ChemGen: Towards Understanding First-Principles Calculation Code Generation Based on Large Language Models

PENG GAO, University of Science and Technology of China, China
FENG QIU, University of Science and Technology of China, China
BAOJIAN HUA*, University of Science and Technology of China, China

First-principles calculation software, grounded in quantum chemistry theories, is indispensable in scientific research. However, the development of such software requires the amalgamation of multidisciplinary knowledge, posing a significant challenge to developers. We propose an approach to utilize large language models (LLMs) for automatically generating code for first-principles calculations. Building on this concept, we have designed and implemented ChemGen, a fully automated framework to assist in generating and evaluating code for first-principles calculations. Meanwhile, we have developed a benchmark named ChemEval, which includes 24 code generation tasks tailored for first-principles calculations. Our experiments, conducted using three leading LLMs—GPT-3.5 Turbo, Gemini Pro, and WizardCoder-Python-13B—indicate that these models can generate functionally correct code for 79.17% of the tasks in ChemEval. Additionally, for each of the LLMs used, the median cyclomatic complexity of the generated code did not exceed 3. Furthermore, the application of the knowledge generation prompting technique improves the accuracy of the produced code.

CCS Concepts: • **Software and its engineering** → *Software development techniques.*

Additional Key Words and Phrases: First-principles calculation software, Large language models, Code generation

ACM Reference Format:
Peng Gao, Feng Qiu, and Baojian Hua. 2024. ChemGen: Towards Understanding First-Principles Calculation Code Generation Based on Large Language Models. 1, 1 (March 2024), 12 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

In today's era of rapid information turnover and continuous technological innovation, computational and simulation methods have become the core drivers of scientific research and engineering innovation. Particularly, first-principles calculation software based on the fundamental principles of quantum mechanics, such as Gaussian [6] and VASP [9] play a crucial role in key areas like drug development and material design.

However, developing first-principles calculation software is a challenging task. Initially, developers must master specialized programming knowledge in performance optimization and parallel

*Corresponding author.

Authors' addresses: Peng Gao, School of Software Engineering, University of Science and Technology of China, Suzhou, 215123, China, gaopeng0311@mail.ustc.edu.cn; Feng Qiu, School of Software Engineering, University of Science and Technology of China, Suzhou, 215123, China, bgkqf@mail.ustc.edu.cn; Baojian Hua, School of Software Engineering, Suzhou Institute for Advanced Research, University of Science and Technology of China, Suzhou, 215123, China, bjhua@ustc.edu.cn.

XXXX-XXXX/2024/3-ART $15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

computing; furthermore, they need to possess interdisciplinary expertise in linear algebra, calculus, and quantum chemistry. Lastly, manually writing code is not only inefficient but also prone to errors.

Recognizing these challenges, we propose the use of large language models (LLMs) for the generation of first-principles calculation codes. LLMs, based on the Transformer architecture [20], are pretrained on massive datasets, enabling them to understand the rules, grammar, and semantics of natural language, thus granting them the ability to generate fluent and natural text [17]. Many studies have shown that LLMs possess strong code generation capabilities [2, 4, 14, 21]. However, there has not been sufficient research on the effectiveness of first-principles calculation code generated by LLMs.

Our Work. In this paper, we introduce ChemGen, a framework designed for the generation and evaluation of first-principles calculation codes through the use of LLMs. ChemGen is divided into two main components: the generation module and the evaluation module. The generation module is crafted to employ various prompting strategies, facilitating the invocation of LLMs to produce codes that adhere to specified requirements. Meanwhile, the evaluation module is tasked with assessing the correctness and complexity of the generated codes, utilizing metrics such as pass@k [2] and cyclomatic complexity [13], to provide an evaluation results statistical report.

To investigate the effectiveness of first-principles calculation codes generated by LLMs, we first established a benchmark named ChemEval, comprising 24 common tasks for developing first-principles calculation code, each paired with a corresponding test suite. We then conducted code generation and evaluation experiments with three advanced LLMs: GPT-3.5 Turbo [15], Gemini Pro [19], and WizardCoder-Python-13B [12], utilizing various parameter settings and prompting strategies. Finally, we performed a detailed analysis of the correctness and complexity of the generated codes.

Our findings indicate that LLMs are capable of accurately addressing a majority of first-principles calculation code generation tasks in ChemEval. Notably, GPT-3.5 Turbo model distinguished itself by achieving a remarkable pass@20 accuracy rate of 79.17%. Additionally, the complexity of codes generated by all evaluated LLMs was generally low. Moreover, our study highlights the varying effectiveness of different prompting strategies: while using code samples as prompts did not yield improvements in code generation accuracy, the adoption of knowledge generation prompting techniques significantly boosted the accuracy of the codes produced by LLMs.

Contributions. To the best of our knowledge, this work represents the first detailed evaluation of first-principles calculation code generated by LLMs, providing quantified results. Key contributions include:

(1) The comprehensive study on LLM-generated first-principles calculation codes, introducing ChemGen, a supporting software prototype.
(2) Detailed evaluation and insights from analysis, with a bespoke benchmark for code generation and testing.

2 BACKGROUND

To be self-contained, we present, in this section, necessary background information on first-principles calculation software and LLMs.

2.1 First-Principles Calculation Software

First-principles calculation software is a tool that utilizes the fundamental laws of quantum mechanics to compute the electronic structure and properties of materials or molecules from scratch. These software packages do not rely on any empirical parameters or semi-empirical methods;

instead, they are directly based on the basic interactions of atoms and electrons constituting the material. Hence, they are referred to as "first-principles" or "ab initio" methods.

These computational methods are primarily based on the Schrödinger equation, and common theoretical frameworks include density functional theory [7] and wave function-based methods. The key to developing such software lies in translating the theoretical equations of quantum mechanics into numerical models and accurately simulating electronic behavior and material properties through computer code.

2.2 LLMs

Introduction. LLMs are based on deep learning techniques, utilizing the Transformer architecture as their foundation. This architecture's self-attention mechanism enables the models to capture semantic connections between various parts of the input text. Trained on extensive datasets, LLMs with hundreds of millions to trillions of parameters, possess the capability to effectively comprehend and generate text.

Prompt engineering. Prompts are the initial texts or instructions provided to a model, aimed at guiding the model to complete predetermined tasks. Prompt engineering is a method to enhance model performance through the careful design and adjustment of these prompts.

Sampling strategies. In text generation, models create a probability distribution for each potential word or token, predicting its likelihood of being selected. Sampling strategies guide the selection of words or sequences from this distribution, aiming to generate coherent and reasonable text. Temperature regulation adjusts the selection process by altering the sharpness or flatness of the probability distribution, while Top-p sampling selects the next word based on the smallest set of words whose cumulative probability exceeds a specific threshold p [8]

3 DESIGN

In this section, we introduce the overall architecture of ChemGen, followed by a description of the two core modules of ChemGen.

3.1 Architecture

We present the architecture of the ChemGen in Figure 1, which consists of two core modules: the generation module and the evaluation module. The generation module is responsible for invoking LLMs to generate first-principles calculation code based on function descriptions (including function signature and comment) and prompts. The evaluation module utilizes the test suites within the benchmark and evaluates generated codes by conducting correctness tests and complexity assessments. It then aggregates the results and generates statistical reports.

3.2 Generation Module

In this section, we present the process to use LLMs to generate code and the prompting strategies we employed.

3.2.1 Code generation process. The code generation module takes the function descriptions from benchmarks as input and outputs the first-principles calculation code generated based on these descriptions. These function descriptions are not directly input into the LLMs; instead, they are combined with prompting strategies before being fed to the LLMs. Subsequently, we extract the generated code from the LLMs' responses. Given the need to generate multiple pieces of code for each task, there is a loop in place until the number of generated codes meets the specified requirements.

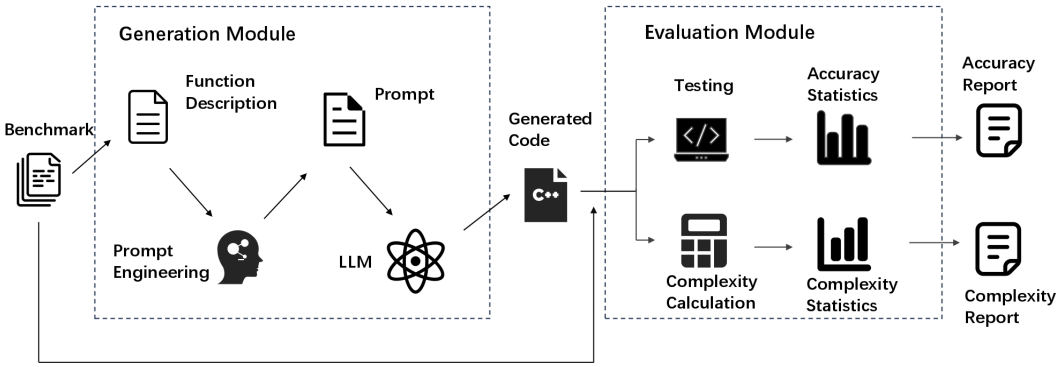

Fig. 1. Overall framework of ChemGen.

3.2.2 Prompting engineering. **Zero-Shot Prompting**. Zero-shot prompting strategy is a method that guides LLMs to complete tasks through direct natural language instructions without relying on examples [10]. Our strategy, designed based on zero-shot prompting, encompasses two core elements: firstly, natural language task instructions to guide LLMs to generate code, and secondly, specific code functionality requirements defined by the ChemEval. We show the template of zero-shot prompting in Figure 2.

As a professional developer specializing in first-principles calculation software, your task is to write the corresponding code based on the code functionality requirements provided. Please read the following guidelines and requirements carefully to ensure your code meets the standards.

Code Implementation Requirements:
1. Functionality: The code must satisfy the given functionality requirements and accurately implement the intended features.
2. Stability: Ensure the code is error-free and executes reliably, handling various edge cases and potential exceptions.
3. Simplicity: The code structure should be simple, clear, and easy to understand. Use appropriate variable and function names and avoid unnecessary complexity.

Response Format:
1. Your response should only include the function body.
2. Please do not include any additional textual information or explanatory comments.

Functionality Requirements:
[Specific function signature and comment]

Fig. 2. Zero-shot prompting template.

Code Sample Prompting. The prompting strategy introduced is dedicated to guiding LLMs to identify and correct errors within code samples, aiming to produce correct code with minimal corrections. The template of this prompt strategy was based on zero-shot prompting template and was enhanced by adding a section for code samples. Additionally, the task description at the beginning of the template was adjusted in accordance with this strategy. Importantly, this approach is considered a zero-shot prompting strategy, as our prompts do not provide examples to demonstrate how to complete the task.

Knowledge Generation Prompting. Knowledge Generation Prompting is a technique aimed at optimizing code generation by LLMs, guiding LLMs to accurately extract and apply the key knowledge necessary for completing tasks [11]. Initially, we specify the code functionality requirements, leading LLMs to identify essential knowledge points, such as relevant mathematical formulas and theoretical foundations. Then, based on these identified knowledge points, we provide LLMs with precise programming instructions, further guiding the model to generate code as required.

3.3 Evaluation Module

This section will detail the evaluation process we employ and the metrics used for assessment.

Correctness Evaluation. The correctness evaluation process involves combining the function signature, generated function body, and test code into a single file for compilation. If the file compiles successfully and runs without errors, runtime issues, or timeouts, yielding an execution result of 0, the code is considered correct. We then perform statistical analysis on the test results of all candidate codes and produce a report.

To quantify the correctness of candidate codes, we adopted the widely recognized pass@k metric. This metric measures the probability of selecting at least one correct code when arbitrarily trying k codes from a given unordered set of candidate codes. Assuming the total number of candidate codes generated for a specific task is n, and the number of codes that have passed testing and are considered correct is c, then the formula for pass@k is as follows:

$$\text{pass@}k := \mathbb{E}_{\text{problems}}\left[1 - \frac{\binom{n-c}{k}}{\binom{n}{k}}\right] \tag{1}$$

Complexity Evaluation. To enhance the efficiency of processing a large number of code samples, we automated the complexity analysis process. In Algorithm 1, we demonstrate our method for calculating the cyclomatic complexity of generated code. We utilize Lizard [22], a tool extensively adopted for measuring the code's cyclomatic complexity. Following this, we aggregate the data, ultimately producing a comprehensive report on cyclomatic complexity. This metric quantifies complexity by calculating the number of control flow paths within the program, aiding us in gaining a deeper understanding of the complexity of the code structure.

4 EVALUATION

In this section, we present the experimental results by answering research questions.

4.1 Research Questions

By presenting the experimental results, we mainly investigate the following research questions:
RQ1: Effectiveness. How is the accuracy of the first-principles calculation codes generated by LLMs?
RQ2: Complexity. What is the complexity of the generated code?
RQ3: Prompt strategy selection. How do prompt techniques affect the accuracy of the generated code?

4.2 Experimental Setup

Benchmark. We meticulously selected 24 representative first-principles calculation code generation tasks and personally designed and implemented their function descriptions, reference codes, and test cases in C++ language. These tasks are divided into three main categories, as detailed in Table 1.

Algorithm 1: The Algorithm for Calculating Cyclomatic Complexity

Input : all_code_with_id: A list of items, each item consisting of a generated function body and its associated task ID
tasks: benchmark
Output : list of cyclomatic complexity

```
Function calcCyclomaticComplexity(code_with_id, tasks):
    cclist ← [];
    for item in all_code_with_id do
        function_body ← getFunctionBody(item);
        task_id ← getTaskId(item);
        function_signature ← getFunctionSignature(tasks, task_id);
        code ← combine(function_signature, function_body);
        info ← lizardAnalyzeSourceCode(code);
        cc ← getCyclomaticComplexity(info);
        cclist ← append(cclist, cc);
    end
    return cclist;
```

Table 1. Categories of tasks in ChemEval.

Category	Quantity
Wavefunction and Electron Density Processing (WEDP)	10
Physical Properties Analysis (PPA)	5
Energy Calculation and Analysis (ECA)	9

Selected LLMs. We carefully selected three leading LLMs, as detailed in Table 2, for our experiments. These models have demonstrated outstanding performance across a variety of code generation tasks and benchmarks.

Table 2. Selected LLMs for experiments.

Model Name	Modality	Conversation Supported	Size
GPT-3.5 Turbo	Language	Yes	Unknown
Gemini Pro	Language	Yes	Unknown
WizardCoder-Python-13B	Code	No	13B

Environment. All experiments and evaluations were conducted on a server equipped with 10 Intel Xeon Scalable processor cores and 40GB of memory, running the Ubuntu 20.04 operating system. ChemGen mentioned in this paper was implemented using Python 3.10.12, while the compilation of C++ code was carried out using the G++ 11.4.0 compiler.

4.3 RQ1: Effectiveness

We employed zero-shot prompting technique to generate first-principles calculation codes using LLMs under three different sampling parameter configurations. In Figure 3, we detailed these

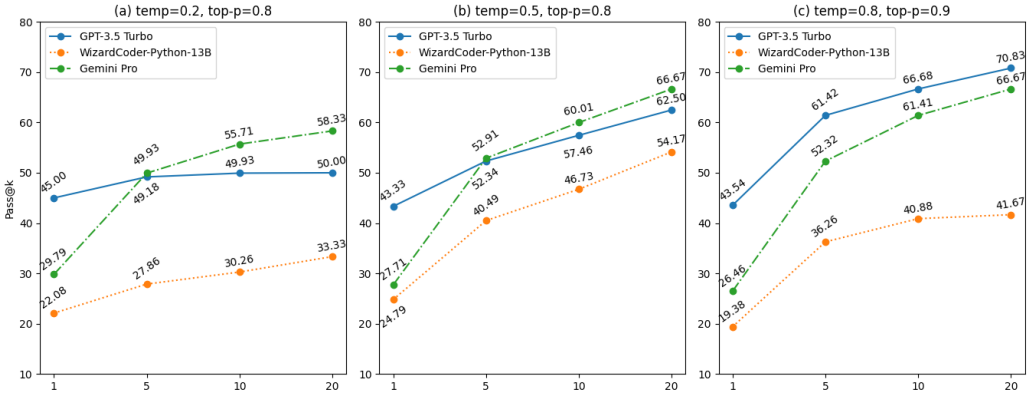

Fig. 3. Pass@k results for zero-shot prompting.

parameter settings and the pass@k scores of the codes generated under each configuration. The results showed that both GPT-3.5 Turbo and Gemini Pro performed excellently across all parameter configurations. At temperature settings of 0.2 and 0.5, Gemini Pro achieved the highest accuracy at pass@5 and above, reaching up to 66.67%. It is worth mentioning that GPT-3.5 Turbo demonstrated outstanding performance in pass@1, with the lowest pass@1 reaching 43.33%, significantly outperforming other models' pass@1 of below 30%. When the temperature was 0.8 and top-p was 0.9, GPT-3.5 Turbo surpassed all other models, achieving the highest pass@20 of 70.83%. WizardCoder-Python-13B showed relatively better performance at a temperature of 0.5, with a pass@20 of 54.17%, but had a significant gap compared to the first two models under other settings.

To deeply understand the characteristics of generated code, we present detailed statistics on key aspects including code duplication rate, compilation success rate, and the proportion of code passing tests in Table 3. Equation 2 outlines our approach to calculating code duplication rates for each model, where uc represents the count of unique code samples after comments and whitespace are removed, and ac is the total amount of code generated, both under specific temperature and top-p settings.

$$\text{Duplicate} = 1 - \frac{uc}{ac} \tag{2}$$

Table 3. Statistics of generated code.

Model Name	Temperature	Top-p	Duplicate	Compile	Correct
GPT-3.5 Turbo	0.2	0.8	65.0%	95.8%	49.2%
	0.5	0.8	36.7%	95.6%	46.5%
	0.8	0.9	21.5%	95.2%	46.0%
Gemini Pro	0.2	0.8	44.0%	73.8%	29.8%
	0.5	0.8	17.1%	75.1%	27.7%
	0.8	0.9	9.7%	76.9%	26.5%
WizardCoder-Python-13B	0.2	0.8	51.7%	90.4%	30.4%
	0.5	0.8	19.0%	89.8%	31.3%
	0.8	0.9	2.7%	85.8%	25.6%

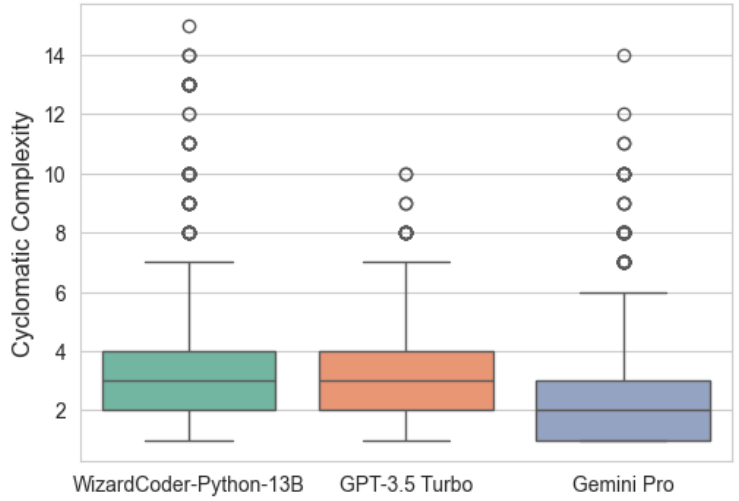

Fig. 4. Cyclomatic complexity results.

Table 3 illustrates a gradual decrease in code duplication rate with increasing temperature parameters and top-p values, aligning with our expectations. This phenomenon can be attributed to the higher temperature and top-p values fostering more diverse outputs from the model. Further analysis revealed that GPT-3.5 Turbo and WizardCoder-Python-13B exhibit better performance in terms of compilation success rate, with Gemini Pro's code having a compilation success rate of approximately 75%. Upon reviewing the codes that failed to compile, the primary issues identified include the use of undefined functions, passing arguments with types or numbers that do not match the function declarations, improper use of operators, and attempts to modify const variables.

Notably, GPT-3.5 Turbo distinguishes itself by achieving a high code pass rate, which explains its exceptionally high scores in the pass@1 metric. Surprisingly, despite Gemini Pro performing relatively well in the pass@k metric, it has the lowest code pass rate. This indicates that while Gemini Pro can provide correct code for a broader range of problems, the number of correct codes it generates is quite limited. In summary, these models are capable of generating code based on first-principles calculations, with pass@1 of at most 19.38% - 45.0% and pass@20 of at most 33.33% - 70.83%. Particularly, GPT-3.5 Turbo exhibits superior performance, achieving the highest levels in pass@k, compilation success rate, and code correctness rate.

4.4 RQ2: Complexity

In this section, we analyze the cyclomatic complexity of codes generated by three models and display their distribution in Figure 4. The observed results indicate that Gemini Pro exhibits lower values in terms of cyclomatic complexity distribution. Specifically, the first quartile is 1, the median is 2, and the third quartile is 3, which are all one unit lower than those of the other two models. This finding highlights Gemini Pro's potential advantage in generating more concise code. And we conducted a two-tailed Wilcoxon rank-sum test to compare the cyclomatic complexity of codes across three distinct models. The results reveal significant statistical differences among the models, underscored by p-values of 5.15e-3, 8.50e-40, and 1.29e-54, respectively.

Figure 4 also shows that the cyclomatic complexity of code generated by each model includes outliers. GPT-3.5 Turbo has fewer and lower outliers, with the highest cyclomatic complexity only

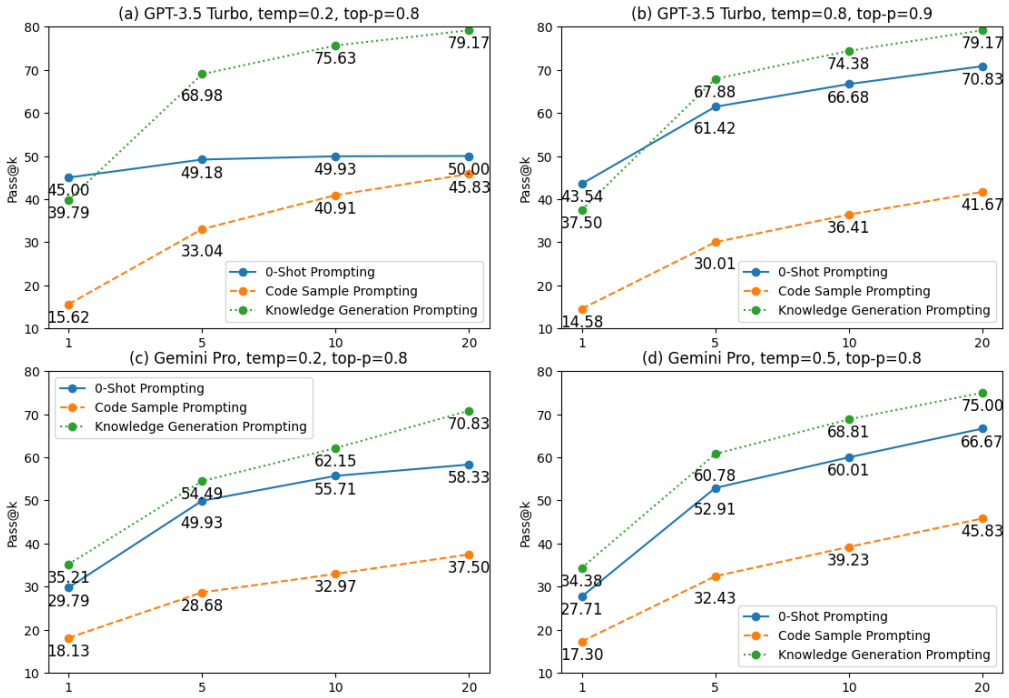

Fig. 5. Pass@k results for three prompting techniques.

reaching 10. In some cases, the complexity of code generated by other models is relatively high, even exceeding the cyclomatic complexity upper limit of 10 suggested by Thomas McCabe. We conducted a meticulous manual review of codes with a cyclomatic complexity greater than 10. The review revealed that the primary reason for the high complexity was the exhaustive legality checks performed on each input parameter. Additionally, we observed that a minority of the codes contained nested for loops ranging up to five or six levels. In summary, the codes generated by these LLMs demonstrate low complexity, with their cyclomatic complexity medians being less than or equal to 3.

4.5 RQ3: Prompt strategy selection

We employed code sample prompting and knowledge generation prompting techniques to generate codes for first-principles calculations and displayed their pass@k results in Figure 5. To achieve this, we selected the failed code samples that were most similar to the correct codes, using CodeBLEU [18] as the similarity metric, to use in our code sample prompting strategy. The experiment was conducted using models with conversational capabilities. Additionally, the parameter settings for this experiment were determined based on these models' best and worst performance observed under zero-shot prompting conditions.

Figure 5 clearly illustrates that, regardless of the parameter settings, introducing code samples as prompts significantly impairs the performance of both models. Detailed analysis indicates that for the GPT-3.5 Turbo, the pass@k results reduction ranges from 4.17% to 29.38%. For Gemini Pro, the decrease in the pass@k metric varies from a minimum of 10.41% to a maximum of 20.84%. After manually reviewing some of the generated code, we found that despite the prompts clearly stating

that the provided code contains errors, some model responses incorrectly treat the code in the prompts as correct, replying with, "The provided code is correct and has no issues." Even more perplexingly, some responses from the models are completely unforeseen, wrongly suggesting that the error in the code is due to the absence of the cmath header file. However, upon thorough review, we confirmed that the issue of the missing cmath header file does not exist in the provided prompts.

Conversely Figure 5 displays a significant improvement in the pass@k scores for both GPT-3.5 Turbo and Gemini Pro after applying the knowledge generation prompting strategy. Specifically, GPT-3.5 Turbo experienced a slight decrease of about 6% in pass@1, but its scores increased from 6.46% to 29.17% for pass@5 and above, reaching up to 79.17%. Similarly, Gemini Pro showed an improvement in performance across all pass@k evaluations after adopting the knowledge generation prompt strategy, especially when the temperature was set to 0.2, with a 12.50% increase in its pass@20 score. In summary, code sample prompting technique led to a decrease in the accuracy. Conversely, integrating relevant knowledge into the prompts notably improved pass@k scores.

5 DISCUSSION

5.1 Benchmark

Although our constructed ChemEval test set already covers a range of common development tasks in first-principles calculation software, we recognize the need to further expand the variety of tasks to enrich and perfect it. Therefore, we plan to continuously optimize ChemEval to more comprehensively reflect the programming challenges in this field.

5.2 Prompting techniques

In this study, we evaluated the effectiveness of zero-shot prompting, code sample prompting, and knowledge generation prompting techniques. However, there are many strategies proven to significantly improve code generation accuracy, such as chain of thought, that have not yet been included in our evaluation scope. In the future, we plan to extend our research to explore the potential impact of these efficient strategies on first-principles calculation code generation.

5.3 LLMs

While this study tested three advanced LLMs, there are many excellent models that remain unexplored. Fortunately, our evaluation framework is designed to be flexible and can easily be extended to other models. In the future, we will improve the framework to support the evaluation of a broader range of LLMs.

6 RELATED WORK

The potential of LLMs in code generation has attracted significant research interest. Chen et al. [2] developed the Codex model by fine-tuning it on existing GitHub code. They employed a crafted benchmark named HumanEval to assess Codex's performance in automatic programming tasks. Du et al. [3] introduced another benchmark, ClassEval, consisting of 100 programming tasks, to evaluate the efficacy of several leading LLMs in class-level code generation. Furthermore, Nguyen et al. [14] demonstrated how GitHub Copilot could automatically generate solutions for LeetCode programming challenges, with the accuracy of these solutions validated through the LeetCode platform.

Furthermore, LLMs have begun to play a role in the development, use, and testing of first-principles calculation software. Microsoft's research team leveraged GPT-4 to automatically generate

Python scripts and input files, thereby invoking specialized first-principles calculation software to perform calculation tasks [1]. In addition, they explored guiding GPT-4 in developing Hartree-Fock calculation methods using C++, although the evaluation of its correctness and complexity remains pending. In another research case, Qiu et al. [16] utilized LLMs to automatically generate a multitude of input files for fuzz testing the Siesta [5] software, successfully identifying 40 defects.

7 CONCLUSION

In this paper, we investigated the code generation for first-principles calculations by LLMs. Using ChemGen and ChemEval, we generated first-principles calculation codes with three LLMs and evaluated these codes. The results show that these models were able to solve up to 79.17% of the problems in ChemEval, with the median cyclomatic complexity of the codes generated by each model being less than or equal to 3. We also found that incorporating task-related knowledge into the prompts significantly enhances pass@k of the generated codes. Based on these findings, we believe LLMs can provide valuable support to developers in related fields.

REFERENCES

[1] Microsoft Research AI4Science and Microsoft Azure Quantum. 2023. The Impact of Large Language Models on Scientific Discovery: a Preliminary Study using GPT-4. arXiv:2311.07361 [cs.CL]

[2] Mark Chen, Jerry Tworek, et al. 2021. Evaluating Large Language Models Trained on Code. arXiv:2107.03374 [cs.LG]

[3] Xueying Du, Mingwei Liu, Kaixin Wang, Hanlin Wang, Junwei Liu, Yixuan Chen, Jiayi Feng, Chaofeng Sha, Xin Peng, and Yiling Lou. 2023. ClassEval: A Manually-Crafted Benchmark for Evaluating LLMs on Class-level Code Generation. arXiv:2308.01861 [cs.CL]

[4] James Finnie-Ansley, Paul Denny, et al. 2022. The robots are coming: Exploring the implications of openai codex on introductory programming. In *Proceedings of the 24th Australasian Computing Education Conference*. Association for Computing Machinery, New York, NY, USA, 10–19.

[5] Alberto García, Nick Papior, et al. 2020. Siesta: Recent developments and applications. *The Journal of chemical physics* 152, 20 (2020).

[6] W Hehre, W Latham, R Ditchfield, M Newton, and J Pople. 1970. Gaussian 70, quantum chemistry program exchange program no. 236. *University of Indiana, Bloomington, IN* (1970).

[7] Pierre Hohenberg and Walter Kohn. 1964. Inhomogeneous electron gas. *Physical review* 136, 3B (1964), B864.

[8] Ari Holtzman, Jan Buys, Li Du, Maxwell Forbes, and Yejin Choi. 2020. The Curious Case of Neural Text Degeneration. arXiv:1904.09751 [cs.CL]

[9] Georg Kresse and Jürgen Furthmüller. 1996. Efficient iterative schemes for ab initio total-energy calculations using a plane-wave basis set. *Physical review B* 54, 16 (1996), 11169.

[10] Hugo Larochelle, Dumitru Erhan, and Yoshua Bengio. 2008. Zero-data learning of new tasks. In *Proceedings of the 23rd National Conference on Artificial Intelligence - Volume 2* (Chicago, Illinois) *(AAAI'08)*. AAAI Press, 646–651.

[11] Jiacheng Liu, Alisa Liu, Ximing Lu, Sean Welleck, Peter West, Ronan Le Bras, Yejin Choi, and Hannaneh Hajishirzi. 2022. Generated Knowledge Prompting for Commonsense Reasoning. arXiv:2110.08387 [cs.CL]

[12] Ziyang Luo, Can Xu, et al. 2023. WizardCoder: Empowering Code Large Language Models with Evol-Instruct. arXiv:2306.08568 [cs.CL]

[13] Thomas J. McCabe. 1976. A complexity measure. In *Proceedings of the 2nd International Conference on Software Engineering* (San Francisco, California, USA) *(ICSE '76)*. IEEE Computer Society Press, Washington, DC, USA, 407.

[14] Nhan Nguyen and Sarah Nadi. 2022. An empirical evaluation of GitHub copilot's code suggestions. In *Proceedings of the 19th International Conference on Mining Software Repositories* (Pittsburgh, Pennsylvania) *(MSR '22)*. Association for Computing Machinery, New York, NY, USA, 1––5. https://doi.org/10.1145/3524842.3528470

[15] OpenAI. 2021. *Models – OpenAI API.* OpenAI. Retrieved March 8, 2024 from https://platform.openai.com/docs/models/gpt-3-5-turbo

[16] Feng Qiu, Pu Ji, Baojian Hua, and Yang Wang. 2023. Chemfuzz: Large language models-assisted fuzzing for quantum chemistry software bug detection. In *23rd IEEE International Conference on Software Quality, Reliability, and Security. Accompany.(QRS 2023)*.

[17] Xipeng Qiu, Tianxiang Sun, Yige Xu, Yunfan Shao, Ning Dai, and Xuanjing Huang. 2020. Pre-trained models for natural language processing: A survey. *Science China Technological Sciences* 63, 10 (2020), 1872–1897.

[18] Shuo Ren, Daya Guo, et al. 2020. CodeBLEU: a Method for Automatic Evaluation of Code Synthesis. arXiv:2009.10297 [cs.SE]

[19] Gemini Team, Rohan Anil, et al. 2023. Gemini: A Family of Highly Capable Multimodal Models. arXiv:2312.11805 [cs.CL]
[20] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. 2023. Attention Is All You Need. arXiv:1706.03762 [cs.CL]
[21] Andrew D White, Glen M Hocky, et al. 2023. Assessment of chemistry knowledge in large language models that generate code. *Digital Discovery* 2, 2 (2023), 368–376.
[22] Terry Yin. 2012. *Lizard: A simple code complexity analyzer.* Retrieved March 8, 2024 from https://github.com/terryyin/lizard