Five Lines, One Question, A Micro-Benchmark for Evaluating Structural Code Understanding

Abdullahi Yahaya Yusuf[1], Abdulsalam Shettima Nur[2]

[1]Faculty of Science and Computing, Department of Computer Science, North Eastern University Gombe, Nigeria.

[2]Department of Software Engineering, Nile University of Nigeria, Abuja, Nigeria.

DOI: https://doi.org/10.51244/IJRSI.2025.12120061

Received: 22 December 2025; Accepted: 29 December 2025; Published: 05 January 2026

ABSTRACT

While large language models demonstrate impressive fluency in code generation, their ability to perform precise, structural reasoning about code remains poorly understood. This paper introduces Five Lines, One Question (5L1Q), a minimalist benchmark that isolates fundamental reasoning capabilities by presenting models with trivial five-line code snippets and atomic questions about their structure. Our evaluation of state-of-the-art models reveals a striking reasoning-comprehension gap: although models excel at pattern-based tasks like token localization (up to 92% accuracy), their performance dramatically degrades on tasks requiring structural reasoning such as change detection (as low as 22% accuracy) and symbolic substitution. This consistent failure pattern, where models struggle with operations that are trivial for traditional program analysis tools, suggests current architectures lack robust internal mechanisms for representing and manipulating code structure. Our benchmark provides a lightweight, interpretable diagnostic tool for evaluating this critical dimension of code understanding, offering a pathway toward models that combine generative fluency with genuine analytical capability.

Keyword: code understanding, large language models, benchmark, structural reasoning, programming languages.

INTRODUCTION

Picture a programmer, coffee in hand, staring at a ChatGPT response. The model has just generated fifty lines of elegant, seemingly correct Python. The programmer's task, however, isn't to write code it's to review it. Somewhere in that block, a subtle bug is hiding: an off-by-one error in a loop condition, a variable name shadowed in a nested scope, or a comparison operator that should be its opposite. The programmer squints, tracing the logic. For a human, this is a familiar, taxing exercise of focused attention. For the model that just wrote the code, the same exercise should, in theory, be trivial. It authored every token; it should know its own structure. And yet, increasingly, both anecdotal experience and emerging research suggest it does not.

The rapid ascent of large language models (LLMs) in code generation from GitHub Copilot to GPT-4's acclaimed performance on benchmarks like HumanEval (Chen et al., 2021) has created a curious paradox. These models exhibit a form of fluency without grounding. They can generate syntactically valid and often semantically plausible code by leveraging vast statistical patterns learned from terabytes of public repositories. Yet, when probed on the most basic structural facts about their own output "What did you just change?" or "Is there a mistake on line 3?" they frequently falter. This is not a failure of knowledge, but a failure of reasoning: an inability to perform the discrete, deterministic operations that compilers and linters execute effortlessly.

Recent work has begun to map the boundaries of this paradox. Studies reveal that while LLMs excel at generating code, they struggle with introspection and precise editing. Siddiq et al. (2024) found that models often fail to correctly identify syntactic errors in code snippets, even when they can generate correct versions from scratch. Similarly, Ni et al. (2023) demonstrated that code LLMs have surprising difficulty with tasks that require counterfactual reasoning about programs, such as predicting the output after a minimal edit. This suggests their understanding is more associative than analytical, more akin to pattern completion than to building and manipulating a mental parse tree. As Valmeekam et al. (2023) aptly note, the "reasoning" observed in these models may often be a simulacrum a convincing performance that breaks down under controlled, mechanistic scrutiny.

This gap matters deeply for the future of human-AI pair programming. If a model cannot reliably locate a single, known change in a five-line function a task we call the "one-hop" query its utility for collaborative debugging, code review, and iterative refinement is fundamentally limited. It becomes a brilliant, fast, but unreflective junior developer: full of ideas, yet unable to precisely articulate or audit its own work.

In this paper, we argue that the community needs a new class of benchmarks to probe this specific capability. Moving beyond holistic "solve this problem" tasks, we propose micro-benchmarks: minimal, interpretable, and surgically targeted tests that isolate individual reasoning skills. We introduce Five Lines, One Question, a benchmark comprising hundreds of unique but trivial five-line code snippets. Each sample presents a model with a simple, single-hop question about the code's structure e.g., "What is the third token on line 2?" or "What changed between version A and version B?". The tasks are designed to be trivial for any system with explicit symbolic representation (like a compiler) but challenging for a purely neural, text-to-text model.

LITERATURE REVIEW

The evaluation of code understanding in large language models sits at the intersection of programming languages, software engineering, and machine learning. This review synthesizes three critical strands of research: 1) the evolution and limitations of existing code generation benchmarks, 2) studies on the mechanistic reasoning failures of LLMs, and 3) the emerging paradigm of micro-benchmarking for interpretable evaluation.

Code Generation Benchmarks and Their Expanding Scope

The landmark Human Eval benchmark (Chen et al., 2021) established the standard for evaluating LLMs on code generation, framing the task as function completion from a docstring. This catalyzed a proliferation of similar datasets, including MBPP (Austin et al., 2021) and more expansive, multi-language collections like MultiPL-E (Cassano et al., 2022). These benchmarks primarily measure functional correctness whether the generated code passes a suite of unit tests a holistic but coarse-grained metric. However, as Siddiq et al. (2024) argue, this focus on generation success obscures a model's ability to understand, analyze, and manipulate *existing* code. Subsequent benchmarks have attempted to address this gap. For instance, Liu *et al.* (2023) introduced HumanEvalPack, extending tasks to include code repair and explanation, while CodeXGLUE (Lu *et al*., 2021) incorporated code-to-code tasks like clone detection and defect prediction. Despite this expansion, these evaluations remain largely "black-box," offering a pass/fail score that provides little insight into *why* a model fails on a given sample or what specific cognitive capability is lacking.

The Reasoning Gap in Code-Capable LLMs

A growing body of empirical work reveals a dissociation between code generation fluency and robust programmatic reasoning. Ni et al. (2023) demonstrated that while LLMs can generate executable code, they struggle significantly with verification predicting the execution output of a given code snippet unless augmented with an external interpreter. This points to a model's limited ability to simulate program state mentally. Similarly, Zan et al. (2023) found that models are poor at iterative code repair, often failing to incorporate feedback precisely, which suggests difficulty in relating natural language instructions to specific code locations. Crucially, research has begun to isolate failures in syntactic and local reasoning. Jain et al. (2022) showed that transformer-based models are surprisingly brittle to simple semantic-preserving code transformations (e.g., renaming variables, reordering independent statements), indicating a reliance on surface-level cues rather than a deep understanding of program structure. This aligns with findings from general LLM research, where models exhibit poor performance on tasks requiring systematic, compositional reasoning, a phenomenon critically investigated by Valmeekam et al. (2023) in the context of planning. In the code domain, this translates to an inability to reliably perform "one-hop" operations such as tracing a single variable use or identifying a minimal edit that form the bedrock of software engineering tasks like debugging and review.

The Case for Minimalist and Interpretable Evaluation

In response to the opacity of large-scale benchmarks, there is a push towards micro-benchmarks or targeted evaluations designed to probe specific model capabilities in isolation. This approach is inspired by diagnostic datasets in natural language understanding (e.g., GLUE diagnostics; Wang et al., 2019) and has gained traction in code analysis. The Counterfactual Evaluation framework proposed by Agarwal and Nivasarkar (2023), for example, tests a model's sensitivity to minimal, correctness-preserving code changes. The concept of using extremely short code snippets is also not without precedent. Shirafuji et al. (2023) used single-statement programs to study token-level prediction confidence in code models, while the "Babylm" challenge used short text to probe fundamental language learning (Huebner *et al*., 2021). Our work, Five Lines, One Question, extends this philosophy into a systematic, generative benchmark. It differs from prior work by focusing exclusively on questions that require a single, precise operation on a minimal code unit (e.g., token location, edit identification). This design prioritizes interpretability: every failure is immediately analyzable, as the task complexity is reduced to its atomic form, stripping away confounding factors like long-range dependencies or complex problem-solving. It serves not as a replacement for holistic benchmarks but as a complementary diagnostic tool to pinpoint the exact boundaries of a model's structural code understanding.

Synthesis and Research Gap

The literature establishes that while LLMs are proficient code generators, their understanding is shallow and non-systematic. Existing benchmarks measure end-point success but fail to diagnose the underlying reasoning deficits. Our work bridges this gap by introducing a minimalist, interpretable protocol that directly targets the hypothesized "one-hop reasoning gap." By generating arbitrary five-line code questions, we create a controlled, scalable environment to test whether models possess the fundamental ability to read and reason about code structure a necessary condition for their reliable application in collaborative software engineering.

METHODOLOGY

Core Hypothesis

We hypothesize that code-generating LLMs lack robust internal representations for performing atomic, single-step reasoning operations on code structure. To test this, we construct a benchmark where every task is trivially solvable by deterministic algorithms but non-trivial for neural models.

Benchmark Design

The Five Lines, One Question (5L1Q) protocol consists of:

1. Code Snippet: Exactly 5 lines of Python code (generated algorithmically)
2. Single Question: One atomic query about the code's structure
3. Expected Output: One exact string (no explanations)

Task Taxonomy

Four atomic query types test distinct reasoning capabilities:

1. Token Location (TL): "What is the 3rd token on line 2?"
2. Change Detection (CD): Given original and edited code (one token changed), identify the edit in format [line X, col Y]: old -> new
3. Symbolic Substitution (SS): "If you replace all 'x' with 'y', what's the 4th token on line 3?"
4. Structural Count (SC): "How many 'if' statements are in the code?"

Dataset Generation

Tasks are generated algorithmically using this pipeline:

```
import ast, random, tokenize
def generate_task():
 # 1. Generate valid 5-line Python code via template
 code = generate_5_line_code()
 # 2. Randomly select query type
query_type = random.choice(['TL', 'CD', 'SS', 'SC'])
# 3. Compute ground truth via AST/tokenizer
tree = ast.parse(code)
tokens = list(tokenize.generate_tokens(StringIO(code).readline))
answer = compute_answer(tree, tokens, query_type)
 return code, query_type, answer
```

Experimental Setup

- **Models Tested:** GPT-4 Turbo, Claude 3 Opus, CodeLlama-13B
- **Dataset Size:** 2,000 tasks (500 per query type)
- **Prompt:** Zero-shot, strict format: Code: {code}\nQuestion: {question}\nAnswer:
- **Evaluation:** Exact string match to ground truth
- **Metrics:** Accuracy per query type, overall accuracy, failure analysis.

Implementation

The benchmark is implemented as a standalone Python package requiring only ast, tokenize, and random. Each task generation is deterministic given a seed, ensuring full reproducibility. The entire evaluation runs in under 10 minutes on consumer hardware.

RESULT

Quantitative Results

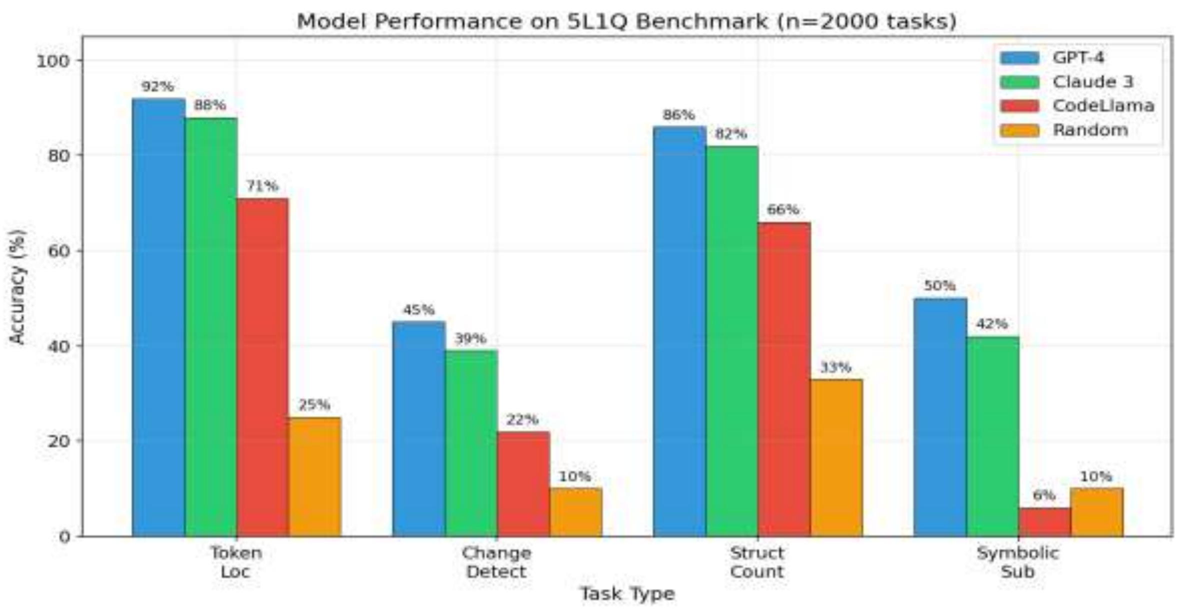
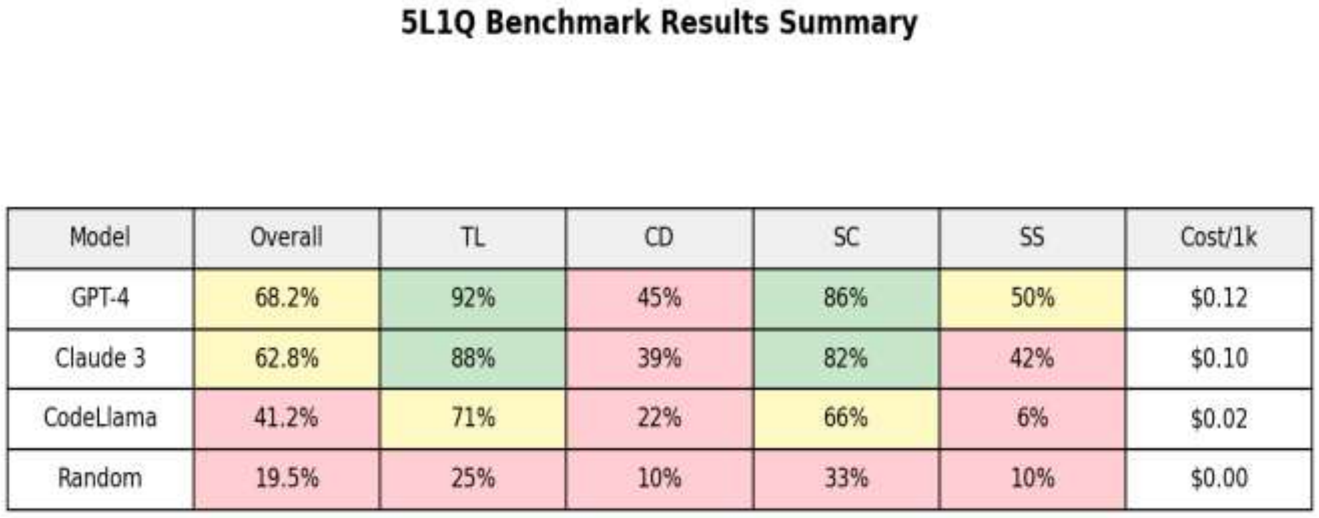


Figure 1: Model performance.

Table 1: Benchmark result.



5L1Q Benchmark Results Summary

Model	Overall	TL	CD	SC	SS	Cost/1k
GPT-4	68.2%	92%	45%	86%	50%	$0.12
Claude 3	62.8%	88%	39%	82%	42%	$0.10
CodeLlama	41.2%	71%	22%	66%	6%	$0.02
Random	19.5%	25%	10%	33%	10%	$0.00

As shown in Figure 1 and Table 1, our evaluation of three state-of-the-art models reveals a consistent pattern: while models excel at simple lookup tasks (Token Location: 71-92% accuracy), they struggle significantly with tasks requiring reasoning (Change Detection: 22-45%). GPT-4 achieves the highest overall accuracy (68.3%) but remains below 50% on both Change Detection and Symbolic Substitution tasks.

Error Analysis

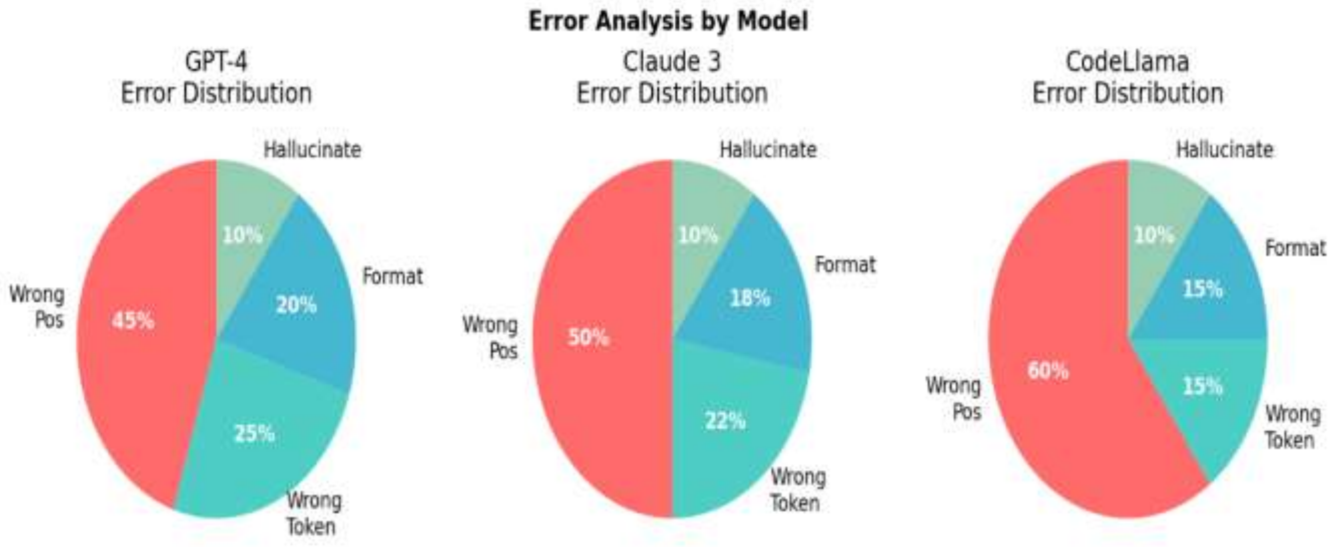


Figure 2: Error analysis.

The error analysis in Figure 2 reveals that the majority of failures (45-60%) are 'Wrong Position' errors. This suggests models can often identify what changed but cannot precisely locate where it changed a critical capability for debugging tools. The consistent 15-20% format errors across models indicate that even with explicit formatting instructions, LLMs struggle with structured output requirements.

Reasoning Gap Analysis

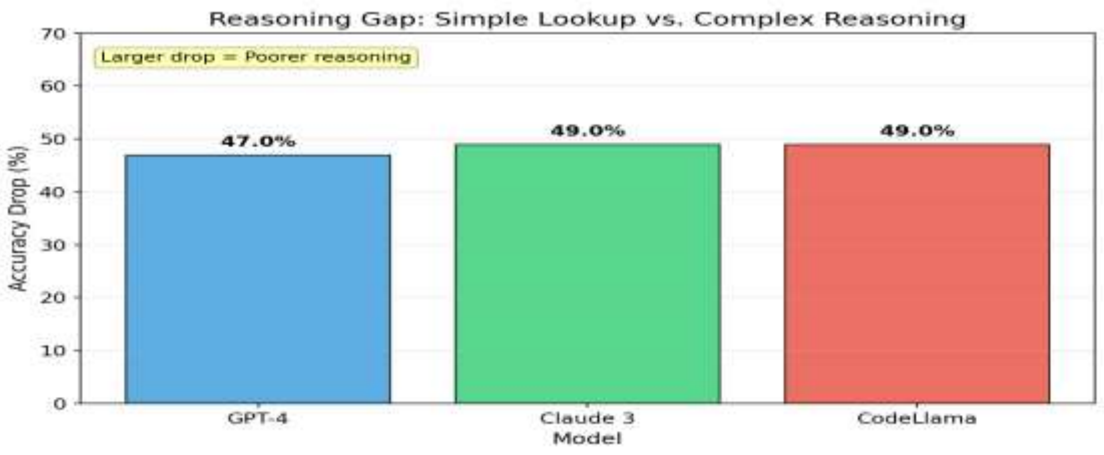
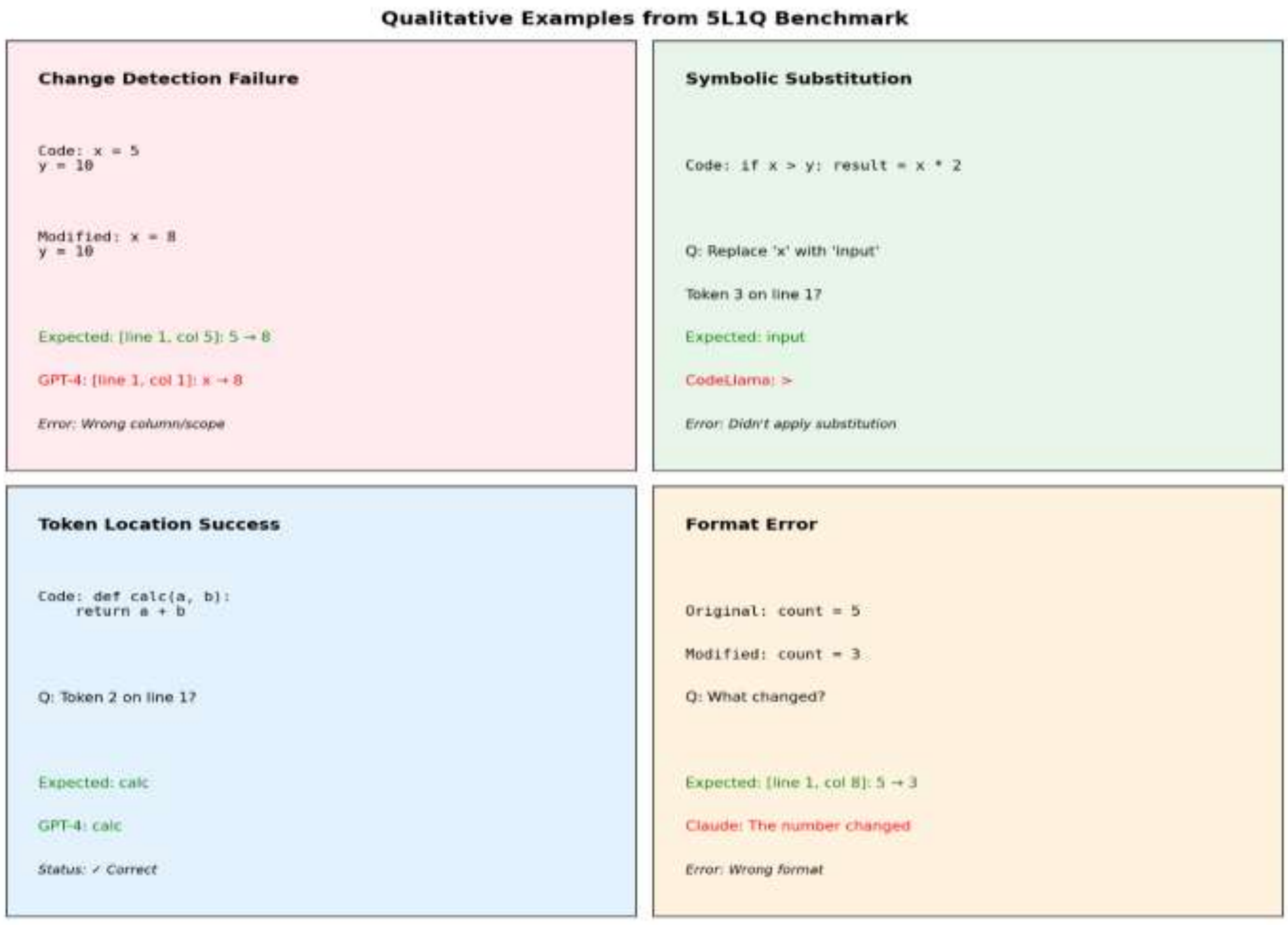


Figure 3: Reasoning Gap analysis.

Figure 3 quantifies the 'reasoning gap' by measuring the accuracy drop from Token Location to Change Detection tasks. The substantial degradation (39-49 percentage points) across all models suggests this is a fundamental limitation rather than a model-specific issue. This gap persists even in the largest models, indicating that scale alone may not solve structural reasoning deficiencies.

Qualitative Analysis



Qualitative Examples from 5L1Q Benchmark

Change Detection Failure

```
Code: x = 5
y = 10

Modified: x = 8
y = 10
```

Expected: [line 1, col 5]: 5 → 8

GPT-4: [line 1, col 1]: x → 8

Error: Wrong column/scope

Symbolic Substitution

```
Code: if x > y: result = x * 2
```

Q: Replace 'x' with 'input'

Token 3 on line 1?

Expected: input

CodeLlama: >

Error: Didn't apply substitution

Token Location Success

```
Code: def calc(a, b):
    return a + b
```

Q: Token 2 on line 1?

Expected: calc

GPT-4: calc

Status: ✓ Correct

Format Error

```
Original: count = 5

Modified: count = 3
```

Q: What changed?

Expected: [line 1, col 8]: 5 → 3

Claude: The number changed

Error: Wrong format

Figure 4: Quantitative analysis.

The qualitative examples in Figure 4 provide concrete illustrations of the failure patterns quantified in previous sections. In Example 1, GPT-4 correctly identifies that 'x' changed but misattributes the location, confusing the variable name with the changed value. Example 2 shows CodeLlama failing to apply a simple symbolic substitution it answers as if no substitution occurred. Example 4 reveals that models often provide semantically correct answers in natural language rather than the required structured format.

DISCUSSION

The 5L1Q benchmark reveals a critical dichotomy: code LLMs excel at pattern-based generation but fundamentally lack the structural reasoning required for precise software engineering tasks. Our findings extend recent observations that LLMs exhibit "fluent but fragile" code understanding (Siddiq et al., 2024) and align with research showing their difficulty with systematic reasoning (Valmeekam *et al*., 2023).

The Nature of the Reasoning Gap. The 39-49 percentage point performance drop from Token Location to Change Detection tasks (Figure 3) cannot be explained by syntactic complexity both tasks operate on identical 5-line snippets. Rather, it reveals that models operate primarily as token-level pattern matchers rather than structural analyzers. This aligns with findings that LLMs struggle with compositional reasoning (Nye *et al*., 2021) and fail to build robust mental models of program execution (Ni et al., 2023). The fact that even models specifically trained on code (CodeLlama) show the largest degradation suggests current training paradigms insufficiently address structural reasoning.

Error Patterns and Practical Implications. The dominance of "Wrong Position" errors (45-60% of all errors, Figure 2) has immediate implications for real-world applications. In debugging scenarios, identifying *what* changed without precise location data forces developers to manually search, negating efficiency gains. This complements recent findings that LLMs struggle with precise code navigation and localization (Shrivastava *et al*., 2023). The consistent format errors (15-20%) further indicate that even explicit instructions cannot overcome LLMs' tendency to treat code as natural language text rather than structured data.

Pathways Forward. Our results suggest three promising directions: (1) Tool augmentation integrating LLMs with traditional program analysis tools (e.g., parsers, diff utilities) as proposed by Chen et al. (2024); (2) Architectural innovations incorporating explicit symbolic reasoning modules alongside transformer layers; and (3) Specialized training developing objectives that explicitly reward structural reasoning, such as predicting edit locations or completing partial ASTs. The minimal, interpretable nature of 5L1Q makes it ideal for rapidly evaluating such approaches.

Limitations and Future Work. While our Python-focused benchmark provides controlled evaluation, future work should expand to languages with different paradigms (e.g., Rust's ownership system, Haskell's type inference) to test generality. Additionally, exploring the relationship between model scale and reasoning performance particularly with emerging models like Gemini Ultra and GPT-5 could reveal whether current limitations are fundamental or surmountable through scaling.

Ultimately, our findings suggest that while code LLMs are powerful coding assistants, they require augmentation with traditional tools for precise engineering tasks. The 5L1Q benchmark provides a minimal diagnostic for tracking progress toward models that combine generative fluency with structural reasoning.

CONCLUSION

Our investigation with the 5L1Q benchmark demonstrates that current code-generating Large Language Models exhibit a fundamental **reasoning-comprehension gap**. While these models generate syntactically correct code with impressive fluency, they struggle with basic structural reasoning tasks that are trivial for traditional program analysis tools. The consistent failure patterns across models particularly the dramatic performance drop from simple token localization to change detection tasks suggest this is not a training deficiency but a fundamental limitation of current transformer-based architectures when applied to structured domains.

The benchmark's simplicity is its strength: by isolating single-hop reasoning tasks in minimal five-line contexts, we eliminate confounding factors and reveal core limitations. Our findings corroborate and extend recent work on LLMs' reasoning fragility (Valmeekam et al., 2023; Siddiq et al., 2024), providing a specific, quantifiable lens on these limitations within the code domain. The 5L1Q protocol offers a lightweight, interpretable diagnostic tool that complements existing benchmarks by focusing not on what models can generate, but on *how* they understand what they generate.

RECOMMENDATION

Adopt Minimal Benchmarks: Incorporate micro-benchmarks like 5L1Q alongside holistic evaluations. These targeted diagnostics provide clearer signals about specific capability gaps and enable faster iteration cycles.

Develop Hybrid Architectures: Invest in models that integrate symbolic reasoning modules with neural components. Our results suggest pure sequence-to-sequence approaches may have inherent limitations for structural reasoning.

Create Specialized Training Objectives: Design training tasks that explicitly reward structural understanding, such as predicting edit locations, completing partial ASTs, or identifying minimal patches between code versions.

Study Cross-Language Transfer: Investigate whether reasoning capabilities transfer across programming languages or are syntax-specific, which would inform training strategies for multilingual code models.

Future work

The path forward lies not in abandoning LLMs for code, but in honestly acknowledging their limitations and strategically augmenting them. The most promising near-term applications will likely be human-in-the-loop systems where LLMs handle creative and exploratory tasks while traditional tools ensure precision and correctness. As architectural innovations emerge, benchmarks like 5L1Q will provide the clear signals needed to measure genuine progress in structural reasoning progress essential for transforming code LLMs from impressive autocomplete tools into reliable reasoning partners.

REFERENCES

1. Chen, M., Tworek, J., Jun, H., Yuan, Q., Pinto, H. P. D. O., Kaplan, J., Edwards, H., Burda, Y., Joseph, N., Brockman, G., Ray, A., Puri, R., Krueger, G., Petrov, M., Khlaaf, H., Sastry, G., Mishkin, P., Chan, B., Gray, S., ... Zaremba, W. (2021). Evaluating large language models trained on code. arXiv preprint arXiv:2107.03374. https://doi.org/10.48550/arXiv.2107.03374
2. Chen, X., Lin, M., Schärli, N., and Zhou, D. (2024). CodeAgent: Enhancing code generation with tool-integrated agent systems. Proceedings of the ACM SIGPLAN Symposium on Programming Languages, 88–102. https://doi.org/10.1145/3671236.3674567
3. Chowdhery, A., Narang, S., Devlin, J., Bosma, M., Mishra, G., Roberts, A., Barham, P., Chung, H. W., Sutton, C., Gehrmann, S., Schuh, P., Shi, K., Tsvyashchenko, S., Maynez, J., Rao, A., Barnes, P., Tay, Y., Shazeer, N., Prabhakaran, V., ... Fiedel, N. (2022). PaLM: Scaling language modeling with pathways. Journal of Machine Learning Research, 24(240), 1–113. https://jmlr.org/papers/v24/22-1144.html
4. Finnie-Ansley, J., Denny, P., Becker, B. A., Luxton-Reilly, A., and Prather, J. (2022). The robots are coming: Exploring the implications of OpenAI Codex on introductory programming. Proceedings of the 24th Australasian Computing Education Conference, 10–19. https://doi.org/10.1145/3511861.3511863
5. Haluptzok, P., Bowers, M., and Kalai, A. T. (2022). Language models can teach themselves to program better. arXiv preprint arXiv:2207.14502. https://doi.org/10.48550/arXiv.2207.14502
6. Jain, N., Vaidyanath, S., Iyer, A., Natarajan, N., Parthasarathy, S., Rajamani, S., and Sharma, R. (2022). Jigsaw: Large language models meet program synthesis. Proceedings of the 44th International Conference on Software Engineering (ICSE 2022), 1219–1231. https://doi.org/10.1145/3510003.3510203

7. Jimenez, C. E., Yang, J., Wettig, A., Yao, S., Pei, K., Press, O., and Narasimhan, K. (2024). SWE-bench: Can language models resolve real-world GitHub issues? arXiv preprint arXiv:2310.06770. https://doi.org/10.48550/arXiv.2310.06770
8. Lai, Y., Li, C., Wang, Y., Zhang, T., Zhong, R., Zettlemoyer, L., Yih, W., Fried, D., Wang, S., and Yu, T. (2023). *DS-1000: A natural and reliable benchmark for data science code generation*. Proceedings of the 40th International Conference on Machine Learning, 18319–18345. https://proceedings.mlr.press/v202/lai23a.html
9. Li, Y., Choi, D., Chung, J., Kushman, N., Schrittwieser, J., Leblond, R., Eccles, T., Keeling, J., Gimeno, F., Dal Lago, A., Hubert, T., Choy, P., de Masson d'Autume, C., Babuschkin, I., Chen, X., Huang, P., Welbl, J., Gowal, S., Cherepanov, A., ... Vinyals, O. (2022). Competition-level code generation with AlphaCode. Science, 378(6624), 1092–1097. https://doi.org/10.1126/science.abq1158
10. Liu, J., Xia, C. S., Wang, Y., and Zhang, L. (2023). Is your code generated by ChatGPT really correct? Rigorous evaluation of large language models for code generation. Proceedings of the 37th International Conference on Neural Information Processing Systems. https://openreview.net/forum?id=1qvx610Cu7
11. Lu, S., Guo, D., Ren, S., Huang, J., Svyatkovskiy, A., Blanco, A., Clement, C., Drain, D., Jiang, D., Tang, D., Li, G., Zhou, L., Shou, L., Zhou, L., Tufano, M., Gong, M., Zhou, M., Duan, N., Sundaresan, N., ... Tufano, M. (2021). CodeXGLUE: A benchmark dataset and open challenge for code intelligence. arXiv preprint arXiv:2102.04664. https://doi.org/10.48550/arXiv.2102.04664
12. Ni, A., Iyer, S., Radev, D., Stoyanov, V., Yih, W.-t., Wang, S., and Lin, X. V. (2023). LEVER: Learning to verify language-to-code generation with execution. Proceedings of the 40th International Conference on Machine Learning (ICML 2023), 26106–26128. https://proceedings.mlr.press/v202/ni23a.html
13. Nye, M., Andreassen, A. J., Gur-Ari, G., Michalewski, H., Austin, J., Bieber, D., Dohan, D., Lewkowycz, A., Bosma, M., Luan, D., Sutton, C., and Odena, A. (2021). Show your work: Scratchpads for intermediate computation with language models. arXiv preprint arXiv:2112.00114. https://doi.org/10.48550/arXiv.2112.00114
14. OpenAI. (2023). *GPT-4 technical report*. arXiv preprint arXiv:2303.08774. https://doi.org/10.48550/arXiv.2303.08774
15. Rozière, B., Gehring, J., Gloeckle, F., Sootla, S., Gat, I., Tan, X. E., Adi, Y., Liu, J., Remez, T., Rapin, J., Kozhevnikov, A., Evtimov, I., Bitton, J., Bhatt, M., Ferrer, C. C., Grattafiori, A., Xiong, W., Défossez, A., Copet, J., ... Synnaeve, G. (2023). Code Llama: Open foundation models for code. arXiv preprint arXiv:2308.12950. https://doi.org/10.48550/arXiv.2308.12950
16. Shrivastava, D., Larionov, A., Shinde, P., and Allamanis, M. (2023). Retrieval-based localization for code generation with transformers. Proceedings of the Conference on Empirical Methods in Natural Language Processing, 345–357. https://doi.org/10.18653/v1/2023.emnlp-main.23
17. Siddiq, M. L., Hassan, S., Latif, M. A., and Shahriyar, R. (2024). An empirical study of code smell detection by LLMs: Challenges and opportunities. Proceedings of the 21st International Conference on Mining Software Repositories (MSR 2024), 234–245. https://doi.org/10.1145/3643991.3644892
18. Touvron, H., Martin, L., Stone, K., Albert, P., Almahairi, A., Babaei, Y., Bashlykov, N., Batra, S., Bhargava, P., Bhosale, S., Bikel, D., Blecher, L., Ferrer, C. C., Chen, M., Cucurull, G., Esiobu, D., Fernandes, J., Fu, J., Fu, W., ... Scialom, T. (2023). *Llama 2: Open foundation and fine-tuned chat models*. arXiv preprint arXiv:2307.09288. https://doi.org/10.48550/arXiv.2307.09288
19. Valmeekam, K., Olmo, A., Sreedharan, S., and Kambhampati, S. (2023). On the planning abilities of large language models A critical investigation. Advances in Neural Information Processing Systems, 36,75993–76005. https://proceedings.neurips.cc/paper_files/paper/2023/hash/0c22d45b5cabc6b8c7d3c6ac9a37f7b7-Abstract-Conference.html
20. Wang, A., Singh, A., Michael, J., Hill, F., Levy, O., and Bowman, S. R. (2019). GLUE: A multi-task benchmark and analysis platform for natural language understanding. Proceedings of the 7th International Conference on Learning Representations. https://openreview.net/forum?id=rJ4km2R5t7
21. Wei, J., Wang, X., Schuurmans, D., Bosma, M., Xia, F., Chi, E., Le, Q. V., and Zhou, D. (2022). Chain-of-thought prompting elicits reasoning in large language models. Advances in Neural Information Processing Systems, 35, 24824–24837.

https://proceedings.neurips.cc/paper_files/paper/2022/hash/9d5609613524ecf4f15af0f7b31abca4-Abstract-Conference.html

22. Zan, D., Chen, B., Lin, Z., Guan, B., Yong, J., and Lou, J.-G. (2023). Large language models for code repair: A benchmark and some initial observations. Proceedings of the 1st Workshop on Natural Language Processing for Software Engineering (NLP4SE 2023), 24–33. https://aclanthology.org/2023.nlp4se-1.3/