Uncovering Systematic Failures of LLMs in Verifying Code Against Natural Language Specifications

Haolin Jin*, Huaming Chen*
*School of Electrical and Computer Engineering, The University of Sydney, Sydney, Australia
{haolin.jin, huaming.chen}@sydney.edu.au

***Abstract*—Large language models (LLMs) have become essential tools in software development, widely used for requirements engineering, code generation and review tasks. Software engineers increasingly rely on LLMs to assess whether system code implementation satisfy task requirements, thereby enhancing code robustness and accuracy. However, it remains unclear whether LLMs can reliably determine whether the code complies fully with the given task descriptions, which is usually natural language specifications. In this paper, we uncover a systematic failure of LLMs in evaluating whether code aligns with natural language requirements. Specifically, using widely adopted benchmarks, we employ unified prompts to judge code correctness. Our results reveal that LLMs frequently misclassify correct code implementations as either "not satisfying requirements" or containing potential defects. Surprisingly, more complex prompting, especially when leveraging prompt engineering techniques involving explanations and proposed corrections, leads to higher misjudgment rate, which highlights the critical reliability issues in using LLMs as code review assistants. We further analyze the root causes of these misjudgments, and propose two improved prompting strategies for mitigation. For the first time, our findings reveals unrecognized limitations in LLMs to match code with requirements. We also offer novel insights and practical guidance for effective use of LLMs in automated code review and task-oriented agent scenarios.**

***Index Terms*—Large Language Models, Code Review, Prompt Engineering, Code Understanding**

I. Introduction

As large language models (LLMs) have demonstrated increasing capability in code synthesis [1], [2], a growing body of research and tooling efforts have begun exploring their application for automated code review and verification [3]–[5]. Traditional code reviews require developers to manually verify the alignment between code logic and requirements, a process that is both time consuming and error prone. LLMs show significant potential to reduce this burden by automating code assessments and suggesting improvements during the code-review process. For instance, recent studies have leveraged models such as GPT-4o to evaluate code submissions and determine whether they meet quality standards or require revisions [6].

In software engineering, verifying that source code aligns with its task requirements remains a challenge [7], [8]. Requirements engineering is widely recognized as crucial to clearly defining, understanding, and aligning project objectives with stakeholder needs, thereby reducing risks associated with errors, delays, and project failures [9]. Recent studies start to investigate the potential of LLMs to bridge this gap. Advanced models such as GPT-4o have shown promising accuracy in identifying unmet requirements from textual descriptions [10], indicating their potential as reliable "virtual reviewers" even without test cases or formal implementations. To further improve review effectiveness, researchers have explored self-critical mechanisms within LLMs, such as the Self-Refine framework, where models iteratively critique and refine their outputs without additional training [11].

Ideally, LLMs should accurately understand functional requirements described in natural language and reliably judge whether provided code satisfies these requirements, assisting developers in identifying defects or confirming correctness. However, in scenarios lacking test cases or reference implementations, the reliability of LLMs in performing such "description-to-code" evaluations remains unclear. In this work, we investigate whether LLMs can correctly determine code correctness when provided with precise task descriptions and correct implementations. Our preliminary experiments reveal a concerning phenomenon: LLMs frequently issue false negative judgments, incorrectly concluding that correct implementations fail to meet stated requirements. This systematic failure may arise from inherent biases and hallucinations within LLMs [12], or from improper prompt designs. Models tend to overly critique correct code, resulting in a high false negative rate. Surprisingly, our extended experiments involving varied prompt designs [13] indicate that increasing the prompt complexity, such as requiring explicit explanations and suggested corrections, counterintuitively leads to higher rates of misjudgment. This finding contradicts the common assumption that incorporating explanatory steps typically enhances the reasoning and accuracy of LLMs. Instead, detailed prompts may inadvertently introduce biases toward excessive fault finding, causing models to detect non-existent errors in otherwise correct implementations.

In this paper, we present an empirical investigation with observations that carry significant implications for automated software engineering practices. Frequent false negatives by LLMs acting as automated code review assistants can severely impact their practical utility, overwhelming developers with misleading feedback and potentially diminishing trust in these tools. Particularly within automated code generation pipelines,

2643-1572/25/$31.00 ©2025 IEEE
DOI 10.1109/ASE63991.2025.00323

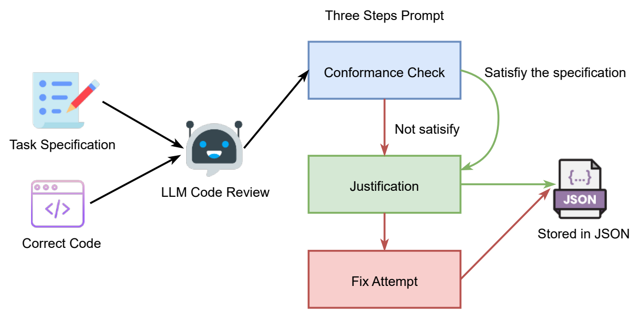

Fig. 1. Our Overall Evaluation Workflow.

where an LLM may evaluate its own generated code [14], unreliable assessments will compromise the overall effectiveness of automated engineering workflow. Therefore, understanding why LLMs systematically misjudge correct implementations and identifying strategies to mitigate these failures becomes crucial. By providing these novel insights, we aim to raise community awareness regarding the limitations of LLMs in code evaluation and to inform future research efforts. Overall, this work makes three key contributions:

1) **False negatives discovery** – We reveal that LLMs frequently misjudge correct code as failing to meet requirements, indicating their bias towards over-correction rather than accurate verification.
2) **Empirical evaluation** – Using the state-of-the-art models, our experiments highlight the key weaknesses in LLMs' ability to assess code conformance with natural language requirements.
3) **Mitigation proposals with improved prompt strategies** – We propose a mitigation strategy that leverages refined prompt designs to reduce false negatives and improve assessment reliability.

II. Methodology

A. Research Question

To investigate the causes and implications of LLMs' judgments in code evaluation, we propose the following research questions:

- **RQ1:** Without test cases, to what extent can LLMs reliably assess whether a program conforms to its specification?
- **RQ2:** How does prompt design affect the accuracy of LLM-based code evaluation?
- **RQ3:** What factors cause LLMs to incorrectly classify correct code as faulty, and can these misjudgments be effectively mitigated through improved prompt design?

These research questions cover three key dimensions: understanding the root causes of LLMs' misjudgments, exploring potential mitigation solutions, and evaluating their effectiveness. Subsequent sections of this paper will present experiments and discussions addressing each of these questions.

B. Preliminary Experiment Setup

To address RQ1 and RQ2, we designed a series of experiments to evaluate the performance of different LLMs on code requirement conformance tasks and to analyze the impact of prompt design. For this purpose, we selected three widely used code evaluation benchmarks: HumanEval [15], MBPP [16], and QuixBugs [17], providing over 700 samples for evaluation in total. In this work, we currently focus on Python as the main programming language.

HumanEval provides manually written problem descriptions, corresponding reference implementations and test cases. We used the provided descriptions and correct reference implementations as model inputs, instructing models to judge if the implementation met the described requirements. MBPP includes programming tasks described in natural language along with correct implementations, and we similarly used these correct code implementations with their requirement descriptions for testing. QuixBugs contains a small set of algorithm problems with known defects. Through these diverse datasets, we covered various scenarios ranging from competitive programming solutions to real-world bug fixes, ensuring the generalizability of our observations.

Regarding model selection, we evaluated three mainstream LLMs: GPT-4o, Claude-3.5-sonnet, and Google Gemini-2.0 flash. These models were chosen as representatives of state-of-the-art industry LLMs, recognized for their strong general purpose capabilities in code understanding and generation tasks. Their superior performance makes them promising candidates for automated code review [18]–[20]. However, precisely because of their capabilities, we aimed to examine if they exhibit consistent error patterns. Moreover, these models have consistently been among the top choices for code review and code synthesis tasks in previous works [21]–[23].

Fig. 1 illustrates the overall workflow: the LLM firstly receives the task specification and corresponding correct code implementation, guided by our designed prompts. The LLM will then assess whether the code fulfills the stated requirements, initially performing a binary classification ("Yes" if compliant, "No" if non-compliant). These judgments are systematically recorded into structured JSON files for consistency and ease of data analysis. In cases where the LLM incorrectly classifies a correct implementation as non-compliant (a false negative judgment), it is further instructed to articulate a detailed rationale for its assessment and subsequently propose a revised version of the code. This approach enables the collection of enriched information for following analysis of misjudgments.

C. Prompt Design

To ensure a fair comparison, we developed a unified three-step prompt template for all models inspired by recent advances in prompt engineering [24]–[28]. These steps work in tandem to pivot the investigation of LLMs' judgments in code evaluation.

- **Judgment:** Read the natural language requirement and the provided code implementation, then answer the ques-

TABLE I
REQUIREMENT CONFORMANCE RECOGNITION RATE (RCRR ↑, IN %) OF THREE LLMS ON THE HUMANEVAL, MBPP, AND QUIXBUGS BENCHMARKS UNDER THREE PROMPTING APPROACHES (DIRECT, DIRECT + EXPLAIN, AND THREE STEP FULL PROMPT). HIGHER RCRR INDICATES BETTER RECOGNITION OF CORRECT IMPLEMENTATIONS AS SATISFYING THEIR TASK REQUIREMENTS.

Model	**Approch**	**HumanEval**	**MBPP**	**Quixbugs**
		RCRR ↑	**RCRR ↑**	**RCRR ↑**
GPT-4o	Direct	52.4	63.7	62.5
	Direct + Explain	22.0	38.2	50.0
	Full	11.0	30.9	30.0
Gemini-2.0-flash	Direct	59.1	66.5	72.5
	Direct + Explain	56.7	64.9	75.0
	Full	53.0	61.5	72.5
Claude-3-5-sonnet	Direct	78.0	56.9	80.0
	Direct + Explain	69.5	66.1	77.5
	Full	67.0	55.3	75.0

tion, “Does the code meet the requirement?” with a simple answer “Yes” or “No”.

- **Explanation:** Request the model to provide a rationale for its judgment, explaining why the code may not meet the requirements, with a detailed analysis of discrepancies between code logic and requirements.
- **Fix:** If the model judged the code as not meeting the requirement, it was instructed to provide corrected code after the explanation; otherwise, this step could be skipped or marked as unnecessary.

Multi-stage prompting strategies have been applied for code evaluation, aiming to enhance the reasoning performance of LLMs. One common approach first instructs the model to perform a step-by-step analysis of the code’s functionality, and then asks it to summarize the analysis with a binary decision (correct or not). This often outperforms simple prompts by encouraging reasoning through both requirements and code logic before judgment [29]. Such findings provide evidence-based support for our design of structured, explanation-driven prompting in code evaluation. Similar two-phase prompting method have also been applied for program repair. For example, [30], [31] prompt LLMs to first generate a chain-of-thought diagnosis of the bug, followed by providing a patch for fixing. In our prompt, the model is initially asked to explain any discrepancies between the code and the requirements. It will use related reasoning to propose a corrected version of the code if the initial answer is negative.

III. OBSERVATION AND PRELIMINARY RESULT

To address our research questions, our experiments consider different dimensions for LLMs: accuracy of judgment against ground truth labels, and whether rationales identified real defects or hallucinations. To isolate the impact of prompt complexity, we designed three progressively more complex prompting strategies: (1) Direct (judgment only), (2) Direct+Explain (judgment with reasoning), and (3) Full (three-step prompt consisting of judgment, explanation, and suggested repair). We then compared the misjudgement rates across these conditions. Our evaluation metric was the Requirement-Conformance Recognition Rate (RCRR), representing the accuracy with which a model correctly judges whether a given code satisfies a natural-language requirement. The metric is formally defined in Equation 1:

$$\text{RCRR} = \frac{N_{\text{correct judgment}}}{N_{\text{Total correct\_code samples}}} \tag{1}$$

Overall, our results indicate an inverse relationship between prompt complexity and model performance. For instance, on the HumanEval dataset, GPT-4o’s RCRR declined markedly from 52.4% with the simple direct prompt to merely 11.0% using the full three-step prompt. It results in a substantial reduction of 41.4%. Similar trends were observed on the MBPP and QuixBugs datasets, with drops of 32.8% and 32.5%, respectively. Other models exhibited the same pattern: accuracy consistently decreased as explanation step and repair suggestions were introduced into the prompt. Notably, simpler “judgment-only” prompts often yielded the best performance, particularly on HumanEval and MBPP datasets. The consistency of these declines across diverse models and datasets underscores the robustness and generalizability of this phenomenon.

IV. FINDING EVALUATION

Next, we examine the underlying reasons why multi-step prompts negatively impact accuracy. Our analysis identifies an increased rate of false negatives as the primary factor: when models are prompted to explain and propose fixes, models become significantly more prone to incorrectly classifying conforming code as erroneous. In other words, introducing reasoning and correction steps induces an overcorrection bias: LLMs tend to assume flaws exist and suggest unnecessary modifications, even when the provided implementation is already correct. This trend is particularly surprising, given contrasting findings from prior research that generally support step-by-step prompting to improve logical reasoning accuracy. For instance, Wei et al. demonstrate that intermediate reasoning steps significantly boost performance on complex reasoning tasks [32]. In contrast, a recent work also suggest similar bias may exist in the provided code snippet [33]. Our study extends this observation by establishing a more general scenario, in which we highlight a specific domain where the “step-by-step” approach can degrade rather than improve model performance in code requirement verification.

A. *RQ1: LLM Judgment Accuracy*

Regarding RQ1 (the capability of LLMs to judge code correctness without test cases), we find current LLMs exhibit limited effectiveness, with RCRR ranging from only **52%** to **78%**. This indicates that automated code-to-requirement conformance checking remains an unresolved challenge.

B. *RQ2: Prompt Effectiveness*

Addressing RQ2 (the impact of prompt design), we observe a significant effect of prompting strategy on model performance. In particular, the comprehensive three-step prompt

TABLE II
RCRR FOR LLMS ON THE HUMANEVAL, MBPP, AND QUIXBUGS BENCHMARKS UNDER THREE PROMPTING STRATEGIES. ORIGINAL FOR THE DIRECT PROMPT AND THE TWO-PHASE REFLECTIVE AND BEHAVIORAL COMPARISON PROMPTS MARKEDLY OUTPERFORM THE ORIGINAL PROMPT ACROSS ALL MODELS AND DATASETS.

Model	**Approch**	**HumanEval**	**MBPP**	**Quixbugs**
		RCRR ↑	**RCRR ↑**	**RCRR ↑**
GPT-4o	Original	52.4	63.7	62.5
	Two-Phase Reflective	72.0	48.3	70.0
	Behavioral Comparison	**85.4**	**68.9**	**90.0**
Gemini-2.0-flash	Original	59.1	66.5	72.5
	Two-Phase Reflective	67.7	**69.1**	82.5
	Behavioral Comparison	**73.8**	62.5	85.0
Claude-3-5-sonnet	Original	78.0	56.9	80.0
	Two-Phase Reflective	**82.9**	**66.1**	77.5
	Behavioral Comparison	78.0	62.7	**82.5**

(judgment, explanation, and fix) causes substantial accuracy reductions of approximately 20% to 40% compared to simpler direct prompts. GPT-4o exhibits the highest sensitivity to prompt complexity, with accuracy declines exceeding 40% on the most challenging tasks. In contrast, Gemini shows relatively greater stability, while Claude maintains comparatively better robustness across certain datasets. This variability indicates that different model architectures or training methodologies respond differently to chain-of-thought prompting strategies, though all exhibit some degree of performance degradation. Prior studies similarly report that even minor prompt modifications, such as slight variations in problem descriptions, can significantly affect LLM performance on code tasks [34].

C. *RQ3: Analysis and Mitigation*

Our analysis identifies a pronounced "over-correction" tendency as the primary factor driving performance decline, when LLMs are prompted to explain and fix code, the model is required to justify its answer and then patch the code, it develops a bias toward assuming defects exist even when the implementation is already correct and the code already satisfies requirements, resulting in unnecessary and incorrect modifications. To address this issue, we developed two alternative prompting strategies, the Two-Phase Reflective Prompt and the Behavioral Comparison Prompt, which eliminate the mandatory "fix" step and clearly separate requirement comprehension from code auditing, thus reducing the tendency toward premature fault finding.

Two-Phase Reflective Prompt
"Phase 1 – Extract Contract Obligations." Read the requirement and extract its intended functional obligations. List the main things the code is expected to do, including input-output behavior, edge-case handling, and any conditions or constraints.
"Phase 2 – Audit and Verdict." Carefully examine the code and check whether it fulfills each obligation you listed above. If an obligation is fully met, mark it as Satisfied; if it is partially or incorrectly implemented, mark it as Not satisfied. Based on the audit, decide: Does the code fulfill all essential obligations from the requirement?

1) Two-Phase Reflective Prompt: Draws inspiration from contract based reviews in software engineering [35]. It is structured in two stages: initially, the model explicitly extracts functional obligations from natural language requirements, presenting them clearly as bullet pointed lists to avoid over interpretation. In the second stage, the model audits the code implementation against each previously extracted obligation. This structured approach encourages the model to focus on actual functional requirements rather than superficial implementation details or stylistic differences, thereby effectively reducing the over correction bias.

2) Behavioral Comparison Prompt: Inspired by traditional black-box software testing, separately summarizes the core functionality, input-output behaviors, and boundary conditions from the requirement description and then independently summarizes the functionality actually implemented in the code. Finally, these two behavior descriptions are explicitly compared to determine consistency.

Behavioral Comparison Prompt
Please summarize the main functions and boundary conditions that the program should implement. Then read the code and describe what functions the code actually completes and how the key steps are implemented. Finally, compare the code behavior with the requirements point-by-point to determine whether they are consistent.

Table II shows significant improvements in RCRR across models using the Two-Phase Reflective Prompt. For instance, GPT-4o's performance on HumanEval increased from 52.4% with the original prompt to 72.0%, an improvement of 61%, with Behavioral Comparison further elevating performance to 85.4%. Similar trends in the MBPP and QuixBugs datasets, highlighting the profound influence of prompt strategies on model performance. We attribute the superior effectiveness of these two prompting methods to their explicit redirection of the model's attention toward "essential functional differences between requirements and code", thus avoiding the overly critical stance commonly triggered by step-by-step explanation and correction prompts. This overly critical stance contradicts the original intent of classical Prompt Engineering and Chain-of-Thought methods [32], [34], which aim to improve reasoning by introducing intermediate reasoning.

V. CONCLUSION

In this paper, we have identified systematic misjudgments by LLMs when verifying code conformance to natural language task specifications. Our results revealed that increasing prompt complexity substantially reduces the models' Requirement Conformance Recognition Rate (RCRR), frequently leading to correct implementations being misclassified as non-conforming. This finding challenges prevailing prompt engineering assumptions, raising a critical limitation due to an "over-correction" bias. We have further proposed two prompt strategies for mitigation, although both strategies improved LLM performance on code requirement verification tasks. While some misjudgments may persisted, we believe our findings offer new and promising directions for developing more reliable and effective automated code review systems.

References

[1] H. Jin, L. Huang, H. Cai, J. Yan, B. Li, and H. Chen, "From llms to llm-based agents for software engineering: A survey of current, challenges and future," 2025. [Online]. Available: https://arxiv.org/abs/2408.02479

[2] J. Austin, A. Odena, M. Nye, M. Bosma, H. Michalewski, D. Dohan, E. Jiang, C. Cai, M. Terry, Q. Le *et al.*, "Program synthesis with large language models," *arXiv preprint arXiv:2108.07732*, 2021.

[3] F. F. Xu, U. Alon, G. Neubig, and V. J. Hellendoorn, "A systematic evaluation of large language models of code," in *Proceedings of the 6th ACM SIGPLAN international symposium on machine programming*, 2022, pp. 1–10.

[4] Z. Rasheed, M. A. Sami, M. Waseem, K.-K. Kemell, X. Wang, A. Nguyen, K. Systä, and P. Abrahamsson, "Ai-powered code review with llms: Early results," *arXiv preprint arXiv:2404.18496*, 2024.

[5] Y. Cai, Z. Hou, D. Sanan, X. Luan, Y. Lin, J. Sun, and J. S. Dong, "Automated program refinement: Guide and verify code large language model with refinement calculus," *Proceedings of the ACM on Programming Languages*, vol. 9, no. POPL, pp. 2057–2089, 2025.

[6] 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," *Advances in Neural Information Processing Systems*, vol. 36, pp. 21 558–21 572, 2023.

[7] S. Shankar, J. Zamfirescu-Pereira, B. Hartmann, A. Parameswaran, and I. Arawjo, "Who validates the validators? aligning llm-assisted evaluation of llm outputs with human preferences," in *Proceedings of the 37th Annual ACM Symposium on User Interface Software and Technology*, 2024, pp. 1–14.

[8] H. Jin, H. Chen, Q. Lu, and L. Zhu, "Towards advancing code generation with large language models: A research roadmap," 2025. [Online]. Available: https://arxiv.org/abs/2501.11354

[9] J. O. Couder, D. Gomez, and O. Ochoa, "Requirements verification through the analysis of source code by large language models," in *SoutheastCon 2024*. IEEE, 2024, pp. 75–80.

[10] L. M. Reinpold, M. Schieseck, L. P. Wagner, F. Gehlhoff, and A. Fay, "Exploring llms for verifying technical system specifications against requirements," *arXiv preprint arXiv:2411.11582*, 2024.

[11] A. Madaan, N. Tandon, P. Gupta, S. Hallinan, L. Gao, S. Wiegreffe, U. Alon, N. Dziri, S. Prabhumoye, Y. Yang *et al.*, "Self-refine: Iterative refinement with self-feedback," *Advances in Neural Information Processing Systems*, vol. 36, pp. 46 534–46 594, 2023.

[12] Z. Ji, N. Lee, R. Frieske, T. Yu, D. Su, Y. Xu, E. Ishii, Y. J. Bang, A. Madotto, and P. Fung, "Survey of hallucination in natural language generation," *ACM Computing Surveys*, vol. 55, no. 12, pp. 1–38, 2023.

[13] J. White, Q. Fu, S. Hays, M. Sandborn, C. Olea, H. Gilbert, A. Elnashar, J. Spencer-Smith, and D. C. Schmidt, "A prompt pattern catalog to enhance prompt engineering with chatgpt," *arXiv preprint arXiv:2302.11382*, 2023.

[14] A. Panickssery, S. Bowman, and S. Feng, "Llm evaluators recognize and favor their own generations," *Advances in Neural Information Processing Systems*, vol. 37, pp. 68 772–68 802, 2024.

[15] M. Chen, J. Tworek, H. Jun, Q. Yuan, H. P. D. O. Pinto, J. Kaplan, H. Edwards, Y. Burda, N. Joseph, G. Brockman *et al.*, "Evaluating large language models trained on code," *arXiv preprint arXiv:2107.03374*, 2021.

[16] J. Austin, A. Odena, M. Nye, M. Bosma, H. Michalewski, D. Dohan, E. Jiang, C. Cai, M. Terry, Q. Le, and C. Sutton, "Program synthesis with large language models," 2021. [Online]. Available: https://arxiv.org/abs/2108.07732

[17] H. Ye, M. Martinez, T. Durieux, and M. Monperrus, "A comprehensive study of automatic program repair on the quixbugs benchmark," *Journal of Systems and Software*, vol. 171, p. 110825, 2021.

[18] Z. Rasheed, M. A. Sami, M. Waseem, K.-K. Kemell, X. Wang, A. Nguyen, K. Systä, and P. Abrahamsson, "Ai-powered code review with llms: Early results," 2024. [Online]. Available: https://arxiv.org/abs/2404.18496

[19] G. Fragiadakis, C. Diou, G. Kousiouris, and M. Nikolaidou, "Evaluating human-ai collaboration: A review and methodological framework," 2025. [Online]. Available: https://arxiv.org/abs/2407.19098

[20] L. Zhu, X. Wang, and X. Wang, "Judgelm: Fine-tuned large language models are scalable judges," 2025. [Online]. Available: https://arxiv.org/abs/2310.17631

[21] J. Wang and Y. Chen, "A review on code generation with llms: Application and evaluation," in *2023 IEEE International Conference on Medical Artificial Intelligence (MedAI)*, 2023, pp. 284–289.

[22] S. Joel, J. J. Wu, and F. H. Fard, "A survey on llm-based code generation for low-resource and domain-specific programming languages," 2024. [Online]. Available: https://arxiv.org/abs/2410.03981

[23] M. Weyssow, A. Kamanda, X. Zhou, and H. Sahraoui, "Codeultrafeedback: An llm-as-a-judge dataset for aligning large language models to coding preferences," 2024. [Online]. Available: https://arxiv.org/abs/2403.09032

[24] J. Wei, X. Wang, D. Schuurmans, M. Bosma, B. Ichter, F. Xia, E. Chi, Q. Le, and D. Zhou, "Chain-of-thought prompting elicits reasoning in large language models," 2023. [Online]. Available: https://arxiv.org/abs/2201.11903

[25] S. Yao, J. Zhao, D. Yu, N. Du, I. Shafran, K. Narasimhan, and Y. Cao, "React: Synergizing reasoning and acting in language models," 2023. [Online]. Available: https://arxiv.org/abs/2210.03629

[26] B. Jin, C. Xie, J. Zhang, K. K. Roy, Y. Zhang, Z. Li, R. Li, X. Tang, S. Wang, Y. Meng, and J. Han, "Graph chain-of-thought: Augmenting large language models by reasoning on graphs," 2024. [Online]. Available: https://arxiv.org/abs/2404.07103

[27] R. Araya, "Do chains-of-thoughts of large language models suffer from hallucinations, cognitive biases, or phobias in bayesian reasoning?" 2025. [Online]. Available: https://arxiv.org/abs/2503.15268

[28] S. A. Akbar, M. M. Hossain, T. Wood, S.-C. Chin, E. M. Salinas, V. Alvarez, and E. Cornejo, "Hallumeasure: Fine-grained hallucination measurement using chain-of-thought reasoning," in *Proceedings of the 2024 Conference on Empirical Methods in Natural Language Processing*, 2024, pp. 15 020–15 037.

[29] W. Tong and T. Zhang, "Codejudge: Evaluating code generation with large language models," *arXiv preprint arXiv:2410.02184*, 2024.

[30] X. Yin, C. Ni, S. Wang, Z. Li, L. Zeng, and X. Yang, "Thinkrepair: Self-directed automated program repair," in *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2024, pp. 1274–1286.

[31] Y. Wang, P. Ji, C. Yang, K. Li, M. Hu, J. Li, and G. Sartoretti, "Mcts-judge: Test-time scaling in llm-as-a-judge for code correctness evaluation," *arXiv preprint arXiv:2502.12468*, 2025.

[32] J. Wei, X. Wang, D. Schuurmans, M. Bosma, F. Xia, E. Chi, Q. V. Le, D. Zhou *et al.*, "Chain-of-thought prompting elicits reasoning in large language models," *Advances in neural information processing systems*, vol. 35, pp. 24 824–24 837, 2022.

[33] J. Moon, Y. Hwang, D. Lee, T. Kang, Y. Kim, and K. Jung, "Don't judge code by its cover: Exploring biases in llm judges for code evaluation," *arXiv preprint arXiv:2505.16222*, 2025.

[34] T. Kojima, S. S. Gu, M. Reid, Y. Matsuo, and Y. Iwasawa, "Large language models are zero-shot reasoners," *Advances in neural information processing systems*, vol. 35, pp. 22 199–22 213, 2022.

[35] B. Meyer, "Applying'design by contract'," *Computer*, vol. 25, no. 10, pp. 40–51, 1992.