Instruction Placement within Prompts

David Gao
Department of Computer Science
Vanderbilt University
Nashville, United States
david.gao@vanderbilt.edu

***Abstract*—The world of software engineering has been transformed by large language models (LLMs) due to their ability to generate sequences from human prompts. This has the potential of accelerating innovation in all sectors around the world, and we must find the best ways to leverage this technology to hone its capabilities to our benefit.**

In this study, we find that GPT 3.5 generates code at various levels of correctness across different context lengths. This is statistically significant, and we propose the possibility of a hypothetical "sweet spot" where context is optimized to feed an LLM just the right amount of information to do a task in the best manner possible. We also find that the code generated by GPT 3.5 is significantly more concise when the instruction is placed within the context in comparison to placement before or after the context. This study introduces some possibilities for further research and presents statistically significant results from a small sample size.

***Index Terms*—prompt engineering, software engineering, large language models**

I. Introduction

Ever since the inception of large language models (LLMs), people have started using these models for a variety of tasks since they have proved to be very effective at understanding natural language [22]. These models have grown in size, and have demonstrated amazing performance at scale, and even rival humans at many cognitive tasks [24]. Once this power was revealed, there were many studies done in applying the LLMs and other transformer based models to specific tasks, ranging from personal assistance to programming, among many other applications which are still being explored as you read this paper [9].

Though many models exist, one of the most popular means of use for many people is through ChatGPT, a platform developed by OpenAI [17]. ChatGPT is powered by a general purpose large language model, meaning it does its best to respond to any prompt that you give it, no matter the topic [10], [27]. This has given any person, no matter their technical skill level, the ability to accomplish tasks at a scale that was inconceivable before. Even if programming is outside the skillset of an individual, they have the ability to automate tasks by crafting creative prompts and leveraging tools like ChatGPT to assist them. This concept has given rise to a field called prompt engineering, focusing on strategies to elicit the best responses from LLMs [26].

A very important application of these developments is on the software development lifecycle. Since code is part of the training corpus of these large language models, they have the ability to interpret and generate programs. This has led to a whole new subfield of research, which is the usage of various strategies to improve the programming capability of large language models [19]. This ability to generate code has the capability of transforming the future of computer science, beginning at the education level [1], [18]. It will even influence the way in which more seasoned programmers in industry settings program [29].

In this paper, we hope to study the impact of prompt structure on the quality of code generation from the various models that OpenAI uses on its ChatGPT platform. Specifically, we propose the following research question which encapsulates the goals of this research study:

> How does the placement of an instruction in relation to the context in a prompt impact the generational ability of large language models for Python code?

To answer this question, we conduct a controlled experiment on various programming tasks which are accompanied by the context that the task is in relation to. We define three locations where an instruction may be placed:

- Before the context of a prompt.
- After the context of a prompt.
- Within the context of a prompt.

Using this framework, we test 25 different prompts of different **lengths of context** and **complexity of instructions** and evaluate the performance of the same prompt with different instruction placement in order to try to answer our research question. These are tested independently across multiple trials, and statistically evaluated across the length of difficulty of a prompt.

With this analysis, we hope to provide an insight into this specific area of prompt engineering, and guide users and researchers of large language models towards better strategies of generating code, and placement of instructions in general. There have been no specific studies that we know of that address this problem, so this research paper can serve as a baseline and inspiration for future work in instruction placement.

The rest of this paper is organized as follows. Section II reviews existing work that relates to this topic, and discusses any insights we draw from past studies. In Section III, we outline the methodology of our experiment, and dive into our methods. Section IV analyzes the results from the experiment. Then, Section V discusses the results and ponders the implications. In Section VI, we consider the limitations of the work that we have done, and lay out potential areas for future work. Our work is summarized and concluded in Section VII.

II. Existing Work

In this section, we will review existing work done in prompt engineering, use of LLMs for code generation, and the societal effects of LLMs generated code in various settings.

A. LLMs for Code Generation

Automated code generation has been an area of interest for software engineering long before transformer models came along. However, with the powers of large language models, more researchers have become curious about their capabilities in generating code.

One of the most important advances in this area are the large language models which are trained with code data specifically, starting with the Codex model released by OpenAI in 2021 [2]. This has brought on an avalanche of research in training new models to generate code in a multitude of languages [15], [16], [21]. All of this work has shown significant progress and has been incorporated into popular software engineering tools such as GitHub Copilot.

There has also been work done to evaluate the capabilities of these models and tools. Starting with the Copilot, which we just discussed, various studies have evaluated the effectiveness of the code completion tool [3], [8], [14]. Beyond the Copilot, various models have been tested on coding tasks that are common for software engineers to come across such as algorithm development and optimization [11], program repair [6], and program synthesis [28] among many other tasks.

B. Prompt Engineering

There has also been a lot of work done on the strategies to optimize the output of large language models in prompt engineering. Since the outputs of large language models are probabilistic, the changing of a prompt can change the outcome and potentially solve issues from previous outputs [4]. This has been looked at from a wide variety of settings outside of computer science such as models which generate artwork [13], legal assessment [20], and healthcare applications [23].

Though prompting techniques have been defined by White *et al.* [26], there is still a lot of work to be done in the declaration of best practices in different settings when prompting an LLM.

C. LLM Impact on the Future of Code

Lastly, we look at studies which evaluate the impact of large language models on various coding activities. We begin with the impact on students who will lead the next generation of programmers. Students now have access to these models, so many may take shortcuts, utilizing the LLMs to code for them, rather than learning the skills themselves [1]. This can be quite an issue when the code generated by popular models like GPT can have varying degrees of correctness [12].

III. Methodology

In this section, we will cover specific research questions we want to explore and the methodology in which the experiments were conducted.

A. Research Questions

We break down our research down to three main research questions which help us answer our overall research question:

1) Does the placement of instructions result in differences in the functional correctness of code produced by an LLM across different instruction complexities and context lengths?
2) Does the placement of instructions result in differences in the brevity of solutions produced by an LLM across different instruction complexities and context lengths?
3) Does the placement of instructions result in differences in the pure response length produced by an LLM across different instruction complexities and context lengths?

B. Test Data

The dataset that we run our experiments on is a hand selected sample of Python programming questions created by GPT 4 and manually revised by the author as shown in Table I. These programming questions are general purpose programming questions that revolve around implementation of a single function in the context of existing code. Each question consists of:

- An existing body of code (which acts as context for the LLM)
- An instruction (which instructs the LLM in the code generation task)

They range in topics covered, from simple mathematical operations to sorting algorithms to try to simulate the tasks a software engineer would come across, and each question has two categories that define it:

- Difficulty (a measure of the complexity of the instruction): We group the difficulty of an instruction into four buckets using the Flesch-Kincaid readability score where a higher score represents sentences that are easier to understand [7].
    1) >80 : Most readable, most comparable to a 5th or 6th grade reading level
    2) 60-80 : Fairly readable, most comparable to a middle school reading level
    3) 40-60 : Medium, most comparable to a high school reading level
    4) 20-40 : Fairly difficult, most comparable to a college reading level
    5) <20 : Most difficult, most comparable to college graduate and professional reading level
- Length (a measure of the length of the context): We group the length of prompts into four levels:
    - Question 1 (<500 characters)
    - Question 2 (500-1000 characters)
    - Question 3 (1000-1500 characters)

TABLE I
TESTING FRAMEWORK

Flesch-Kincaid Score	**Length of Context (in characters)**			
	<500	**500-1000**	**1000-1500**	**>1500**
>80	Before/After/Within	Before/After/Within	Before/After/Within	Before/After/Within
60-80	Before/After/Within	Before/After/Within	Before/After/Within	Before/After/Within
40-60	Before/After/Within	Before/After/Within	Before/After/Within	Before/After/Within
20-40	Before/After/Within	Before/After/Within	Before/After/Within	Before/After/Within
<20	Before/After/Within	Before/After/Within	Before/After/Within	Before/After/Within

– Question 4 (>1500 characters)

To build this dataset, the author sampled Python questions by prompting GPT-4, and requiring the response to fall within the specified character count. This was then verified with the capabilities of Google Docs to ensure adherence to the character limits.

For each selected Python question, the author then prompted GPT-4 to create a corresponding instruction to the question it just created within the Flesch-Kincaid scale specified. This was verified by online Flesch-Kincaid calculators. This was repeated until five instructions were created which fall into the categories defined above.

C. Experimental Design

The approach to prompting was standard across the entire experiment, and GPT-3.5 was the model evaluated. In order to overcome confusion of context, each question was tested independently in its own session. The LLM was also not restricted in its output, maintaining an environment that would be most similar to public usage by developers.

For each programming question, three prompts were created with the instruction and context defined before and no other guides for the LLM.

1) The instruction followed by the context, where the function that is to be implemented simply contains a ”pass” Python command as a placeholder.
2) The context followed by the instruction, where the function that is to be implemented simply contains a ”pass” Python command as a placeholder.
3) The instruction within the context, where the instruction exists as a comment in the location where the unimplemented function will be.

With these three prompts, each was submitted to the two models independently, and the output was written to a file. The file was then manually cleaned up to remove any non-code portions that were outputted by the LLM.

These files were then run through a test suite of 10 test cases for the designated question to give a numerical score for functional correctness. They were also evaluated across other measures and the final three measures that we use to evaluate the solution of an LLM-generated code block were:

1) **Functional Correctness:** the number of test cases passed out of the 10 provided
2) **Code Conciseness:** the number of lines of code that was used to solve the problem
3) **Token Verboseness:** the total number of tokens used in the output of the LLM in responding to the prompt

Then, each problem laid out in Table 1 is run, and the responses are saved in a directory along with the token count of the output (a feature provided by OpenAI in their API response). Once all responses are generated, the Python code for the designated function was extracted manually by the author and placed within a separate file containing the the rest of the context. During manual extraction, the lines of code in the solution were counted by the author and recorded.

IV. Results and Discussion

In this section, we present the results of our experiments and display analyses done with the data to answer each research question. We also discuss implications of the results and real-world impact of what we found.

A. Research Question Analysis

We will explore the results, and discuss their role in the answering of our research questions.

*1) **Does the placement of instructions result in differences in the functional correctness of code produced by an LLM across different instruction complexities and context lengths?**:* If we take a look at Table III, it seems like there isn’t really an obvious pattern when it comes to the correctness of the solutions generated by the LLM. Question 1, the shortest context size, seems to indicate that the placement of the prompt within the context has the best performance with 36 total test cases passed across all complexities of instructions. We theorize that at this scale, the problem is pretty much a Masked Language Model (MLM) question as proposed in the original BERT paper [5]. This, in comparison to instructions before and after the context, is a much more defined task, which possibly contributes to the success.

As we move onto longer questions, however, it seems like there is less of a difference between placing the instruction before the context and placing the instruction within the context. We theorize that by introducing, the problem beforehand, the model is encouraged to ”think” more about the solution while parsing through the context. This might lead to some chain-of-thought behavior which improves its performance [25].

Once we get to the third question, the correctness seems to even out. We hypothesize that this might be a ”sweet spot” where the LLM is getting the right amount of information to solve a problem correctly without overloading its attention mechanism [22].

TABLE II
FUNCTIONAL CORRECTNESS RESULTS (CORRECT TEST CASES OUT OF 10)

	Q1: <500			**Q2: 500-1000**			**Q3: 1000-1500**			**Q4: >1500**			
Flesch-Kincaid	Pre	Intra	Post	Pre	Intra	Post	Pre	Intra	Post	Pre	Intra	Post	**Sum**
>80	10	10	1	9	9	0	10	10	10	8	8	8	**93**
60-80	1	10	1	9	9	9	10	10	10	8	8	8	**93**
40-60	1	1	10	9	9	9	10	10	10	1	1	8	**79**
20-40	6	5	10	9	9	0	10	10	10	8	8	8	**93**
<20	5	10	10	9	9	9	10	10	10	8	8	8	**106**
Sum	**23**	**36**	**32**	**45**	**45**	**27**	**50**	**50**	**50**	**33**	**33**	**40**	**464**

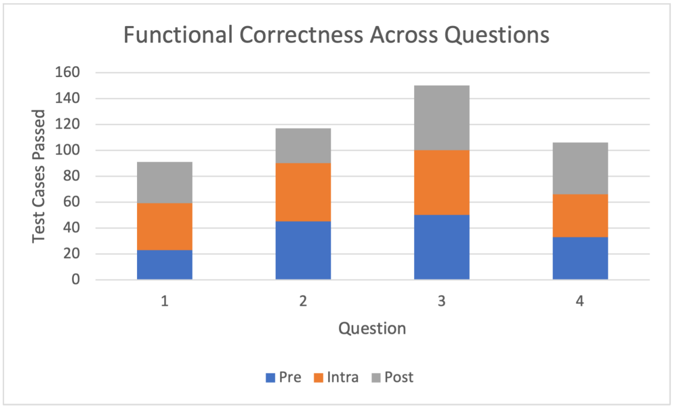

Fig. 1. Correctness Evaluation across questions.

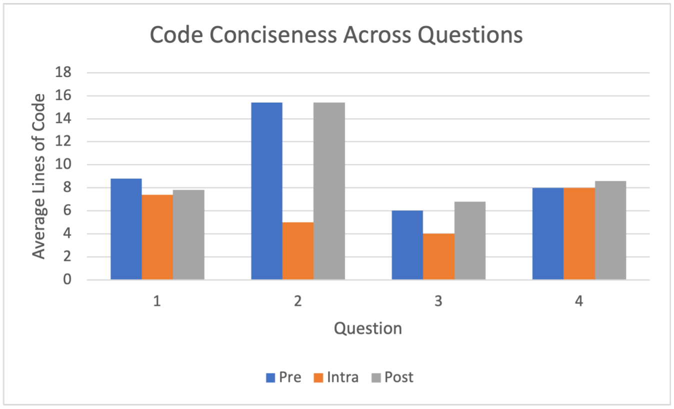

Fig. 2. Code Conciseness across questions.

TABLE III
ANOVA FOR FUNCTIONAL CORRECTNESS

	Df	Sum Sq	Mean Sq	F value	Pr(>F)
arrangement	2	6.6	3.32	0.388	0.68033
length	3	125.5	41.82	4.895	0.00464 **
complexity	4	30.4	7.60	0.889	0.47718
Residuals	50	427.2	8.54		

This also contributes to what we see in the fourth question, where putting the prompt after the context passes the most test cases. As the context length increases too much, the model might begin to ”forget” certain things that were introduced earlier, so the instruction at the end ensures the correct response.

In Table III, we perform an analysis of variance test (ANOVA) with randomized complete block designs (RCBD) to check the significance of the differences in test cases passed. We test if the test cases passed is a result of the instruction placement, adding in the context length and instruction complexity as blocking variables to isolate the instruction placement. We see that the F value for our instruction placement variable, arrangement, is not significant. However, we do see that the length of context is significant in the ANOVA test. This statistically proves that the question has a significant effect on the correctness of the response, and that is all we can conclude. We may hypothesize about the ”sweet spot” mentioned earlier, but there are too many factors such as question difficulty which impact these results to make a definite conclusion.

If this hypothetical sweet spot exists, however, the implications could be very beneficial to LLM users everywhere.

2) ***Does the placement of instructions result in differences in the brevity of solutions produced by an LLM across different instruction complexities and context lengths?:*** We now evaluate on our second research question, code conciseness, with the amount of lines of code used in the solution generated by the GPT 3.5 model. If we look at the results shown in Table VI, we can see the lines of code given by the LLM in response to all of the prompts. This measure is somewhat arbitrary, given that each unique coding task has its limits in the amount of code it can be expressed in, but we can see one very obvious pattern.

In each question, the prompts where the instruction is placed within the context has the most concise code! This brings up our Masked Language Model theory once again, since it does seem to be a pattern across all context lengths this time. This is visualized in Fig. 2. and you can clearly see that our placement of instructions within context, or ”intra”, always has less lines of code.

We also test the statistical significance of this hypothesis using ANOVA with RCBD as seen in Table IV, same as before. This time, we can see that both arrangement and length have a significant F value, which means that we can reject the null hypothesis that the lines of code are equal in the population, and they are influenced by instruction placement (arrangement) and context length (length).

Since we do see statistical significance in the arrangement variable, we decided to implement Tukey’s HSD test to

TABLE IV
CODE CONCISENESS RESULTS (LINES OF CODE)

	Q1: <500			**Q2: 500-1000**			**Q3: 1000-1500**			**Q4: >1500**			
Flesch-Kincaid	Pre	Intra	Post	Pre	Intra	Post	Pre	Intra	Post	Pre	Intra	Post	**Average(rounded)**
>80	7	9	6	12	0	12	6	4	6	12	8	9	**8**
60-80	8	9	6	16	12	18	6	4	6	8	8	10	**9**
40-60	6	6	8	19	1	13	6	4	6	9	8	8	**8**
20-40	16	7	5	12	0	12	6	4	6	8	8	8	**8**
<20	7	6	14	18	12	22	6	4	10	8	8	8	**10**
Average(rounded)	**9**	**7**	**8**	**15**	**5**	**15**	**6**	**4**	**7**	**9**	**8**	**9**	**8**

TABLE V
ANOVA FOR CODE CONCISENESS

Source	Df	Sum Sq	Mean Sq	F value	Pr(>F)
arrangement	2	175.4	87.72	7.439	0.00149 **
length	3	306.7	102.24	8.670	9.83e-05 ***
complexity	4	67.2	16.81	1.425	0.23927
Residuals	50	589.6	11.79		

TABLE VI
SIGNIFICANT TUKEY COMPARISONS FOR CODE CONCISENESS

Factor	**Comp**	**Diff**	**Lower CI**[a]	**Upper CI**[a]	**p adj**
Place	Intra-Pre	-1.70	-6.322	-1.077	0.00366
Place	Post-Intra	3.55	0.927	6.172	0.00545

[a]95% Confidence Intervals are used

obtain confidence intervals for pairwise comparisons as an extension of the ANOVA as seen in Table V. We extracted the significant confidence intervals to display. The pairwise difference between ”intra” and ”pre” has a confidence interval that is ranges from -6.322 to -1.077. This emphasizes the significance that the instruction placement within the context results in less lines of code than placing the instruction before the context. The same is for the second confidence interval, which shows the difference between ”intra” and ”post”, demonstrating more concise code in comparison to the placement of instructions after the context.

3) ***Does the placement of instructions result in differences in the pure response length produced by an LLM across different instruction complexities and context lengths?:*** Lastly, we look at the wordiness of a response across the different prompts in Table VIII. This is an important factor for prompt engineering since companies like OpenAI charge users based on token input and output. Unfortunately, we don’t see any significant patterns in these measurements. We hypothesize that this is due to the stochastic nature of generation tasks. LLMs generate each word based on its probability conditional on previously generated words, so the sequences that are generated are somewhat random in nature, and can have a variety of results.

We confirm this through the ANOVA with RCBD once again and see no significant results. We leave this to future studies to analyze.

B. Implications

We will discuss what all of this means for the field of Natural Language Processing (NLP) and the impact on LLM use.

This work expands upon our previous knowledge on prompt engineering and can serve as a guide for future work while acting as a rule of thumb for prompt engineers. The proposed ”sweet spot” hypothesized in this work may be a great realization for many tasks, and can help users break down larger tasks and extend shorter tasks to achieve optimal responses from LLMs. There may also be an entirely new research vertical where different transformer architectures and their context limits are tested for ”sweet spots” in order to optimize their production.

We also see many implications in the transformation of prompts into different formats such as the Masked Language Model. If this can be exploited by users, it might lead to an entirely different way in communicating with LLMs.

V. LIMITATIONS AND FUTURE WORK

One of the main limitations of this work is the limited size of the tests conducted. While we have tried to ensure that the questions tested represent a variety of programming topics of different difficulties, there may be some bias from the sample selection. We can see this within some of the results, since the length of context significantly impacts two of our research questions as discussed earlier. This can be addressed by further experimentation at scale. Hopefully, larger samples across each context length would remove bias and uncover possible statistical significance that can be attributed to the actual context length rather than discrepancies within perceived question difficulty.

Furthermore, a significant bias comes from our limited knowledge of the corpus that the GPT model was trained on. A subset of the test data set may have significant overlap with training data, leading to responses that are memorized. We have no knowledge of this for GPT, so similar experiments on open sourced LLMs such as Llama 2 would be provide better insight and can maintain consistency on out of sample testing. We leave this for future studies.

Another future direction might be the testing of instruction placement in natural language prompts, rather than code, since the issue of instruction-context assortment still is an important factor in the performance of an LLM for general usage. There also lies value in prompts with multiple instructions

TABLE VII
TOKEN VERBOSENESS RESULTS (TOKENS OUTPUTTED)

	Q1: <500			**Q2: 500-1000**			**Q3: 1000-1500**			**Q4: >1500**			
Flesch-Kincaid	Pre	Intra	Post	Pre	Intra	Post	Pre	Intra	Post	Pre	Intra	Post	**Average(rounded)**
>80	210	281	294	213	350	470	137	655	396	503	406	290	**350**
60-80	305	265	394	560	277	328	208	374	186	200	184	414	**308**
40-60	335	364	229	285	139	583	398	374	156	904	721	470	**413**
20-40	495	276	325	605	215	422	138	374	228	293	228	249	**321**
<20	337	370	530	274	274	562	204	414	438	248	414	407	**373**
Average(rounded)	**336**	**311**	**354**	**387**	**251**	**473**	**217**	**438**	**281**	**430**	**391**	**366**	**353**

TABLE VIII
ANOVA FOR TOKEN VERBOSENESS

Source	Df	Sum Sq	Mean Sq	F value	Pr(>F)
arrangement	2	7550	3775	0.157	0.855
length	3	62173	20724	0.863	0.467
complexity	4	85097	21274	0.886	0.479
Residuals	50	1201098	24022		

and the possibilities in the instruction placement distribution of a prompt.

VI. CONCLUSION

We conclude this paper with the key takeaways and findings of the research study.

- **Large Language Model architecture may have a range of input tokens where their responses are the best.** We discovered the potential of a ”sweet spot” where the large language model that we used for the study, GPT 3.5, achieved the best results. This is a sweet spot in terms of the length of input given to the LLM for context in terms of characters. This can be optimized by users everywhere to obtain the best results, and may improve productivity when prompting any language model. The key is the LLM needs enough information to complete the task exactly how you want it to, but not too much information to the point that it loses track of some of the things you told it.
- **Prompting LLMs in a Masked Language Model format may result in better results.** By changing the way in which we prompt large language models, we might adapt to the strategies which allow for easier retrieval for the models rather than what is natural for people. This might require a shift in the prompting strategies of everyday users, but may result in more efficient responses, especially in the area of code generation.
- **There are many unknowns out there in the field of prompt engineering.** Specifically, there can be so much more learned about how we can harness the probabilistic generation capabilities of these large language models. The use of LLMs for software engineering still has a ways to go, as shown by our results, but our findings show that there is a lot of potential to what humans can achieve with these tools, if they know how to utilize them.

Beyond these key points, I would like to discuss a bit more on the trajectory of prompt engineering for software engineering tasks. Currently, software engineering has already been transformed by the use of LLMs, and programmers of all levels have the ability to generate code with these models, and it poses an important question: What should be the goal of LLM usage in software engineering? Do we want to automate more tasks and achieve high productivity or do we want to enhance the existing capabilities of software engineers? These two choices proposed are not necessarily orthogonal, since we have no idea what innovations may spring up in the near future, and it is important that we evaluate the effects before we dive into their possibilities.

The Github repository containing the experiments can be found at: https://github.com/davidgaofc/instruction-placement-experiments.

ACKNOWLEDGEMENT

I acknowledge the use of ChatGPT in the production and evaluation of the experiments done in this study.

REFERENCES

[1] Brett A Becker, Paul Denny, James Finnie-Ansley, Andrew Luxton-Reilly, James Prather, and Eddie Antonio Santos. Programming is hard-or at least it used to be: Educational opportunities and challenges of ai code generation. In *Proceedings of the 54th ACM Technical Symposium on Computer Science Education V. 1*, pages 500–506, 2023.

[2] 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. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374*, 2021.

[3] Arghavan Moradi Dakhel, Vahid Majdinasab, Amin Nikanjam, Foutse Khomh, Michel C Desmarais, and Zhen Ming Jack Jiang. Github copilot ai pair programmer: Asset or liability? *Journal of Systems and Software*, 203:111734, 2023.

[4] Paul Denny, Viraj Kumar, and Nasser Giacaman. Conversing with copilot: Exploring prompt engineering for solving cs1 problems using natural language. In *Proceedings of the 54th ACM Technical Symposium on Computer Science Education V. 1*, pages 1136–1142, 2023.

[5] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. Bert: Pre-training of deep bidirectional transformers for language understanding. *arXiv preprint arXiv:1810.04805*, 2018.

[6] Zhiyu Fan, Xiang Gao, Martin Mirchev, Abhik Roychoudhury, and Shin Hwei Tan. Automated repair of programs from large language models. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*, pages 1469–1481. IEEE, 2023.

[7] Rudolf Flesch. Flesch-kincaid readability test. *Retrieved October*, 26(3):2007, 2007.

[8] Saki Imai. Is github copilot a substitute for human pair-programming? an empirical study. In *Proceedings of the ACM/IEEE 44th International Conference on Software Engineering: Companion Proceedings*, pages 319–321, 2022.

[9] Jean Kaddour, Joshua Harris, Maximilian Mozes, Herbie Bradley, Roberta Raileanu, and Robert McHardy. Challenges and applications of large language models. *arXiv preprint arXiv:2307.10169*, 2023.

[10] Jan Kocoń, Igor Cichecki, Oliwier Kaszyca, Mateusz Kochanek, Dominika Szydło, Joanna Baran, Julita Bielaniewicz, Marcin Gruza, Arkadiusz Janz, Kamil Kanclerz, et al. Chatgpt: Jack of all trades, master of none. *Information Fusion*, page 101861, 2023.

[11] Yujia Li, David Choi, Junyoung Chung, Nate Kushman, Julian Schrittwieser, Rémi Leblond, Tom Eccles, James Keeling, Felix Gimeno, Agustin Dal Lago, et al. Competition-level code generation with alphacode. *Science*, 378(6624):1092–1097, 2022.

[12] Jiawei Liu, Chunqiu Steven Xia, Yuyao Wang, and Lingming Zhang. Is your code generated by chatgpt really correct? rigorous evaluation of large language models for code generation. *arXiv preprint arXiv:2305.01210*, 2023.

[13] Vivian Liu and Lydia B Chilton. Design guidelines for prompt engineering text-to-image generative models. In *Proceedings of the 2022 CHI Conference on Human Factors in Computing Systems*, pages 1–23, 2022.

[14] Nhan Nguyen and Sarah Nadi. An empirical evaluation of github copilot's code suggestions. In *Proceedings of the 19th International Conference on Mining Software Repositories*, pages 1–5, 2022.

[15] Erik Nijkamp, Bo Pang, Hiroaki Hayashi, Lifu Tu, Huan Wang, Yingbo Zhou, Silvio Savarese, and Caiming Xiong. Codegen: An open large language model for code with multi-turn program synthesis. *arXiv preprint arXiv:2203.13474*, 2022.

[16] Gabriel Poesia, Oleksandr Polozov, Vu Le, Ashish Tiwari, Gustavo Soares, Christopher Meek, and Sumit Gulwani. Synchromesh: Reliable code generation from pre-trained language models. *arXiv preprint arXiv:2201.11227*, 2022.

[17] John Schulman, Barret Zoph, Christina Kim, Jacob Hilton, Jacob Menick, Jiayi Weng, Juan Felipe Ceron Uribe, Liam Fedus, Luke Metz, Michael Pokorny, et al. Chatgpt: Optimizing language models for dialogue. *OpenAI blog*, 2022.

[18] Brad Sheese, Mark Liffiton, Jaromir Savelka, and Paul Denny. Patterns of student help-seeking when using a large language model-powered programming assistant. *arXiv preprint arXiv:2310.16984*, 2023.

[19] Jiho Shin, Clark Tang, Tahmineh Mohati, Maleknaz Nayebi, Song Wang, and Hadi Hemmati. Prompt engineering or fine tuning: An empirical assessment of large language models in automated software engineering tasks. *arXiv preprint arXiv:2310.10508*, 2023.

[20] Dietrich Trautmann, Alina Petrova, and Frank Schilder. Legal prompt engineering for multilingual legal judgement prediction. *arXiv preprint arXiv:2212.02199*, 2022.

[21] Priyan Vaithilingam, Tianyi Zhang, and Elena L Glassman. Expectation vs. experience: Evaluating the usability of code generation tools powered by large language models. In *Chi conference on human factors in computing systems extended abstracts*, pages 1–7, 2022.

[22] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. *Advances in neural information processing systems*, 30, 2017.

[23] Jiaqi Wang, Enze Shi, Sigang Yu, Zihao Wu, Chong Ma, Haixing Dai, Qiushi Yang, Yanqing Kang, Jinru Wu, Huawen Hu, et al. Prompt engineering for healthcare: Methodologies and applications. *arXiv preprint arXiv:2304.14670*, 2023.

[24] Jason Wei, Yi Tay, Rishi Bommasani, Colin Raffel, Barret Zoph, Sebastian Borgeaud, Dani Yogatama, Maarten Bosma, Denny Zhou, Donald Metzler, et al. Emergent abilities of large language models. *arXiv preprint arXiv:2206.07682*, 2022.

[25] Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Fei Xia, Ed Chi, Quoc V Le, Denny Zhou, et al. Chain-of-thought prompting elicits reasoning in large language models. *Advances in Neural Information Processing Systems*, 35:24824–24837, 2022.

[26] Jules White, Quchen Fu, Sam Hays, Michael Sandborn, Carlos Olea, Henry Gilbert, Ashraf Elnashar, Jesse Spencer-Smith, and Douglas C Schmidt. A prompt pattern catalog to enhance prompt engineering with chatgpt. *arXiv preprint arXiv:2302.11382*, 2023.

[27] Tianyu Wu, Shizhu He, Jingping Liu, Siqi Sun, Kang Liu, Qing-Long Han, and Yang Tang. A brief overview of chatgpt: The history, status quo and potential future development. *IEEE/CAA Journal of Automatica Sinica*, 10(5):1122–1136, 2023.

[28] Frank F Xu, Uri Alon, Graham Neubig, and Vincent Josua Hellendoorn. A systematic evaluation of large language models of code. In *Proceedings of the 6th ACM SIGPLAN International Symposium on Machine Programming*, pages 1–10, 2022.

[29] Zibin Zheng, Kaiwen Ning, Jiachi Chen, Yanlin Wang, Wenqing Chen, Lianghong Guo, and Weicheng Wang. Towards an understanding of large language models in software engineering tasks. *arXiv preprint arXiv:2308.11396*, 2023.