# **Towards Advancing Code Generation with Large Language Models: A Research Roadmap**

HAOLIN JIN, University of Sydney, Australia HUAMING CHEN, University of Sydney, Australia QINGHUA LU, CSIRO's Data61, Australia LIMING ZHU, CSIRO's Data61, Australia

Recently, we have witnessed the rapid development of large language models, which have demonstrated excellent capabilities in the downstream task of code generation. However, despite their potential, LLM-based code generation still faces numerous technical and evaluation challenges, particularly when embedded in real-world development. In this paper, we present our vision for current research directions, and provide an in-depth analysis of existing studies on this task. We propose a six-layer vision framework that categorizes code generation process into distinct phases, namely Input Phase, Orchestration Phase, Development Phase, and Validation Phase. Additionally, we outline our vision workflow, which reflects on the currently prevalent frameworks. We systematically analyse the challenges faced by large language models, including those LLM-based agent frameworks, in code generation tasks. With these, we offer various perspectives and actionable recommendations in this area. Our aim is to provide guidelines for improving the reliability, robustness and usability of LLM-based code generation systems. Ultimately, this work seeks to address persistent challenges and to provide practical suggestions for a more pragmatic LLM-based solution for future code generation endeavors.

CCS Concepts: • Software and its engineering;

Additional Key Words and Phrases: Large Language Models, Code Generation, LLM-based Agent

#### **ACM Reference Format:**

#### 1 Introduction

XXX, XXX

Code generation, also known as program synthesis, aims to automatically generate code based on specified inputs [1]. In recent years, Large Language Models (LLMs) emerge as a prominent solution to generate code from natural language descriptions, saving time for developers and serving as benchmarks for evaluation [3, 29]. Evaluation typically involves generating code from descriptions of natural language and validating it against the test sets [1, 11, 26, 28]. Other key tasks include code translation [33, 46] and comprehension [9]. However, the evaluation process

Authors' Contact Information: Haolin Jin, hjin3177@uni.sydney.edu.au, University of Sydney, Sydney, NSW, Australia; Huaming Chen, huaming.chen@sydney.edu.au, University of Sydney, Sydney, NSW, Australia; Qinghua Lu, Qinghua.Lu@data61.csiro.au, CSIRO's Data61, Sydney, NSW, Australia; Liming Zhu, liming.zhu@data61.csiro.au, CSIRO's Data61, Sydney, NSW, Australia.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

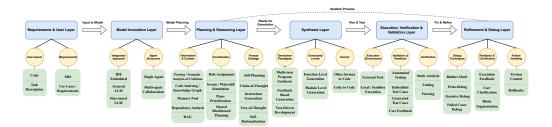


Fig. 1. Proposed Six-Layer Architecture for LLM-Based Code Generation.

may introduce bias since it heavily relies on experienced professionals. In the meantime, specialized models like CodeGen and DeepSeek-Coder have been developed for tasks such as code completion, optimization even debugging and reviews. These models demonstrate the growing potential of LLMs in automating programming tasks.

Despite the wide usage for basic coding, professional programmers prioritize the accuracy and robustness of the generated code. It is identified that inherent shortcomings in LLMs, such as limited contextual understanding [10, 21] and hallucinations [18], may lead to contents with semantic and syntactic errors. To address these challenges, LLM-based agent frameworks are presented for self-refinement and enhancing the model's understanding of tasks and the accuracy of responses with prompt engineering. While they have shown significant improvements over baselines, they still face limitations and challenges across multiple studies.

Previous works explore the potential of LLMs for code generation and introduce benchmarks such as HumanEval [3] and MBPP [1]. The reasoning capabilities of LLMs have also inspired various survey works, highlighting their potential to automate mundane coding tasks, allowing programmers to focus on higher-level activities [2, 19, 40]. However, these works place strong emphasis on model performance evaluation, offering limited insights into framework-level implementation and analysis. For instance, recent attention has been given to LLM-based agent frameworks [20, 40], yet the construction and implementation of these frameworks have not been thoroughly examined.

In this paper, we present our vision and reflections on the application of LLMs in code generation, featuring their current advancements while sharing our perspectives on future researches. We also explore the challenges in code generation from both technical and evaluation viewpoints, offering our proposals as potential solutions. Although extensive research has focused on LLMs' downstream tasks, particularly code generation, focusing efforts on fundamental objectives for pragmatic solutions over merely achieving higher benchmarks is crucial. Through this work, we aim to provide the community with a comprehensive overview of the current trends and challenges in LLM-based code generation, fostering deeper understanding and more impactful future research.

#### 2 Vision for LLM-Based Code Generation

LLMs' capabilities have evolved from basic sentiment analysis to creative tasks [5], with significant improvements from early models like GPT-3.5 to the more advanced GPT-4, featuring a larger parameter scale, improved learning schemes, and deeper alignment with human feedback [22, 39]. However, existing works often highlight repetitive architectures and workflows. To investigate this, we propose a six-layer vision framework to distill core components and streamline the foundation for more efficient and practical solutions.

#### 2.1 Conceptual Framework

Layer 1-3 Figure 1 presents a six-layer collective architecture for how LLMs handle code generation task in current works. The first layer captures user input or the task requirements; with common use cases highlighted in green. This flows into the second Model Invocation Layer, which encompasses different modeling approaches. For example, 'Cursor' act as an IDE using LLM APIs for code generation and basic chat functions, while models could be further fine-tuned specifically for code generation. Following that, the agent or model employs prompt engineering for planning and self-reasoning. We observe most LLM-based code generation studies include this step, frequently employing role assignment and task decomposition, as seen in MetaGPT, ChatDev's Chat Chain [13, 32]. Context Understanding involves techniques like code parsing and semantic analysis to build an internal representation of the project. Methods such as knowledge graphs and dependency analysis help map relationships among files, libraries, and APIs. Retrieval-Augmented Generation (RAG) [23] further enhance the model's knowledge base with external documentation and repositories.

Layer 4-5 The fourth layer, the synthesis layer, involves code generation after planning and reasoning. Similar techniques are included in many agent frameworks to increase code accuracy, such as feeding feedback into the model's analysis to reduce computational overhead and ensure adherence to guidelines [4]. Additionally, multi-turn program synthesis, as in CodeGen [29], is another typical paradigm, with formats for synthesis including natural language-to-code and code-to-code translation. The fifth layer involves running the generated code. Many frameworks utilize external tools [45], while some studies advocate manual validation [43]. Validation can be categorized into two groups: using built-in test sets from benchmark itself, or test sets generated by LLMs [14].

Layer 6 The final layer, Refinement & Debug, includes three key modules. A common method is iterative debugging, where the model performs self-refinement based on prior runtime results, often focusing on erroneous test cases for separate analysis [4]. Another feedback mechanism is user clarification, addressing the ambiguous task descriptions by requesting user input for clearer instructions [27]. As for failure handling, which is not yet widely used, it typically involves storing code snapshots in a registry [12]. If errors occur, the system can revert to a previous version, a workflow reminiscent of daily programming practices. If the result remains unsatisfactory after this layer, a common approach is to return to the planning and reasoning phase and repeat a similar process until the code passes tests or the maximum iterations limit is reached.

## 2.2 Our Vision

In Section 2.1, we discuss our perspective on how LLMs approach code generation, providing a detailed analysis of the overall architecture. Figure 2 illustrates our proposed vision workflow, structured into four phases: Input, Orchestration, Development, and Validation. Unlike the six-layer architecture, our vision does not aim to construct a concrete framework. Instead, it synthesizes insights from existing works to highlight how LLMs have fundamentally transformed software development, providing a comprehensive workflow that serves as a clear roadmap for future research.

*Input Phase* Currently, LLMs in code generation predominantly follow a 'prompt in, code out' approach. Our vision expands this with multi-modal inputs, such as flowcharts as supplementary diagrams, to better reflect real-world software development and reduce the communication overhead [17]. However, LLMs often generate erroneous or non-functional code due to lack of clear information [11, 30, 42]. Discrepancies in task descriptions can severely hinder the model's comprehension, resulting in outputs that fail to meet user requirements. To address this, our

<sup>1</sup>https://www.cursor.com/

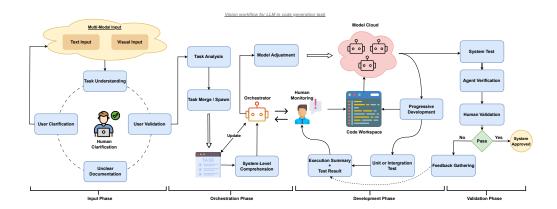


Fig. 2. Our vision workflow for LLM-Based Code Generation.

vision incorporates a <u>clarity check</u> process. Our approach advocates for 'frequent human-model interaction', allowing users to verify LLMs' understanding of tasks and clarify ambiguities. Similar to ClarifyGPT [27], the goal is to reduce hallucinations or speculative outputs from LLMs, and improve LLMs reliability for code generation.

Orchestration Phase The concept of 'orchestration', originally from cloud computing [36], is now widely used in multi-LLM agent frameworks for collaborative operations [40]. Our vision introduces Dynamic Task Creation, enabling on-demand task generation and iterative refinement during development, leveraging execution results and human feedback. During the orchestration phase, the orchestrator LLM performs system-level comprehension of the current task list to dynamically adjust the agents number based on task complexity. This step enables the agent to achieve a clearer understanding of the overarching task and workflow. Once the orchestrator attains a comprehensive understanding of the development tasks, it executes model adjustment to generate multiple agents to support subsequent development processes. These agents are then stored in a model cloud for further utilization.

**Development Phase** Traditional LLM-based code generation often focuses on module or function level outputs, and simulate workflows like Scrum or Test-Driven Development (TDD) [24]. However, for multi-agent frameworks, it can incur heavy overhead due to the crowd collaboration of agents. Also it lacks transparency since their intermediate processes are typically opaque (blackbox), increasing the risk of introducing errors or security flaws [32]. Moreover, these methods lack the explainability that many developers and stakeholders require, as it can be difficult to trace how decisions are made or how code components evolve.

Our vision workflow promotes frequent human interaction over autonomous generation. Since the ultimate goal of any software product is to serve human needs, consistent and deliberate user scrutiny remains indispensable. In our workflow, users (or development team) actively monitor the LLM's incremental coding process—testing and reviewing each module or function upon completion. This approach mirrors real-world practices, such as front-end developers visually inspecting features as they are written. Ongoing workspace monitoring enables users to collaborate with the orchestrator to address errors or unexpected outcomes, refining tasks and agents mid-development. The developer-in-the-loop paradigm combines human oversight with LLM capabilities, reducing black-box risks, enhancing explainability, and ensuring robust and efficient code generation.

Validation Phase In our vision, this phase emphasizes system—level testing and human validation, as debugging and iterative fixes occur during the Development Phase. Once the code reaches a stable state, the orchestrator conducts system tests, verifying all components, including newly generated modules and integrated code, interoperate as intended under real or simulated production conditions. Successful system tests lead to human validation, where developers and users evaluate functionality, usability, and alignment with project goals. This step ensures that the product meets real-world expectations—covering edge cases, performance targets, or business requirements that might fall outside automated checks. Should any discrepancies arise, the workflow routes back to the Development Phase, allowing the team to integrate the system test results and human feedback into further refinements or expansions of the code. The orchestrator then reassigns tasks based on these new insights, enabling a cyclical process of improvement.

## 3 Challenges and Reflections

In this section, we primarily explore common challenges prevalent across different current studies. Additionally, we provide suggestions and insights on how to mitigate or avoid such challenges.

## 3.1 Technical Challenges

3.1.1 Prompt Sensitivity. The advent of prompt engineering [38] has revealed greater potential for LLMs, enabling them to produce outputs better aligned with user needs. However, variations in prompt expression can yield significantly different responses from LLMs [30]. This makes code generation particularly unpredictable, especially for more complex tasks. It may be further exacerbated by lengthy prompts, increasing the risk of syntax and structural inconsistencies, commonly referred to as non-determinism. While reducing temperature can mitigate randomness, it can't fully resolve the issue, as no definitive solution currently exists [30].

Why this is a challenge? Research on code generation for both standalone LLMs and LLM-based agents often overlooks reproducibility in actual environment. Benchmarks such as HumanEval, MBPP, and APPS [11] involve short, self-contained Python tasks, requiring only simple prompt strategies such as chain-of-thought [37] or few-shot learning [35]. However, iterative debugging with a maximum iteration limit often encourages researchers to select the best result from multiple attempts, introducing randomness and making replication difficult. Such tendencies result in the ongoing difficulty in achieving reproducibility across studies.

**Suggestion** To reduce discrepancies, task description should be specific with added constraints or guidelines to stabilize outputs. In frameworks where LLM-based agents diagnose errors through self-reasoning, the prompt must also be carefully constrained, with few-shot learning to further refine outputs. Additionally, incorporating clarity check from our vision workflow can improve task clarity and reduce hallucinations caused by vague or incomplete prompts.

3.1.2 Usability & Consumption. Usability spans the developer's experience, real-world implementation scenarios, and associated costs. Whether in Q&A formats (an LLM translating between programming languages) or IDE plugin, LLM-based tools permeate daily programming tasks. However, researches show developers using such plugins do not necessarily see large efficiency gains compared to those who do not [41]. This raises a critical question: How can we genuinely enhance the usability of these approaches?

Why this is a challenge? Recent trends show a research shift from pure LLM models to LLM-based agents [20], focusing on applied, macro-level exploration rather than raw model performance. Multi-agent frameworks have recently emerged, harnessing multiple agents for increased flexibility and accuracy [40]. While these approaches score impressively on various benchmarks, they often require extensive agent-to-agent interaction, which is time-consuming and struggle to adapt to

shifting requirement midstream. Additionally, the 'black-box' nature hides internal reasoning and role iteration, reducing trust and comprehension for developers.

Meanwhile, as these frameworks scale, developers may invest more effort orchestrating LLM workflow (crafting prompts, clarifying instructions, building test harnesses) as they would coding manually. Inaccurate output further delay the task with manual debugging. A related concern is whether LLMs proactively request missing information when faced with ambiguities, rather than guessing. Recent community discussions<sup>2</sup> emphasize effective agent–human collaboration requires models to recognize knowledge gaps and ask right questions at the right time. Unfortunately, most multi-agent systems still lack robust mechanisms to detect or flag such uncertainties, forcing developers to retroactively diagnose issues.

Suggestion Studies indicate LLMs perform worse on class-level generation compared to function-level tasks [7]. We argue that future research should prioritize practical usage cases, such as deeper IDE integration and real-world evaluations, rather than only assessing one-off code completions. Token consumption also increases with multi-agent collaborations, as model calls and shared context grows. Effective retrieval-augmented generation and precise content extraction are vital to expand the model's knowledge base without excessive overhead. Moreover, frameworks should bolster the model's ability to detect inaccurate or missing information, promoting clarification or offering alternatives. Developers often avoid LLM-based code generation due to the time spent aligning with and debugging outputs, which negates any efficiency gains. If the model assumes certain functions or libraries exist, usability declines [19, 32]. Our proposed vision workflow (Subsection 2.2) addresses these issues by focusing on project-level tasks and adopting a dynamic model cloud. Frequent developer-orchestrator interactions help reduce token cost while improving explainability and code comprehension by revealing the model's assumptions in real time.

3.1.3 Code Security. Research on the security of LLM-based code generation has received little attention. Most works focus on bias or jailbreak vulnerabilities in model conversations [6, 8], especially for closed-source LLMs, but seldom addresses security concern in code-generation contexts. However, this does not imply that generated code is fully trustworthy or free from hidden security risks in text-based outputs.

Why this is a challenge? In fact, LLM-based code generation can inadvertently introduce security risks by generating unsafe logic or code with latent vulnerabilities [31]. It arises not from malicious intent but from the model unintentionally producing insecure code. Furthermore, LLMs often rely on external knowledge sources that may be compromised, exposing outputs to risks like phishing if references (for example, a pip install link) are tampered with [15]. The code itself typically does not undergo robust testing; using unverified code (lacking integration tests) in a real development environment that handles sensitive data poses significant risks. Moreover, with LLMs now autonomously calling external APIs and utilizing extensive training data, novel attack surfaces are introduced where backdoor attacks can be placed [44]. Multi-agent frameworks further expand the attack surface via user prompts or retrieval data, while current security measures focus on final output, leaving intermediate workflows vulnerable.

**Suggestion** We suggest future research prioritize code security, together with code functional correctness. A module-level approach can be developed to robust vulnerabilities testing. Furthermore, research has indicated that model-generated code can exhibit biases [16], highlighting the importance of addressing such issues during the training process.

<sup>&</sup>lt;sup>2</sup>https://www.latent.space/p/2024-agents

## 3.2 Evaluation Challenges

Evaluation is one most crucial step in code generation, which ensures that the generated code contains no syntactic or semantic errors and actually fulfills the requirements described in the task.

Why this is a challenge? Currently, there are few widely-used benchmarks for code generation. Examples include HumanEval+ from EvalPlus [25] and CodeContests, both of which come with test sets ranging from basic function problems to competition level tasks. However, these benchmarks reveal a major limitation: most focus on function-level or single-file tasks, which do not reflect the complexity of real-world software development. In addition, the majority of these benchmarks lack sufficient edge cases, making it difficult to catch hidden bugs or vulnerabilities [25]. This explains why some studies, despite attaining high pass rates on MBPP or HumanEval, stuggle with more complex problems.

Moreover, evaluation metrics themselves present another challenge. Many existing works rely on simple metrics like Pass@K [46], which does not account for efficiency, performance, or maintainability. Consequently, the model may show partial correctness but fail in more realistic settings. In real-world projects, test sets are often too small or nonexistent—some specialized scenarios (e.g., those akin to Alfworld [34]) remain untested in code generation tasks, further reducing the scope of usability evaluation.

**Suggestion** We argue that future research should aim to create more class-level or multifile benchmarks that better simulate real-world project development. These new tasks should include more complex dependencies and cross-module interactions, enabling us to test how well LLMs handle multi-step integrations instead of trivial function stubs. Existing test sets could be augmented with additional extreme conditions to increase robustness, as demonstrated by EvalPlus [25]. Beyond simple pass/fail tests, we should also measure the readability and maintainability of the generated code—possibly using agent frameworks to integrate external tools for deeper evaluations. Finally, domain-specific benchmarks (e.g., for financial software) would allow us to reflect real-world demands more accurately.

#### 4 Conclusion

In this paper, we present our vision and reflection for applying large language models in code generation task, examing the technical and evaluation challenges that persist in current research works. We introduce a six-layer collective architecture for how current LLMs researches address code generation task, as well as our vision workflow. Ultimately, we anticipate this work provides a comprehensive landscape of current LLM-based code generation approaches, highlights ongoing critical challenges concerning both technical and evaluation perspectives, and offers practical guidelines for future solutions.

#### References

- [1] Jacob Austin, Augustus Odena, Maxwell Nye, Maarten Bosma, Henryk Michalewski, David Dohan, Ellen Jiang, Carrie Cai, Michael Terry, Quoc Le, et al. 2021. Program synthesis with large language models. arXiv preprint arXiv:2108.07732 (2021).
- [2] Yupeng Chang, Xu Wang, Jindong Wang, Yuan Wu, Linyi Yang, Kaijie Zhu, Hao Chen, Xiaoyuan Yi, Cunxiang Wang, Yidong Wang, et al. 2024. A survey on evaluation of large language models. ACM Transactions on Intelligent Systems and Technology 15, 3 (2024), 1–45.
- [3] Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde De Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, et al. 2021. Evaluating large language models trained on code. *arXiv* preprint arXiv:2107.03374 (2021).
- [4] Xinyun Chen, Maxwell Lin, Nathanael Schärli, and Denny Zhou. 2023. Teaching large language models to self-debug. arXiv preprint arXiv:2304.05128 (2023).
- [5] Zenan Chen and Jason Chan. 2024. Large language model in creative work: The role of collaboration modality and user expertise. *Management Science* 70, 12 (2024), 9101–9117.
- [6] Yue Deng, Wenxuan Zhang, Sinno Jialin Pan, and Lidong Bing. 2023. Multilingual jailbreak challenges in large language models. arXiv preprint arXiv:2310.06474 (2023).
- [7] Xueying Du, Mingwei Liu, Kaixin Wang, Hanlin Wang, Junwei Liu, Yixuan Chen, Jiayi Feng, Chaofeng Sha, Xin Peng, and Yiling Lou. 2024. Evaluating large language models in class-level code generation. In Proceedings of the IEEE/ACM 46th International Conference on Software Engineering. 1–13.
- [8] Yucong Duan. 2024. The Large Language Model (LLM) Bias Evaluation (Age Bias). DIKWP Research Group International Standard Evaluation. DOI 10 (2024).
- [9] Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, et al. 2020. Codebert: A pre-trained model for programming and natural languages. arXiv preprint arXiv:2002.08155 (2020).
- [10] Muhammad Usman Hadi, Rizwan Qureshi, Abbas Shah, Muhammad Irfan, Anas Zafar, Muhammad Bilal Shaikh, Naveed Akhtar, Jia Wu, Seyedali Mirjalili, et al. 2023. A survey on large language models: Applications, challenges, limitations, and practical usage. Authorea Preprints (2023).
- [11] Dan Hendrycks, Steven Basart, Saurav Kadavath, Mantas Mazeika, Akul Arora, Ethan Guo, Collin Burns, Samir Puranik, Horace He, Dawn Song, et al. 2021. Measuring coding challenge competence with apps. arXiv preprint arXiv:2105.09938 (2021).
- [12] Samuel Holt, Max Ruiz Luyten, and Mihaela van der Schaar. 2023. L2mac: Large language model automatic computer for unbounded code generation. arXiv preprint arXiv:2310.02003 (2023).
- [13] Sirui Hong, Xiawu Zheng, Jonathan Chen, Yuheng Cheng, Jinlin Wang, Ceyao Zhang, Zili Wang, Steven Ka Shing Yau, Zijuan Lin, Liyang Zhou, et al. 2023. Metagpt: Meta programming for multi-agent collaborative framework. *arXiv* preprint arXiv:2308.00352 (2023).
- [14] Dong Huang, Qingwen Bu, and Heming Cui. 2023. Codecot and beyond: Learning to program and test like a developer. arXiv preprint arXiv:2308.08784 (2023).
- [15] Dong Huang, Qingwen Bu, Jie Zhang, Xiaofei Xie, Junjie Chen, and Heming Cui. 2023. Bias assessment and mitigation in llm-based code generation. arXiv preprint arXiv:2309.14345 (2023).
- [16] Dong Huang, Qingwen Bu, Jie Zhang, Xiaofei Xie, Junjie Chen, and Heming Cui. 2024. Bias Testing and Mitigation in LLM-based Code Generation. arXiv:2309.14345 [cs.SE] https://arxiv.org/abs/2309.14345
- [17] Hanyao Huang, Ou Zheng, Dongdong Wang, Jiayi Yin, Zijin Wang, Shengxuan Ding, Heng Yin, Chuan Xu, Renjie Yang, Qian Zheng, et al. 2023. ChatGPT for shaping the future of dentistry: the potential of multi-modal large language model. *International Journal of Oral Science* 15, 1 (2023), 29.
- [18] Ziwei Ji, Nayeon Lee, Rita Frieske, Tiezheng Yu, Dan Su, Yan Xu, Etsuko Ishii, Ye Jin Bang, Andrea Madotto, and Pascale Fung. 2023. Survey of hallucination in natural language generation. *Comput. Surveys* 55, 12 (2023), 1–38.
- [19] Juyong Jiang, Fan Wang, Jiasi Shen, Sungju Kim, and Sunghun Kim. 2024. A Survey on Large Language Models for Code Generation. arXiv preprint arXiv:2406.00515 (2024).
- [20] Haolin Jin, Linghan Huang, Haipeng Cai, Jun Yan, Bo Li, and Huaming Chen. 2024. From llms to llm-based agents for software engineering: A survey of current, challenges and future. arXiv preprint arXiv:2408.02479 (2024).
- [21] Jean Kaddour, Joshua Harris, Maximilian Mozes, Herbie Bradley, Roberta Raileanu, and Robert McHardy. 2023. Challenges and applications of large language models. arXiv preprint arXiv:2307.10169 (2023).
- [22] Katikapalli Subramanyam Kalyan. 2024. A survey of GPT-3 family large language models including ChatGPT and GPT-4. *Natural Language Processing Journal* 6 (2024), 100048.
- [23] Patrick Lewis, Ethan Perez, Aleksandra Piktus, Fabio Petroni, Vladimir Karpukhin, Naman Goyal, Heinrich Küttler, Mike Lewis, Wen-tau Yih, Tim Rocktäschel, et al. 2020. Retrieval-augmented generation for knowledge-intensive nlp tasks. Advances in Neural Information Processing Systems 33 (2020), 9459–9474.

- [24] Feng Lin, Dong Jae Kim, and TH Chen. 2024. SOEN-101: Code Generation by Emulating Software Process Models Using Large Language Model Agents. arXiv preprint arXiv:2403.15852 (2024).
- [25] Jiawei Liu, Chunqiu Steven Xia, Yuyao Wang, and Lingming Zhang. 2024. Is your code generated by chatgpt really correct? rigorous evaluation of large language models for code generation. Advances in Neural Information Processing Systems 36 (2024).
- [26] Shuai Lu, Daya Guo, Shuo Ren, Junjie Huang, Alexey Svyatkovskiy, Ambrosio Blanco, Colin Clement, Dawn Drain, Daxin Jiang, Duyu Tang, et al. 2021. Codexglue: A machine learning benchmark dataset for code understanding and generation. arXiv preprint arXiv:2102.04664 (2021).
- [27] Fangwen Mu, Lin Shi, Song Wang, Zhuohao Yu, Binquan Zhang, Chenxue Wang, Shichao Liu, and Qing Wang. 2023. ClarifyGPT: Empowering LLM-based Code Generation with Intention Clarification. arXiv preprint arXiv:2310.10996 (2023)
- [28] Ansong Ni, Pengcheng Yin, Yilun Zhao, Martin Riddell, Troy Feng, Rui Shen, Stephen Yin, Ye Liu, Semih Yavuz, Caiming Xiong, et al. 2024. L2ceval: Evaluating language-to-code generation capabilities of large language models. Transactions of the Association for Computational Linguistics 12 (2024), 1311–1329.
- [29] Erik Nijkamp, Bo Pang, Hiroaki Hayashi, Lifu Tu, Huan Wang, Yingbo Zhou, Silvio Savarese, and Caiming Xiong. 2022. Codegen: An open large language model for code with multi-turn program synthesis. arXiv preprint arXiv:2203.13474 (2022).
- [30] Shuyin Ouyang, Jie M Zhang, Mark Harman, and Meng Wang. 2024. An empirical study of the non-determinism of chatgpt in code generation. ACM Transactions on Software Engineering and Methodology (2024).
- [31] Hammond Pearce, Baleegh Ahmad, Benjamin Tan, Brendan Dolan-Gavitt, and Ramesh Karri. 2022. Asleep at the Keyboard? Assessing the Security of GitHub Copilot's Code Contributions. In 2022 IEEE Symposium on Security and Privacy (SP). 754–768. doi:10.1109/SP46214.2022.9833571
- [32] Chen Qian, Wei Liu, Hongzhang Liu, Nuo Chen, Yufan Dang, Jiahao Li, Cheng Yang, Weize Chen, Yusheng Su, Xin Cong, et al. 2024. Chatdev: Communicative agents for software development. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*. 15174–15186.
- [33] Sebastian Ruder, Anders Søgaard, and Ivan Vulić. 2019. Unsupervised cross-lingual representation learning. In Proceedings of the 57th Annual Meeting of the Association for Computational Linguistics: Tutorial Abstracts. 31–38.
- [34] Mohit Shridhar, Xingdi Yuan, Marc-Alexandre Côté, Yonatan Bisk, Adam Trischler, and Matthew Hausknecht. 2021.
  ALFWorld: Aligning Text and Embodied Environments for Interactive Learning. arXiv:2010.03768 [cs.CL] https://arxiv.org/abs/2010.03768
- [35] Yaqing Wang, Quanming Yao, James T Kwok, and Lionel M Ni. 2020. Generalizing from a few examples: A survey on few-shot learning. ACM computing surveys (csur) 53, 3 (2020), 1–34.
- [36] Denis Weerasiri, Moshe Chai Barukh, Boualem Benatallah, Quan Z Sheng, and Rajiv Ranjan. 2017. A taxonomy and survey of cloud resource orchestration techniques. *ACM Computing Surveys (CSUR)* 50, 2 (2017), 1–41.
- [37] Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Fei Xia, Ed Chi, Quoc V Le, Denny Zhou, et al. 2022. Chain-of-thought prompting elicits reasoning in large language models. Advances in neural information processing systems 35 (2022), 24824–24837.
- [38] Jules White, Quchen Fu, Sam Hays, Michael Sandborn, Carlos Olea, Henry Gilbert, Ashraf Elnashar, Jesse Spencer-Smith, and Douglas C Schmidt. 2023. A prompt pattern catalog to enhance prompt engineering with chatgpt. arXiv preprint arXiv:2302.11382 (2023).
- [39] Shengqiong Wu, Hao Fei, Leigang Qu, Wei Ji, and Tat-Seng Chua. 2023. Next-gpt: Any-to-any multimodal llm. arXiv preprint arXiv:2309.05519 (2023).
- [40] Zhiheng Xi, Wenxiang Chen, Xin Guo, Wei He, Yiwen Ding, Boyang Hong, Ming Zhang, Junzhe Wang, Senjie Jin, Enyu Zhou, et al. 2023. The rise and potential of large language model based agents: A survey. arXiv preprint arXiv:2309.07864 (2023).
- [41] Frank F. Xu, Bogdan Vasilescu, and Graham Neubig. 2021. In-IDE Code Generation from Natural Language: Promise and Challenges. arXiv:2101.11149 [cs.SE] https://arxiv.org/abs/2101.11149
- [42] Zhou Yang, Zhensu Sun, Terry Zhuo Yue, Premkumar Devanbu, and David Lo. 2024. Robustness, Security, Privacy, Explainability, Efficiency, and Usability of Large Language Models for Code. arXiv:2403.07506 [cs.SE] https://arxiv. org/abs/2403.07506
- [43] Kechi Zhang, Zhuo Li, Jia Li, Ge Li, and Zhi Jin. 2023. Self-edit: Fault-aware code editor for code generation. arXiv preprint arXiv:2305.04087 (2023).
- [44] Quan Zhang, Binqi Zeng, Chijin Zhou, Gwihwan Go, Heyuan Shi, and Yu Jiang. 2024. Human-imperceptible retrieval poisoning attacks in LLM-powered applications. In Companion Proceedings of the 32nd ACM International Conference on the Foundations of Software Engineering. 502–506.
- [45] Simiao Zhang, Jiaping Wang, Guoliang Dong, Jun Sun, Yueling Zhang, and Geguang Pu. 2024. Experimenting a New Programming Practice with LLMs. arXiv preprint arXiv:2401.01062 (2024).

|        |      |         | ~ .   |         |     |            |     |
|--------|------|---------|-------|---------|-----|------------|-----|
| Haolin | lin. | Huaming | Chen. | Oinghua | Lu. | and Liming | Zhu |