GPTeams: LLM Collaboration in Software Engineering

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

Lincoln Murr
Department of Computer Science
Vanderbilt University
Nashville, United States
lincoln.d.murr@vanderbilt.edu

***Abstract*—Programming tasks are complex, and people in the software engineering industry have continuously sought to automate their code, no matter the context. With the rise of generative AI tools such as ChatGPT, this task has been a hot research topic, and simple programming tasks can be solved very effectively using these transformer-based tools. However, there are always limitations to the size of the input into a model using a single prompt and the ability of the model to maximize its attention mechanism to focus on the important parts of the problem. Inspired by human software engineering interactions, we propose a new algorithm that divides up a task and prompts collaboratively to achieve a final goal. This mechanism splits up a problem using the sliding-window technique, utilizes "worker" prompts for each window, and then combines all worker solutions using a "master" prompt which outputs the final product. This provides an abstraction that can be scaled up infinitely, yielding a new perspective on the limits of large language models. The collaboration between LLM prompts also provides a mechanism for self-evaluation, potentially improving the final output of a task.**

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

I. Introduction

As large language models (LLMs) become integrated into various industries, a large focus has been on the vertical scaling of these models to increase their effectiveness [18]. However, there have also been positive results from researching the interactions between the models and utilizing agents to create synergy in the tasks that we use them for [23].

One specific application of these Large Language Models is the ability to produce code. With generative AI tools such as ChatGPT and Github Copilot [2], software developers have increased productivity in their daily tasks. Though general-purpose LLMs have been adequate at this task, many researchers have explored ways to improve AI generation of code, using strategies such as training code-specific models from scratch [4], [17], fine-tuning models for code [8], [10], and prompt engineering [3], [16].

Specifically, the advent of the prompt engineering field has proved to be a valuable tool for the general population thanks to commercial LLM chatbots such as ChatGPT, Claude, and Bard. Using various prompting techniques, users can elicit better responses from these Large Language Models to solve their problems better [19]. This allows for developers to improve the generative abilities of the services without spending the time, compute, and money needed to make their own models.

As developers utilize general-purpose LLMs with prompt engineering, the need for long context becomes critical, since codebases often comprise many lines of code across multiple files. Large language models are inherently restricted by the size of their input since their attention mechanism [15] can only focus on so much. This requires developers to find unique solutions to ensure a model can understand the big picture of their problem while maintaining a focused understanding of the task at hand.

In this paper, we propose a new framework of prompt engineering, eliciting a simulated collaboration within LLM models. This can be used to break down large software engineering tasks where many parts are interdependent, leveraging prompting to break down a problem into many parts, and then piecing together a final solution after a few intermediary steps. We introduce the following research question for our study:

> How does the collaboration of large language models through prompt engineering impact the quality of code generated by the LLMs for software engineering tasks?

We hope to achieve a new paradigm where the LLMs we have today can be scaled horizontally as "workers", providing a new scaling strategy that trades the exponential costs of larger models with the adjustable costs of team prompting.

To analyze the feasibility of this concept, we abstract a team prompting strategy to split up the context of a programming task to multiple "workers" using a basic sliding window approach, allowing the "workers" to have overlapping contexts, including the task at hand. The results are then combined and optimized by a "master", providing the team's solution to the problem. This flexible framework will be built on top of the GPT-3.5 API and will be tested on long context coding problems curated by the authors.

We evaluate each coding problem with varying amounts of teamwork (ordered from least to greatest) as follows.

1) A zero-shot LLM prompt (single worker).
2) Two "workers", whose responses are combined by a "master" (pair-programming).

3) Three "workers", whose responses are combined by a "master".

From the final product of these teamwork adjustments, we evaluate the results on three metrics:

- Functional Correctness - the number of test cases passed.
- Code Complexity - measured using McCabe's complexity scale [12].
- Performance - measured by the run time of code written.

The rest of this paper is formatted as follows. Section II reviews existing work related to this topic, and discusses any insights we draw from past studies. In Section III, we propose our algorithm for LLM collaboration in software engineering tasks. Section IV reviews our experimental methodology. Then, Section V displays our results. In Section VI, we discuss the implications of our results and the impact they might have. We then discuss the limitations of our work in Section VII, and our work is summarized and concluded in Section VII.

II. Related Work

Many studies have tried to tackle the issue of code generation using LLMs, and we will discuss existing works that relate to our study.

There have been many studies in the area of collaboration by AI agents, which relates directly to the intentions of this study [5], [11], [14], [22].

For example, Talebirad *et al.* examines the development of a multi-agent system to model human interactions [14]. This study proposes a setting where LLMs play different parts and interact with one another, creating a way to mimic collaboration.

A similar concept is also explored by Liu *et al.* where task performance improves when multiple agents collaborate on a query in a dynamic fashion [11]. Systems like this provide an architecture that can boost AI performance and allow LLMs to interact in a way that creates outcomes that may be a new paradigm in AI's rapidly changing status quo.

Wu *et al.* also evaluates the effects of chaining prompts together in a workflow and shows its effectiveness, using the output of one prompt as the input of another [21].

In addition to studying the collaboration of AI agents, there has also been a lot of work that evaluates the effectiveness of large language models with prompt engineering [6], [7], [19].

This study is also inspired by the work in extending LLMs input limits and the amount of content they can track at a time. The Longformer proposed by Beltagy *et al.* is a significant inspiration for this study, with the sliding window technique being the main takeaway to improve the scalability of attention [1].

With these techniques, we hope to improve the ability of LLMs in software engineering tasks, building off the many works in this area [13], [20].

III. Proposed Algorithm

We propose the following algorithm to simulate a software engineering team of language models. This algorithm allows for multiple "worker" prompts to see various parts of the codebase and give their attempt at the solution. Then, a "master" prompt puts all of the "worker" prompts together, giving us a final output. Figure 1 visualizes this process and displays the architecture.

Algorithm 1 Sliding Window Work Distribution Function

Require: $file$, $worker\_count$
  for $i = 1$ **to** $worker\_count$ **do**
    $task, splitable \leftarrow$ SplitFileIntoFunctions($file$)
    $length \leftarrow$ lengthOf($splitable$)
    $baseSegmentLength \leftarrow length \div worker\_count$
    $overlap \leftarrow$ function to calculate worker overlap
    $start \leftarrow$ function to calculate start index
    $end \leftarrow$ function to calculate end index
    $worker\_context \leftarrow extract(splitable, start, end) + task$
    Prompt worker with their context and the task
    Store responses in worker directory
  end for

Algorithm 2 Master Prompt Function

Require: $worker\_output\_directory$
  $myList \leftarrow []$
  for each $file$ in $worker\_output\_directory$ **do**
    $task\_version \leftarrow$ Take only the python code from file
    append $task\_version$ **to** $myList$
  end for
  Join all entries in $myList$
  Prompt master with all worker solutions
  Store response in directory

To begin, we utilize a custom sliding window algorithm to divide the codebase into ***n*** sections with ***n*** being the number of workers utilized. The pseudocode for this is displayed in Algorithm 1. This study defines a "section" as any block of code surrounded by an empty line. We intend this to be a single function or class within the larger scope of the code. We remove all import statements.

We ensure that all workers get to see the task, which is the function we ultimately want to implement with this strategy. Apart from that, they all see partially overlapping code that can contribute to their interpretation of the task at hand.

Once the codebase is split up and each worker has their context, we prompt the workers to get **n** versions of the solution to the task. These are stored within a specific directory.

Then, the master prompt collects all responses from the workers and parses out only the Python code, which is supposed to be the solution to the task function. The master prompt takes all ***n*** solutions and is told to create a final product, taking the good parts from each solution and leaving out the bad parts.

With this concept, we can expand the length of codebases being analyzed without restrictions. We can simply add more workers to accommodate lengthier code.

We also see the potential of adding intermediary aggregating steps once there are too many workers. This would allow for

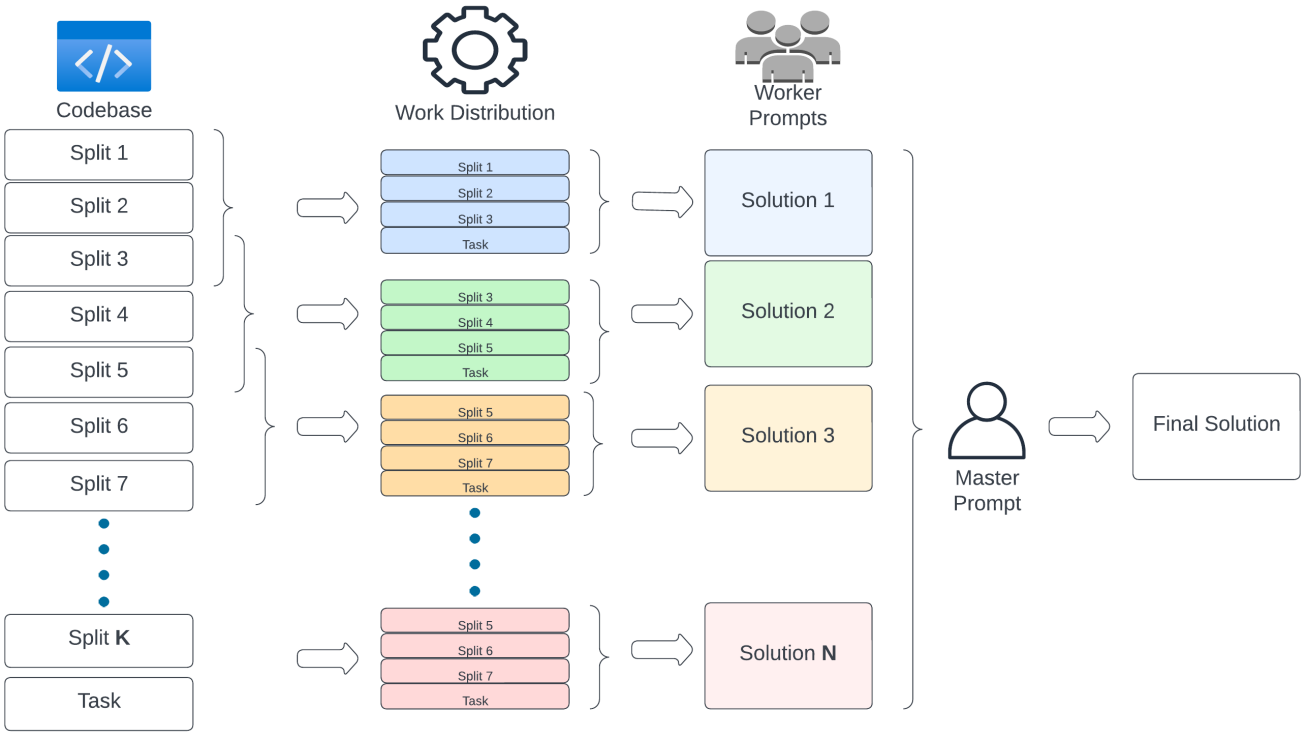

Fig. 1. Architecture of prompting teamwork.

each "assembler" to simply take the responses of n divided by m with m being the number of assemblers. Then, finally, the master prompt can simply put together m solutions to get a final solution. This scales with the structure of a tree, and can grow infinitely for any input size.

IV. Methodology

In this section, we review our testing framework and how it evaluates the various degrees of teamwork our proposed algorithm provides.

A. Research Sub-questions

Within our overall research question, we propose three sub-questions that we aim to answer through our experimental study.

1) How does the number of workers used for collaboration affect the functional correctness of the final Python code generated by our algorithm, and how does it compare to zero-shot prompting?
2) How does the number of workers used for collaboration affect the complexity of the generated Python code?
3) How is the performance of the generated Python code affected by the number of workers used for collaboration?

B. Metrics

To answer these research questions, we measure the following metrics for each programming solution to each question for comparison.

1) Test cases passed: Each question has 10 corresponding test cases, and we take the raw number of test cases passed.
2) McCabe's cyclomatic complexity: Each solution will be scored using the *radon* python package.
3) Runtime: To determine performance, each solution will be scored on various cases using the python *timeit* package.

C. Dataset Generation

We used ChatGPT to generate the coding questions on which we tested the algorithm.

Using an iterative process, we asked ChatGPT to create Python scripts that consist of multiple functions and/or classes. We then prompted to have an empty function marked with "TODO" and instructions on the expected input and output of the function.

This was iterated many times until appropriate and relevant test questions were created. The hope is that all functions with the dataset interact with each other and fit within a theme, imitating real-life software engineering. The task at hand imitates adding a new feature to an existing codebase.

All of the functions that were meant for implementation were designed to be testable via a simple input-to-output mapping, and interacted with the other functions in the existing code base in varied manners.

Here is a brief description of each question and the software engineering skills that they test:

1) Question 1:
   - **Context:** A system consisting of various operations that are done on a graph and a class defining the structure of a graph.
   - **Task:** Implement the *find_articulation_points* function with the input of a graph object and the output of a list of vertices that are articulation points.
2) Question 2:
   - **Context:** A system consisting of various operations that relate to the creation and operations of a binary search tree
   - **Task:** Implement the *balance_binary_search_tree* function with the input of the root node of an unbalanced BST and output the node of the BST after it has been balanced.
3) Question 3:
   - **Context:** A series of functions that define mathematical operations in a two-dimensional space.
   - **Task:** Implement the *calculate_convex_hull* function with the input of a list of points (in the form of tuples) and output a list of tuples representing the vertices of the convex hull.
4) Question 4:
   - **Context:** Multiple classes that define trees and the nodes within them, and typical operations.
   - **Task:** Implement the *find_lowest_common_ancestor* function with the input of a tree object and two node values and output the value of the lowest common ancestor node.
5) Question 5:
   - **Context:** Multiple classes that define a library and the books within them.
   - **Task:** Implement the *calculate_total_value* function with the input of a library and output the total value of all the books within the collection
6) Question 6:
   - **Context:** Multiple classes which define patients, doctors, and their appointments.
   - **Task:** Implement the *cancel_appointment* function with the input of an appointment variable and ensure the appointment is removed from the schedule of the doctor and the patient.
7) Question 7:
   - **Context:** Multiple classes which define a graph and a data analysis tool.
   - **Task:** Implement the *find_all_paths* function with the input of a graph object, start node, and end node. Output a list of lists consisting of all paths from the start node to the end node.
8) Question 8:
   - **Context:** Multiple classes which build a transportation map consisting of cities and roads.
   - **Task:** Implement the *find_shortest_route* function with the input of a transportation map object, a start city object, and an end city object. Output the shortest route as a list of city objects.
9) Question 9:
   - **Context:** Multiple classes which define a warehouse, products, and a delivery vehicle.
   - **Task:** Implement the *optimize_vehicle_loading* function with the input of a warehouse object and a vehicle object. The output is a list of loaded products that maximizes the products in the vehicle.
10) Question 10:
   - **Context:** Multiple classes which define a polynomial and complex numbers.
   - **Task:** Implement the *find_polynomial_roots* function with the input of a polynomial object and the output is a list of complex number objects that represent the roots of the equation.

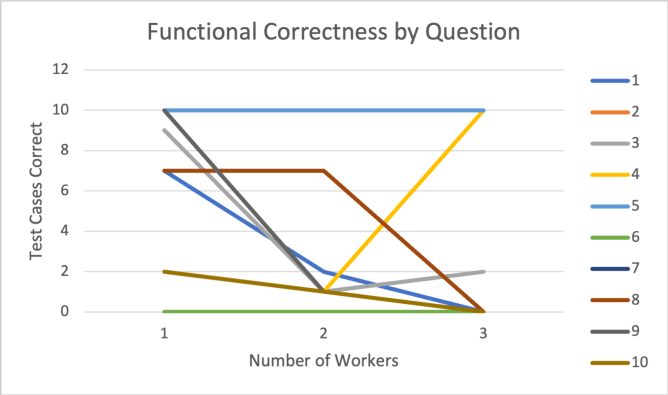

Fig. 2. Correctness Results.

D. Experimental Process

We strictly adhere to the experimental process to maintain the validity of the results.

All questions were stored in a Python file and each question (described above) was tested using a single worker (zero-shot prompting), then two workers, then three workers.

These tests were run independently, and all results, including intermediary worker prompt responses are stored in our repository.

This testing process was automated to generate all responses across all questions and all team sizes.

Then, the authors manually extracted the Python code from each final response and put it into the testing framework, which provided a standardized harness to test each question against all metrics.

TABLE I
FUNCTIONAL CORRECTNESS RESULTS

	Question Number										
Number of Workers	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**Sum**
1 (zero-shot)	7	10	9	10	10	0	7	7	10	2	**72**
2	2	1	1	1	10	0	7	7	1	1	**31**
3	0	10	2	10	10	0	0	0	0	0	**32**
Sum	**9**	**21**	**12**	**21**	**30**	**0**	**14**	**14**	**11**	**3**	**135**

TABLE II
ANOVA FOR FUNCTIONAL CORRECTNESS

	Df	Sum Sq	Mean Sq	F value	Pr(>F)
numworkers	2	121.6	60.78	5.450	0.0157 *
problems	8	168.0	21.00	1.883	0.1339
Residuals	16	178.4	11.15		

TABLE III
SIGNIFICANT TUKEY COMPARISONS FOR FUNCTIONAL CORRECTNESS

Factor	**Comp**	**Diff**	**Lower CI**[a]	**Upper CI**[a]	**p adj**
Numworkers	2-1	-4.55	-8.622	-0.493	0.0270
Numworkers	3-1	-4.44	-8.507	-0.382	0.0311

[a]95% Confidence Intervals are used

V. RESULTS

A. Functional Correctness

We present the results in this section and discuss statistical evaluations done for all metrics. This section will be finalized this weekend.

Table 1 presents the correct test cases (out of 10) for each question across multiple workers. As you can see, the basic zero-shot prompting seems to perform much better across most questions. This is visualized in Fig 2.

Specifically, we see that the zero-shot prompting passes the most test cases in every single question or is tied for the most test cases passed. The results between two workers and three workers do not seem to form any obvious patterns and simply depend on the problem.

To determine the statistical significance of this, we perform an analysis of variance test with blocking for the effects of workers and question on the number of test cases passed. It is important to point out that we omit question 6 from the statistical test due to an inaccuracy when generating the question (the task was impossible to accomplish within the single function due to the nature of the classes).

With this ANOVA displayed in Table II, we see that the number of workers, listed under the variable "numworkers" is statistically significant at the 0.1 alpha level, implying that the number of workers has a significant impact on the number of test cases passed.

This is in line with our observations that we made purely based on the table of results and the graph.

In order to determine the significant differences between the number of workers, we perform Tukey's Test for confidence intervals in our differences. The significant confidence intervals are listed in Table III.

As seen in the table, 95% confidence intervals are used, and the significant differences are between the multiple worker results and the single worker results. We see that the confidence interval's lower and upper bound are negative for "2-1" and "3-1".

Thus, the only statistically significant difference is that the single worker prompting (zero-shot) performs better than the two- and three-worker systems. There is no significant difference between the two multiple-worker systems.

B. Cyclomatic Complexity

Next, we evaluate the McCabe cyclomatic complexity [12] for each coding solution, whether it is correct or incorrect (as long as there are no runtime errors). This metric comes from representing a program as a graph and statically determining the number of paths that exist through the program.

The cyclomatic complexity of each coding solution can be seen in Table IV. Again, we see that the single-worker solution gives the simplest code for many questions. Since we already determined in the previous section that the single-worker solutions are statistically more functionally correct than the other solutions, this complexity could be attributed to a simpler solution that effectively solves the task.

These results are shown in Fig 3. for visualization. We see that the complexity tends to differ less for solutions to the same problem.

We also do the same statistical ANOVA on complexity, removing question 6 again for consistency. The results are displayed in Table V. As the table shows, the "numworkers" variable is insignificant. The "problems" variable is significant, but that is expected since each task varies in difficulty, which translates to complexity in the program.

Since this significance is expected, we do not further examine the results of the statistical tests. We can conclude that the number of workers does not impact the complexity of code written in this experiment.

From this metric, we cannot draw any conclusions from the dataset in this study. However, we do see interesting findings in select questions, such as question 4. Not only does the three-worker solution solve all test cases correctly, but it utilizes less complex code than the single-worker solution (which also solves all test cases correctly). This implies that there are some situations where the splitting of the work may prove to be beneficial, but more research is needed.

C. Performance

Our last metric is the performance of the code generated by our system when using variable amounts of workers. Similar

TABLE IV
CYCLOMATIC COMPLEXITY RESULTS

	Question Number										
Number of Workers	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**Average (rounded)**
1 (zero-shot)	3	1	3	6	2	1	1	8	1	2	**2.8**
2	3	1	4	8	2	1	1	9	3	9	**4.1**
3	3	2	7	3	2	1	5	9	4	8	**4.4**
Average (rounded)	**3**	**1.3**	**4.7**	**5.7**	**2**	**1**	**2.3**	**8.7**	**2.7**	**6.3**	**3.8**

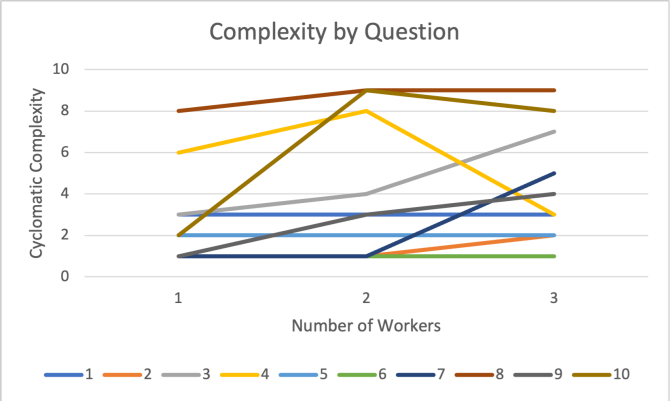

Fig. 3. Complexity Results.

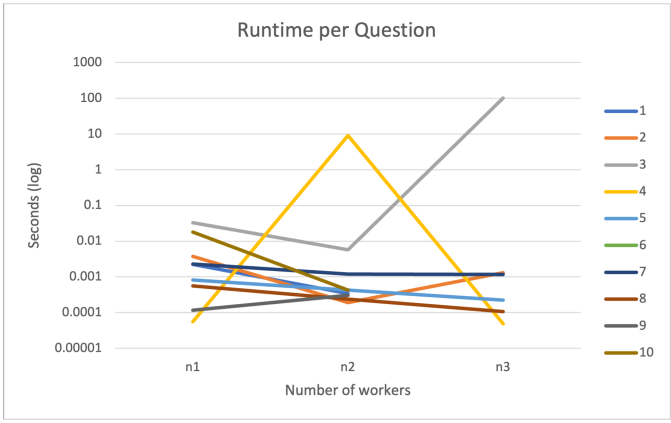

Fig. 4. Performance Results.

TABLE V
ANOVA FOR CYCLOMATIC COMPLEXITY

	Df	Sum Sq	Mean Sq	F value	Pr(>F)
numworkers	2	16.07	8.037	2.542	0.11001
problems	8	141.19	17.648	5.581	0.00174 **
Residuals	16	50.59	3.162		

to the complexity metric, we measure this metric whether or not the solution is correct.

However, we do impose a time-out limit, where a program is cut off after a designated period of time if it does not finish. This is to account for any infinite loops.

When measuring, we also do not base performance on a single run of the program and utilize sampling from the *timeit* library to ensure accuracy of our results.

The results of this evaluation are shown in Table VI. The entries represented by ”N/A” either timed out or had run-time errors for all test cases. There are two major outliers that skew the results of the table. Question 3 with three workers is a large outlier, and question 4 with two workers is also an outlier to a lesser degree. These two outliers make the averages look very different across the various worker amounts.

To determine if there are actual differences, we again perform a similar analysis of variance (ANOVA) with blocking. This can be seen in Table VII. As we can see, there are no significant differences across either variable (”numworkers” or ”problems”).

Thus, from our small sample size, we conclude that the code performance is not significantly impacted by the number of workers used in our algorithm.

VI. DISCUSSION

In this section, we discuss the results of the experiments and consider implications for the future of prompt engineering and the use of general-purpose LLMs.

One of the biggest takeaways we can determine from the results is that our teamwork architecture does not improve the software engineering capabilities of LLMs with more workers. This is specific to the scale that we are testing, and we theorize that at larger scales, where the context of a problem can no longer fit within the input token limit, our multi-worker algorithms will begin to perform better than zero-shot prompting.

Though we see statistically significant results for single-worker prompting, a specific look into the detailed questions gives us more insight as mentioned before. Out of the ten questions, there are only four where the single-worker outperforms both the two and the three-worker solutions. The biggest takeaway from this is that a larger, more diverse dataset is needed, since there is still a lot of variability within our results.

We also remind you of the results from question 4, which point to the potential of the multi-worker systems. Both the single-worker and three-worker solutions passed all test cases, but the three-worker solution achieved this with a complexity of 3 compared to a complexity of 6 from the single-worker solution. The less complex code also had better performance, which you can see in Table VI.

We theorize that in settings similar to Question 4, the aggregation of multiple (potentially non-optimal) solutions by the master prompt can result in higher-quality code. Our ten programming questions did not exemplify this area, but we

TABLE VI
DETAILED PERFORMANCE RESULTS WITH VARIOUS DECIMALS

	Question Number										
Workers	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**Average (rounded)**
1	0.00221	0.00374	0.0325	5.59e-05	0.000812	N/A	0.00227	0.000554	0.000116	0.0181	6.71e-03
2	0.000337	0.000189	0.00572	9.09	0.000423	N/A	0.00118	0.000239	0.000305	0.000428	1.01
3	N/A	0.00131	101	4.92e-05	0.000221	N/A	0.00118	0.000106	N/A	N/A	12.6
Average (rounded)	0.00127	0.00174	33.7	3.03	0.000485	N/A	0.00154	0.000300	0.000210	0.00927	4.56

TABLE VII
ANOVA FOR PERFORMANCE

	Df	Sum Sq	Mean Sq	F value	Pr(>F)
numworkers	2	1210	604.9	1.346	0.294
problems	8	2772	346.5	0.771	0.634
Residuals	12	5843	449.5		

hypothesize that it will be true at a larger scale and for various problems of different complexity.

We can also predict that there is a delicate balance in the number of workers that can be utilized for a single master prompt. We see that the three-worker solutions have an abnormally high variance in test cases passed, and in three problems (1, 9, 10), they receive N/A values for the performance tests. This implies that at some point, the output from each worker, even after it is cleaned, becomes too much for the master prompt to successfully merge all solutions and create an optimized solution without encountering errors and issues. This is similar to a software engineering team. If multiple solutions are proposed, it is difficult to combine all of them to obtain a single cohesive solution at the end. There are always trade-offs and too many options can become overwhelming.

We can also theorize that, if improved, the two-worker prompting can exhibit the same benefits of pair programming, where good code is written due to the synergy of two developers. This way, each worker can act as a developer with a slightly different perspective (due to the sliding window), and their solutions can be combined by the master prompt to perform better than the zero-shot prompting.

This will become more obvious as the length of the context increases and more complex (possibly multiple) tasks are assigned. The single-worker will struggle to capture larger codebases, and the multi-worker solutions will be able to shine. This is more practical in real-life development settings, and we could not capture this essence from our ten questions.

VII. LIMITATIONS AND FUTURE WORK

A. Limitations

One of this work's largest limitations is the dataset's size and generalizability. In practice, we would hope to have a dataset of real-world coding problems corresponding to software engineering tasks and new features to be added. Thus, this study can only be cited as a proof of concept and more work must be done to obtain conclusive results.

The questions within the dataset also pose a limitation to the study. Since they were generated by ChatGPT, they were limited in complexity and length, both of which are factors that we would like to improve on in the next iteration of this project. Additionally, since they were generated by the same LLM they were tested on, it is possible that their inclusion in the LLM's training data or knowledge set could have skewed results.

This study also used GPT-3.5 instead of the state-of-the-art GPT-4 or other similar models due to the financial limitations of this study. This unintentionally demonstrates a real-world issue that may be faced in developer environments - deciding between a cheaper, faster model, and a more expensive, but theoretically more accurate, model.

Also, these questions were formatted such that only a single function needed to be implemented. This is not very representative of real-world software engineering tasks and limits the extensibility of our work. The questions were also limited to basic data structures and algorithms, which have been proven by multiple studies [3], [9] to be relatively easy for large language models to solve, regardless of the context.

We also admit the limits of computational sustainability in this project. As the amount of workers scale up, the cost of model inference would also grow very quickly, leading to larger costs for developers. The trade-off of a potential improvement in ability would have to outweigh the cost in order to be feasible.

B. Future Work

Our future work contains many research directions based on the results of this study.

1) Longer Problems: The first extension of this study is to do the same with longer problems. As mentioned previously, longer problems should give us better results and highlight our proposed algorithm's benefits.

The largest challenge to this extension is to find a set of problems that can be evaluated in a controlled manner. The set of problems also needs to be large enough to test the capabilities of our prompting algorithm.

C. Human Written Tests

Re-running this test on human-written problems could provide more insight into an LLM's ability to reason about problems it has not directly seen before, and provide interesting results more reflective of a real development environment.

D. More Advanced Models

Future work could be to evaluate this test again using GPT-4, or even a mix of multiple models in the different roles.

1) Overlap Experimentation: The second extension of this study is to experiment with the overlap of content between the workers. In this study, it is mathematically obtained from the total length of the problem, but in order to truly understand the nature of the large language model collaboration, we must experiment with the amount of content the workers see which is overlapping.

This can help us determine better whether the master prompt is able to cohesively combine different points of view to create a final solution that effectively tackles the original task.

2) Multi-File Problems: The third extension of this study is to extend the scope of problems to multiple files. This is the most applicable to software engineering tasks in industry. By assigning workers across files, we could obtain a broader point of view and achieve larger-scale software engineering tasks with prompting.

The evaluation of this extension would be the most difficult since any generated code might affect the whole codebase, and the correctness of the generated code is somewhat unknown and might not be testable in a defined and contained environment.

3) Abstraction of Worker-Master to Tree Structure: The fourth extension of this study is to expand our algorithm to represent the relationship of multiple workers to multiple levels of masters. This becomes a tree structure where the tree leaves are workers and each level combines solutions iteratively until a final solution is outputted at the tree's root.

This is a natural extension of our current algorithm and can provide a better understanding of LLM cooperation at every level of the tree. It also helps us with larger-scale problems, overcoming the restrictions of attention in a single prompt.

VIII. Conclusion

We conclude this paper with lessons learned and key takeaways over the course of the study.

- **LLM collaboration can be effective in certain situations.** From the overall results of our study, we see that zero-shot prompting outperforms our algorithm in a statistically significant manner for functional correctness. However, after looking more closely, we see select situations where the team prompting proposed by our algorithm can perform better. This is important within our small sample size since individual cases serve as bearings for future intuition in exploration.
- **LLMs still struggle with the general combination task.** We see this in the increase in errors from the master prompts in multi-worker settings. This implies that LLMs still struggle with the task of combining multiple solutions to a single problem. This is somewhat expected since large language models are trained to predict the next token probabilistically. This area can be explored specifically in the future and holds importance in everyday use of LLMs.
- **One of the biggest challenges in modern-day LLMs is the task of effectively capturing all the most important parts of large contexts.** We have seen this task approached from many different angles such as the Longformer [1] and new file upload capabilities of the GPT-4 model. We approach this problem by simulating a team using prompting, but this will be a continuous area of improvement in the foreseeable future, as we strive towards AGI.

The Github repository containing the experiments and source code can be found at: https://github.com/davidgaofc/GPTeams.

Acknowledgment

I acknowledge the use of GPT models in many parts of the production of this study.

References

[1] Iz Beltagy, Matthew E Peters, and Arman Cohan. Longformer: The long-document transformer. *arXiv preprint arXiv:2004.05150*, 2020.

[2] Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, et al. Language models are few-shot learners. *Advances in neural information processing systems*, 33:1877–1901, 2020.

[3] 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.

[4] Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, et al. Codebert: A pre-trained model for programming and natural languages. *arXiv preprint arXiv:2002.08155*, 2020.

[5] Thorsten Händler. Balancing autonomy and alignment: A multi-dimensional taxonomy for autonomous llm-powered multi-agent architectures. *arXiv preprint arXiv:2310.03659*, 2023.

[6] Takeshi Kojima, Shixiang Shane Gu, Machel Reid, Yutaka Matsuo, and Yusuke Iwasawa. Large language models are zero-shot reasoners. *Advances in neural information processing systems*, 35:22199–22213, 2022.

[7] Aobo Kong, Shiwan Zhao, Hao Chen, Qicheng Li, Yong Qin, Ruiqi Sun, and Xin Zhou. Better zero-shot reasoning with role-play prompting. *arXiv preprint arXiv:2308.07702*, 2023.

[8] Zhenzhong Lan, Mingda Chen, Sebastian Goodman, Kevin Gimpel, Piyush Sharma, and Radu Soricut. Albert: A lite bert for self-supervised learning of language representations. *arXiv preprint arXiv:1909.11942*, 2019.

[9] 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.

[10] Yinhan Liu, Myle Ott, Naman Goyal, Jingfei Du, Mandar Joshi, Danqi Chen, Omer Levy, Mike Lewis, Luke Zettlemoyer, and Veselin Stoyanov. Roberta: A robustly optimized bert pretraining approach. *arXiv preprint arXiv:1907.11692*, 2019.

[11] Zijun Liu, Yanzhe Zhang, Peng Li, Yang Liu, and Diyi Yang. Dynamic llm-agent network: An llm-agent collaboration framework with agent team optimization. *arXiv preprint arXiv:2310.02170*, 2023.

[12] Thomas J McCabe. A complexity measure. *IEEE Transactions on software Engineering*, (4):308–320, 1976.

[13] 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.

[14] Yashar Talebirad and Amirhossein Nadiri. Multi-agent collaboration: Harnessing the power of intelligent llm agents. *arXiv preprint arXiv:2306.03314*, 2023.

[15] 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.

[16] Chaozheng Wang, Yuanhang Yang, Cuiyun Gao, Yun Peng, Hongyu Zhang, and Michael R Lyu. No more fine-tuning? an experimental evaluation of prompt tuning in code intelligence. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 382–394, 2022.

[17] Yue Wang, Weishi Wang, Shafiq Joty, and Steven CH Hoi. Codet5: Identifier-aware unified pre-trained encoder-decoder models for code understanding and generation. *arXiv preprint arXiv:2109.00859*, 2021.

[18] 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.

[19] 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.

[20] Jules White, Sam Hays, Quchen Fu, Jesse Spencer-Smith, and Douglas C Schmidt. Chatgpt prompt patterns for improving code quality, refactoring, requirements elicitation, and software design. *arXiv preprint arXiv:2303.07839*, 2023.

[21] Tongshuang Wu, Ellen Jiang, Aaron Donsbach, Jeff Gray, Alejandra Molina, Michael Terry, and Carrie J Cai. Promptchainer: Chaining large language model prompts through visual programming. In *CHI Conference on Human Factors in Computing Systems Extended Abstracts*, pages 1–10, 2022.

[22] Yue Wu, Xuan Tang, Tom M Mitchell, and Yuanzhi Li. Smartplay: A benchmark for llms as intelligent agents. *arXiv preprint arXiv:2310.01557*, 2023.

[23] Jintian Zhang, Xin Xu, and Shumin Deng. Exploring collaboration mechanisms for llm agents: A social psychology view. *arXiv preprint arXiv:2310.02124*, 2023.