# The Reliability Gap: A Multi-Dimensional Technical Audit of Memory Safety and Logical Integrity in LLM-Generated C Code

Ricardo Jarrin[†,*], Luiza Antonie[†], Ritu Chaturvedi[◇]
[†] University of Guelph
[‡] University of Guelph
[◇] University of Guelph

**Abstract**

The integration of Large Language Models (LLMs) into computer science education offers a scalable solution to the systemic enrollment crisis. However, their non-deterministic nature introduces a critical reliability gap, particularly in C programming - a domain characterized by manual memory management and a high risk of Undefined Behavior (UB). This multi-dimensional audit integrates static structural analysis with dynamic runtime checks of four 2026 frontier models: GPT-OSS 120B, Llama 3.3 70B, MoonshotAI Kimi K2, and Qwen 3 32B. Utilizing an automated evaluation pipeline, 1195 code generations across a gradient of complexity were subjected to static analysis and dynamic runtime instrumentation. Experimental results reveal a "Technical Reliability Gap": while models achieve high Compilation Success Rates (CSR), dynamic analysis identified a frequent incidence of "Definitely Lost" heap memory and "logical hangs". We offer a framework for trust calibration based on these breaking points, assisting educators in mitigating the "oracle hazards" of AI-mediated instruction.

**Keywords:** Large Language Models (LLMs), C Programming, Computer Science Education, Automated Code Evaluation, Memory Safety, Undefined Behavior (UB)

## 1. Introduction

Computer Science (CS) education is undergoing an "Algorithmic Turn" [1], driven by a systemic scalability crisis where surging enrollments outpace instructional resources [2]. To bridge this gap, institutions are increasingly deploying Large Language Models (LLMs) as "virtual assistants" [3, 4]. This automated support is particularly crucial in introductory C programming, where the steep learning curve of manual memory management creates a bottleneck for novice learners [5, 6]. However, unlike memory-safe languages, C demands rigorous precision; erroneous code frequently triggers Undefined Behavior (UB) - unpredictable runtime states that cause silent corruption or machine crashes - disrupting the pedagogical feedback loop [5]. We address this gap via a multi-dimensional audit of four 2026 frontier models: **GPT-OSS 120B**, **Llama 3.3 70B**, **Moonshot Kimi K2 v0905**, and **Qwen 3 32B**. Selected for their state-of-the-art performance [7] and open-weight accessibility [8], these models lack rigorous evaluation regarding memory safety and UB in C programming [9, 10]. We assess whether their accessibility compromises runtime integrity, offering the following contributions: (1) **Empirical Quantification of the Reliability Gap:** We demonstrate that high compile success rates (CSR) frequently mask deep logical failures, identifying 228 unique runtime violations across 1195 iterations. (2) **A "Functional Floor" for Systems Pedagogy:** We establish thresholds for Cyclomatic Complexity (CCN) - a metric quantifying the number of independent linear paths through a program's source code [11] - beyond which model reliability degrades [12]. This provides a calibration map for the deployment of LLMs in C programming curricula.

## 2. Technical Context and Reliability Challenges

LLM evaluation in C programming must move beyond binary compilation success to analyze dynamic runtime integrity. Standard compilers like GCC often overlook critical issues such as uninitialized variables or null-pointer dereferences [6, 9]. Furthermore, LLMs are predisposed to "hallucinations" manifesting as fabricated helper functions or "phantom libraries" [13, 14]. Existing benchmarks like HumanEval frequently suffer from inadequate

[*]rjarrin@uoguelph.ca

test coverage, yielding "false positive" results where code appears correct but contains latent vulnerabilities [15, 16]. Lastly, rigorous evaluation requires specialized instrumentation, such as Valgrind and UndefinedBehaviorSanitizer (UBSan), to detect "Definitely Lost" heap memory and illegal memory access operations [6, 9]. Through dynamic auditing, this research establishes a "functional floor" for AI-generated C code, exposing the pervasive "reliability gap" between syntactic adherence and logical correctness.

## 3. Methodology

### 3.1. Research Design and Experimental Setup

Evaluating the pedagogical utility of LLMs necessitates a transition from anecdotal observation to meticulous verification [17]. This methodology seeks to establish the minimum threshold of runtime integrity required for AI instructional aids [18]. This audit is driven by a primary inquiry: *To what extent can frontier models maintain memory safety and logical termination under high structural complexity (CCN>10)?* We selected four frontier models representing varied engineering philosophies: GPT-OSS 120B (Open-weights sparse Mixture-of-Expert (MoE)), Llama 3.3 70B (Open-weights dense Transformer), MoonshotAI Kimi K2 v0905 (MoE with latent reasoning), and Qwen 3 32B (High-performance dense Transformer) [12]. Leveraging the Groq API, interactions utilized precise version strings to prevent "benchmark leakage" [17]. The end-to-end codebase and 1195-response dataset are open-sourced: https://github.com/rjarrin/llm-reliability-gap.

### 3.2. Content Generation and Runtime Auditing

To eliminate "difficulty bias", we utilized a tripartite curriculum representing introductory C competencies (Pointers, Dynamic Memory, Data Structures) [17, 18]. In **Phase 1 (Problem Generation)**, three models (Llama 3.3, GPT-OSS, Kimi K2) collaboratively generated 300 distinct problems to prevent stylistic bias. Due to its lower parameter count, the highly-efficient Qwen 3 was reserved strictly for Phase 2.

In **Phase 2 (The Audit Pipeline)**, these 300 standardized problems were distributed to all four models. As illustrated in the Technical Appendix, this stateful multi-turn paradigm requires each model to traverse the remaining sequential stages: **(Step 2) C11 Solution Generation**; **(Step 3) Conceptual Explanation**; **(Step 4) Socratic Scaffolding** which implements the 'Socratic tutor' paradigm described by Liffiton et al. [19]; **(Step 5) Learning Objective Synthesis**; and **(Step 6) Automated Test Suite Generation**, outputting a machine-readable JSON block for dynamic validation.

The quantification of model performance is operationalized through a hierarchical auditing pipeline designed to measure the technical integrity of 1195 unique code iterations. Five iterations (from the initial 1200) were excluded due to the model failing to produce code within the specified JSONL syntax. The initial phase utilizes static analysis to characterize the structural quality and stylistic consistency of the generated solutions. Logical complexity is quantified via CCN using the `lizard` library (v1.20.0), providing an objective metric for branching paths. Stylistic adherence is monitored via `clang-format` (v1:14.0.0-1ubuntu1.1) based on the LLVM coding standard [6]. Moreover, the evaluation moves beyond static properties to establish a "functional floor" through dynamic runtime execution. Each solution is subjected to strict compilation using the `gcc` compiler (v.11.4.0) with mandatory academic flags (`-Wall -Werror -std=c11`), where any technical warning is classified as an objective failure. All successfully compiled binaries are then processed through a hybrid memory auditing pipeline using Valgrind (v.1:3.18.1-1ubuntu2) to bridge the "reliability gap" by identifying "Definitely Lost" heap memory and invalid read/write operations [15]. To penalize logical hangs (infinite loops), a strict 5-second timeout threshold was applied during execution. Finally, the analysis utilizes the UBSan tool (via GCC 11.4) to detect runtime illegalities, such as signed integer overflows, which constitute critical "oracle hazards" - the risk of students blindly accepting dangerous code because it was generated by an authoritative AI [9].

## 4. Evaluation

To establish a clear interpretative baseline for Table 1, generated solutions were evaluated across five metrics: **(1) CSR**: Percentage of solutions successfully compiled under GCC 11.4. **(2) Dynamic Reliability ($R$)**: Logical correctness penalized for hangs, calculated as $R = (\frac{n_{pass}}{T}) \times (1 - H)$, where $T = 5$ test cases and $H$ is a binary hang penalty. **(3) Memory Safety**: Subset of reliable solutions ($R = 1.0$) passing Valgrind and UBSan without leaks or undefined behavior. **(4) CCN**: *Avg* and *Max* control flow complexity. **(5) Runtime Violations**: Absolute frequencies of Hangs, un-freed heap memory (Leaks), and fatal memory access violations (UBSan).

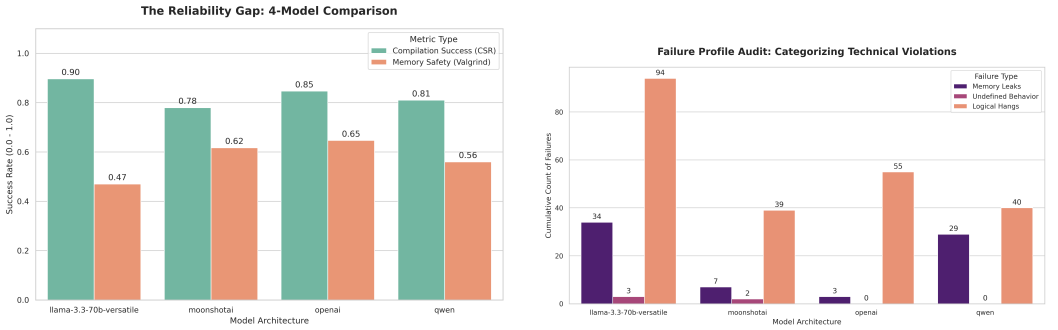
### 4.1. Technical Integrity and Dynamic Reliability

The foundational audit of the 1195 iterations established a significant "Technical Reliability Gap" between syntactic fluency and runtime execution integrity. Table 1 summarizes the high-level performance across all four models, quantifying the frequency of critical failure profiles identified through dynamic instrumentation.

| Model Architecture | CSR (↑) | Safety (↑) | Max CCN | Avg CCN | Hangs (↓) | Leaks (↓) | UBSan (↓) |
|---|---|---|---|---|---|---|---|
| Llama 3.3 70B | 0.897 | 0.470 | 11.11 | 5.46 | 94 | 34 | 3 |
| MoonshotAI Kimi K2 | 0.780 | 0.617 | 14.11 | 5.75 | 39 | 7 | 2 |
| OpenAI GPT-OSS 120B | 0.847 | 0.647 | 17.69 | 7.29 | 55 | 3 | 0 |
| Qwen 3 32B | 0.810 | 0.560 | 13.52 | 6.37 | 40 | 29 | 0 |

*Table 1.* Consolidated Technical Performance Audit (1195 Iterations). Indicators denote desired outcomes: higher is better (↑) for success rates, while lower is better (↓) for runtime violations.

Crucially, compilation success inversely correlates with dynamic reliability. Llama 3.3 70B achieved the highest CSR (0.897), suggesting surface-level syntactic fluency. However, it demonstrated the lowest dynamic reliability (0.470 safety rate). As the massive gap in Figure 1a illustrates, this model exemplifies a "syntactic trap" where code satisfies standard compilers but fails during execution [6]. In contrast, OpenAI (GPT-OSS 120B) demonstrated the highest level of runtime execution integrity. Despite producing the most structurally complex code (Avg CCN = 7.29), it maintained a superior memory safety rate of 0.647 and nearly eliminated heap memory deallocation failures (only 3 identified leaks). This performance suggests that GPT-OSS may better capture the data lifecycle requirements of C programming than other generalist counterparts [12]. MoonshotAI (Kimi K2) achieved the fewest logical hangs (39), potentially reflecting its latent reasoning architecture's ability to self-verify terminal logic before generation.




*(a)* CSR vs. Memory Safety

*(b)* Runtime Violations

*Figure 1.* Technical Integrity Audit.

Llama 3.3's reliability degraded significantly at lower complexity levels, suggesting a vulnerability in handling the manual memory "notional machine" [20]. While Llama excelled at surface-level syntax [15], its failure profile (Figure 1b) reveals it caused nearly half of all identified memory leaks and the majority of UB triggers in the dataset.

### 4.2. **Complexity Stress Test and The Reliability Ceiling**

To identify where model reasoning degrades, we correlated Max CCN with binary safety outcomes (Figure 2). A "Reliability Ceiling" emerges across all architectures. Defined as the complexity threshold beyond which a model's capacity to maintain internal memory states collapses, this ceiling marks a "cognitive limit". Below this limit, models successfully juggle memory rules; above it, they become overwhelmed by branching paths and lose track of essential details (e.g., freeing variables) despite intact syntax. To empirically determine these thresholds, we aggregated the 1195 generated functions into discrete complexity intervals and compared the volume of safe executions ('PASS') against failures ('FAIL'). This categorical distribution, visualized in Figure 2, reveals the exact structural density where performance transitions from stability to systemic failure. The disparity is most visible when comparing models. As shown in the top-left quadrant of Figure 2, Llama 3.3 70B exhibits a distinct inversion of reliability. While it maintains parity in the low-complexity band (CCN 6-10), the 'FAIL' count (red bars) sharply overtakes successful generations once complexity exceeds the CCN>10 threshold. This visual trend is statistically confirmed in the Technical Appendix. Llama 3.3 sees its success rate drop from 51.3% in the low-complexity tier to 42.7% in the high-complexity tier. In contrast, GPT-OSS 120B demonstrates superior cognitive endurance, maintaining a robust volume of 'PASS' outcomes (blue bars) even into the highly complex 16-20 CCN range, retaining a 62.3% success rate even on complex tasks.

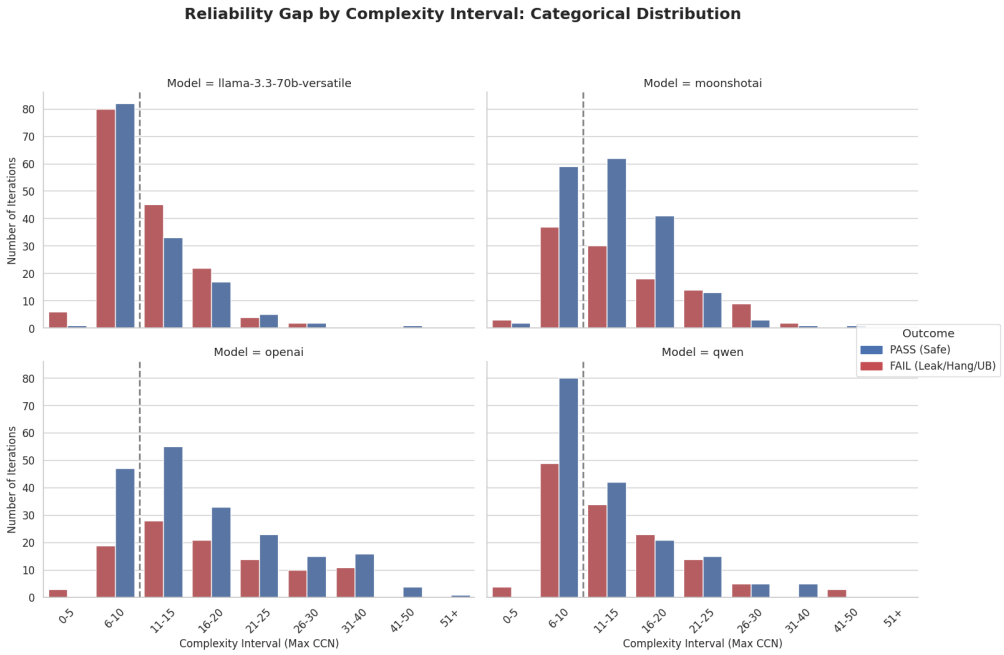



*Figure 2.* Reliability Gap by Complexity Interval: The grouped bar chart illustrates the count of PASS (Blue) vs. FAIL (Red) outcomes across incremental complexity bins. Note the sharp increase in Llama 3.3 failures beyond the CCN 10 threshold.

## 5. **Discussion**

Contrary to the assumption that "clean" code implies safety, our analysis revealed a "Style-Safety Paradox" (Figure 3). High stylistic adherence frequently masked catastrophic failures. Llama 3.3's high CSR paired with a low safety rate unmasks a "Training-Audit Divergence" [10]. The model performs "syntactic mimicry", prioritizing the local statistical likelihood of a token over global logical necessities (e.g., memory guards) [5, 21].

For example, models frequently generated syntactically flawless struct allocations (e.g., `Node* n = malloc(sizeof(Node)); n->data = val;`) but immediately dereferenced the pointers without mandatory `NULL` checks, satisfying the compiler but triggering runtime

violations. While it successfully captures the "shape" of systems code, it fails to maintain the "state" of the heap [15, 22]. This uncoupling of aesthetics from integrity represents a significant "Oracle Hazard" [23]. Students reviewing "clean" LLM-generated code may falsely assume it is safe to execute [6]. This necessitates a shift in evaluation focus from static "code-as-text" to dynamic "code-as-process" [5].


*Figure 3.* The Style-Safety Paradox: Style adherence does not predict memory safety.

## 6. Conclusion

This study conducted a rigorous technical audit of 1195 LLM-generated iterations to establish a "functional floor" for AI-mediated C programming instruction. Our findings identify a pervasive **Technical Reliability Gap**: high syntactic adherence (up to 0.897 CSR) frequently masks catastrophic runtime failures, including 228 unique violations across memory leaks, logical hangs, and UB. We conclude that surface-level compilation is a methodologically insufficient proxy for pedagogical safety in the C programming domain.

### 6.1. Limitations and Future Work

Utilizing LLMs to generate test suites introduces potential self-evaluation bias; future iterations must validate findings against human-authored ground truth datasets. Furthermore, evaluating emergent reasoning architectures (e.g., DeepSeek-R1) across wider tasks will be necessary. Ultimately, there is a critical need for real-time "Pedagogical Guardrails" integrating dynamic instrumentation (Valgrind-UBSan) directly into student IDEs, shifting LLM usage toward a transparent co-pilot paradigm.

## Acknowledgements

We thank the anonymous reviewers for their constructive feedback. This work was supported in part by the Braithwaite Conference Travel Grant.

## References

[1] P. Denny, V. Kumar, and N. Giacaman. “Conversing with Copilot: Exploring Prompt Engineering for Solving CS1 Problems Using Natural Language”. In: *Proc. ACM SIGCSE*. ACM, 2023, pp. 1136–1142. DOI: 10.1145/3545945.3569823.

[2] B. Sheese, M. Liffiton, et al. “Patterns of Student Help-Seeking When Using a Large Language Model-Powered Programming Assistant”. In: *Proc. ACE*. ACM, 2024, pp. 49–57. DOI: 10.1145/3636243.3636249.

[3] H. Kumar, I. Musabirov, et al. "Guiding Students in Using LLMs in Supported Learning Environments". In: *Proc. ACM Hum.-Comput. Interact.* 8.CSCW2 (2024), pp. 1–30. DOI: 10.1145/3687038.

[4] T. Feng, S. Liu, and D. Ghosal. "CourseAssist: Pedagogically Appropriate AI Tutor for Computer Science Education". In: *Proc. ACM CompEd*. ACM, 2024, pp. 310–311. DOI: 10.1145/3649409.3691094.

[5] E. Wen, S. Ma, et al. "KernelVM: Teaching Linux Kernel Programming through a Browser-Based Virtual Machine". In: *Proc. ACM SIGCSE*. ACM, 2025, pp. 1204–1210. DOI: 10.1145/3641554.3701831.

[6] A. Taylor, A. Vassar, et al. "dcc –help: Transforming the Role of the Compiler by Generating Context-Aware Error Explanations". In: *Proc. ACM SIGCSE*. ACM, 2024, pp. 1314–1320. DOI: 10.1145/3626252.3630822.

[7] Y. Rong, T. Du, et al. "Integrating LLM-based code optimization with human-like exclusionary reasoning for computational education". In: *J. King Saud Univ. Comput. Inf. Sci.* 37.5 (2025), p. 87. DOI: 10.1007/s44443-025-00074-7.

[8] S. Liu, Z. Yu, et al. "Can Small Language Models With RAG Replace Large Language Models When Learning Computer Science?" In: *Proc. ACM ITiCSE*. ACM, 2024, pp. 388–393. DOI: 10.1145/3649217.3653554.

[9] T. Doumler and J. Berne. "A framework for systematically addressing undefined behaviour in the C++ Standard". In: ().

[10] B. Zeng, Q. Zhang, et al. *Inducing Vulnerable Code Generation in LLM Coding Assistants*. 2025. DOI: 10.48550/arXiv.2504.15867.

[11] A. Mastropaolo, L. Pascarella, et al. "On the Robustness of Code Generation Techniques: An Empirical Study on GitHub Copilot". In: *Proc. 45th IEEE/ACM ICSE*. IEEE, 2023, pp. 2149–2160. DOI: 10.1109/ICSE48619.2023.00181.

[12] D.-S. Huang, H. Chen, et al., eds. *Advanced Intelligent Computing Technology and Applications: 21st Int. Conf., ICIC 2025, Proceedings, Part X*. Vol. 2573. Springer, 2025. DOI: 10.1007/978-981-96-9994-0.

[13] D. Saunders, W.-C. Li, and T. Wang. "From Turing Test to Chinese Room Argument: How to Apply Artificial Intelligence in Aviation". In: *Transp. Res. Procedia* 88 (2025), pp. 270–277. DOI: 10.1016/j.trpro.2025.05.033.

[14] Z. Li, Z. Wang, et al. "Retrieval-augmented generation for educational application: A systematic survey". In: *Comp. & Ed.: AI* 8 (2025), p. 100417. DOI: 10.1016/j.caeai.2025.100417.

[15] S. Gopali, S. Siami-Namini, et al. "The performance of the LSTM-based code generated by Large Language Models (LLMs) in forecasting time series data". In: *Nat. Lang. Process. J.* 9 (2024), p. 100120. DOI: 10.1016/j.nlp.2024.100120.

[16] J. Prather, P. Denny, et al. "The Robots Are Here: Navigating the Generative AI Revolution in Computing Education". In: *Proc. ITiCSE-WGR*. ACM, 2023, pp. 108–159. DOI: 10.1145/3623762.3633499.

[17] M. Ali, P. Rao, et al. "Using Benchmarking Infrastructure to Evaluate LLM Performance on CS Concept Inventories: Challenges, Opportunities, and Critiques". In: *Proc. ACM ICER*. ACM, 2024, pp. 452–468. DOI: 10.1145/3632620.3671097.

[18] M. A. Foughali. "Some thoughts on teaching introductory programming and the first language dilemma". In: *Proc. Koli Calling*. ACM, 2023, pp. 1–8. DOI: 10.1145/3631802.3631812.

[19] M. Liffiton, B. E. Sheese, et al. "CodeHelp: Using Large Language Models with Guardrails for Scalable Support in Programming Classes". In: *Proc. Koli Calling*. ACM, 2023, pp. 1–11. DOI: 10.1145/3631802.3631830.

[20] J. Hoobergs. "Removing the Notional Machine Discrepancy". In: *Proc. Koli Calling*. ACM, 2023, pp. 1–2. DOI: 10.1145/3631802.3631837.

[21] M. Liu and F. M'Hiri. "Beyond Traditional Teaching: Large Language Models as Simulated Teaching Assistants in Computer Science". In: *Proc. ACM SIGCSE*. ACM, 2024, pp. 743–749. DOI: 10.1145/3626252.3630789.

[22] J. Lee, Y. Kim, et al. "Taming undefined behavior in LLVM". In: ().

[23] K. Z. Zhou, Z. Kilhoffer, et al. "Ethics, Governance, and User Mental Models for Large Language Models in Computing Education". In: *XRDS* 31.1 (2024), pp. 46–51. DOI: 10.1145/3688089.

# Appendix A. Technical Appendix

## A.1. Supplementary Methodology

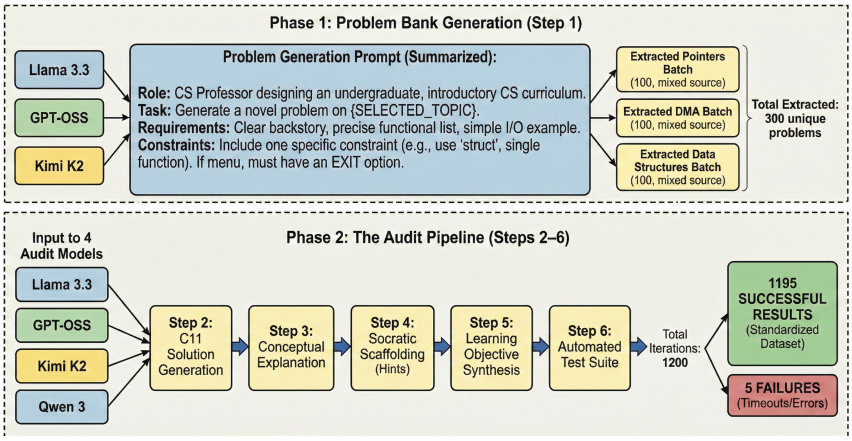



*Figure 4.* The Multi-Turn Content Generation Protocol (Detailed Workflow)

## A.2. Supplementary Results Data

| Model | Complexity Tier | Pass/Total | Success Rate (%) |
|---|---|---|---|
| Llama 3.3 70B | CCN $<$ 10 (Simple) | 77 / 150 | 51.3% |
| | CCN $>$ 10 (Complex) | 64 / 150 | 42.7% |
| MoonshotAI Kimi K2 | CCN $<$ 10 (Simple) | 40 / 72 | 55.6% |
| | CCN $>$ 10 (Complex) | 142 / 223 | 63.7% |
| OpenAI GPT-OSS 120B | CCN $<$ 10 (Simple) | 37 / 48 | 77.1% |
| | CCN $>$ 10 (Complex) | 157 / 252 | **62.3%** |
| Qwen 3 32B | CCN $<$ 10 (Simple) | 69 / 110 | 62.7% |
| | CCN $>$ 10 (Complex) | 99 / 190 | 52.1% |

*Table 2.* Impact of Complexity on Success Rates: Comparing performance below and above the industry-standard complexity threshold (CCN = 10).