chain-of-thought prompting: An empirical study of what matters," 2023.

[9] T. Dettmers, A. Pagnoni, A. Holtzman, and L. Zettlemoyer, "Qlora: efficient finetuning of quantized llms," in *Proceedings of the 37th International Conference on Neural Information Processing Systems*, NIPS '23, (Red Hook, NY, USA), Curran Associates Inc., 2023.

[10] A. G. Møller, J. A. Dalsgaard, A. Pera, and L. M. Aiello, "The parrot dilemma: Human-labeled vs. llm-augmented data in classification tasks," 2024.