Program Code Generation: Single LLMs vs. Multi-Agent Systems

Baskhad Idrisov
IU International University of Applied Sciences
Germany
baskhad.idrisov@gmail.com

Esther Eisenacher
IU International University of Applied Sciences
Germany
esther.eisenacher@iubh.de

Tim Schlippe
IU International University of Applied Sciences
Germany
tim.schlippe@iu.org

***Abstract*—This paper investigates the effectiveness of single large language models (LLMs) versus multi-agent systems in generating program code. Using AutoGen, we examine whether assigning detailed role descriptions enhances code quality in both setups. Key metrics such as *lines of code*, *cyclomatic complexity*, *runtime*, *memory usage*, and *maintainability index* are assessed. Python code generated by both systems is compared to human-written solutions across varying difficulty levels. Results show multi-agent systems hold potential for improving code quality, though the impact of detailed role descriptions warrants further exploration. Of the LLM-generated codes, 11 (46%) successfully solved the tasks, while 5 (21%) required minimal modifications, saving 2.9% to 17.9% of time compared to coding from scratch.**

***Index Terms*—large language models, LLMs, code generation, multi-agent systems, natural language processing, NLP**

I. Introduction

Generative AI has become increasingly integrated into daily life, particularly in text generation, where large language models (LLMs) replicate human-like conversations and provide instant support [1]–[4]. For example, ChatGPT[1], one of the most widely used platforms, quickly gained popularity, amassing over a million users within days of its release [5]. Beyond customer service and personal assistance, LLMs have also expanded into coding, transforming how programming tasks are approached [6], [7]. Integrated Development Environments (IDEs) and unit testing further enhance this by streamlining workflows and making coding more accessible to both experts and novices [8].

Despite these advancements, AI-powered coding solutions still face challenges, such as producing *correct*, *efficient*, and *maintainable* code, especially for less experienced users [9]–[12].

To address these limitations, first multi-agent frameworks for LLMs aim to tackle programming tasks by assigning specialized roles to different agents. AutoGen [13], for example, stands out for its flexibility, allowing developers to program agents using natural language or code across diverse domains. Other notable frameworks include MetaGPT [14], GameGPT [15], ChatDev [16], and MAGIS [17].

However, research has largely centered around metrics for code correctness, such as $pass@k$ [18]–[20] and comprehensive comparisons between AI- and human-generated code remain underexplored, particularly within multi-agent systems.

Consequently, in this paper we conduct a thorough evaluation of Python code generated by multi-agent systems of LLMs versus single LLMs. We assess *efficiency* and *maintainability* using metrics such as *lines of code*, *cyclomatic complexity*, *runtime*, *memory usage*, *Halstead complexity*, and *maintainability index*. Additionally, we compare LLM-generated code against professional human-written solutions, evaluating problems across three difficulty levels: *easy*, *medium*, and *hard*. Our analyzed systems leverage AutoGen [13] to iteratively improve code quality through execution and testing. We also examined whether providing a detailed role description to both single LLMs and multi-agent systems leads to improved code quality compared to merely assigning the role without additional detail.

In the next section, we will give an overview of related work. Section III will outline our experimental setup, followed by experiments and results in Section IV. Finally, Section V will conclude with future directions.

II. Related Work

Various studies on program code generation with LLMs have been carried out with single-agent setups. For example, Codex [19], based on GPT-3.0, was fine-tuned using GitHub code to generate Python code from natural language instructions. [19] tested Codex on their HumanEval dataset and achieved a pass@1 of 28.8%, outperforming GPT-3.0 and GPT-J. [21] evaluated Codex-based GitHub Copilot, showing it produced valid code 91.5% of the time, with a correctness rate of 28.7%. In a later study, [20] compared Copilot, CodeWhisperer, and ChatGPT, with ChatGPT generating the most *valid* (93.3%) and *correct* (65.2%) code. [18] introduced EvalPlus and HumanEval+, an expanded dataset to assess 26 AI models, where GPT-4 was the top performer (76.2% pass@1). Additionally, DeepMind's AlphaCode ranked in the top 54.3% of coding competition participants, and [22] found that GitHub Copilot's best performance was in Java with

This research was supported by the IU International University of Applied Sciences (*IU Incubator*) through internal initial funding for the period from October 2023 to September 2025.

[1]https://openai.com/chatgpt

57% *correctness* on LeetCode problems. [23] compare the *correctness*, *efficiency*, and *maintainability* of human- and LLM-generated code using metrics like *time, space complexity*, and *maintainability indices*. GitHub Copilot solved 50% of LeetCode problems, outperforming other models such as BingAI Chat and GPT-3.5, while 20.6% of LLM-generated codes were *correct*, and 8.7% required minimal modifications to be correct.

Several multi-agent frameworks for LLMs exist. A good overview is given in [24]. AutoGen [13] stands out due to its extensive customization capabilities, allowing developers to create agents that can be programmed through both natural language and coding. Code execution is possible amongst many other functions. This adaptability makes it suitable for a wide range of domains, from technical fields like programming and mathematics to consumer-oriented areas such as entertainment. Consequently, we used AutoGen for the implementation of our LLM-based multi-agent scenarios.

MetaGPT [14] employs a multi-agent system of LLM agents, each assigned distinct roles, to tackle Python programming tasks, outperforming prior systems like ChatDev. GameGPT [15], released in December 2023, similarly uses specialized agents for game development, emphasizing quality through critic feedback. ChatDev [16] operates as a virtual software company, using LLM agents with memory retention and self-reflection for code generation. MAGIS [17] focuses on resolving GitHub issues with superior collaboration.

Despite the success of MetaGPT, ChatDev, GameGPT, and MAGIS, these frameworks rely on predefined roles and the waterfall model, which limits flexibility. In contrast, we propose a more adaptable approach, defining LLM agent roles based on specific tasks, allowing for more flexibility in real-world applications and non-specialized teams.

III. Experimental Setup

In this section, we will first describe the experimental setup used to evaluate program code generation with multi-agent system and single LLM setups. Additionally, we will present the coding problems selected for evaluation and the use of GPT-4o mini as the primary model in our LLM setups.

A. Overview of our Analyzed LLM Setups

Our goal was to explore whether a multi-agent system of LLM agents, each assigned a role similar to that of a human developer team, outperforms single LLMs. Since both single LLMs and multi-agent systems can be given varying levels of detail about their roles, we also assessed whether providing a detailed role description improves code quality compared to simply assigning the role. A detailed description could guide the agents, but it might also limit their flexibility. Consequently, to evaluate code quality, we tested the following setups:

- Single LLM without role description
- Single LLM with role description
- Multi-agent system without role description
- Multi-agent system with role description

Figure 1 and Figure 2 demonstrate our investigated multi-agent systems and the single LLMs.

1) Multi-Agent System Setup: As shown in Figure 2, our *multi-agent system setup* consists of a collaborative multi-agent team where each agent has a clear responsibility and communicates with the others to improve and refine the solution:

- **Mathematician:** This agent is responsible for applying mathematical theories and providing mathematical models to help solve the problem. It communicates with other agents, such as the Algorithm Engineer, but does not write any code or design algorithms. It focuses purely on the mathematical aspect of the problem.
- **Algorithm Engineer:** The Algorithm Engineer designs the algorithm based on the mathematical model provided by the Mathematician. It does not write the code but creates a logical solution using established algorithm patterns and structures. If the algorithm is insufficient, it can request improvements from the Mathematician.
- **Python Developer:** This agent is responsible for writing the actual Python code. It implements the algorithm provided by the Algorithm Engineer and focuses on writing clean, efficient, and modular Python code. If the code needs to be improved, the Python Developer requests changes from the Algorithm Engineer.
- **Code Executor:** The Code Executor runs the Python code created by the Python Developer. It checks whether the output is correct by comparing it with the expected results. If the output is wrong, the Code Executor requests an updated mathematical model from the Mathematician to improve the code. If the output is correct, it terminates the process by sending the *final source code*.

In the *multi-agent system with role description*, each LLM agent is prompted to assume the roles of the *mathematician*, *algorithm engineer*, *developer*, and *code executor*, along with detailed descriptions of their respective tasks. In the *multi-agent system without role description*, each LLM agent is instructed to perform its designated tasks without any detailed guidance.

2) Single LLM Setup: As shown in Figure 2, in our *single LLM setup*, only one single LLM agent is responsible for the program code generation.

In the *single LLM with role description*, the single LLM agent is prompted to assume the roles of the *mathematician*, *algorithm engineer*, *developer*, and *code executor*, along with detailed descriptions of their respective tasks. In the *single LLM without role description*, the single LLM agent is instructed to perform its designated tasks without any detailed guidance.

B. Coding Problems

LeetCode is an online platform for improving coding skills, supporting multiple programming languages like Java, Python, and C++ [22], [25]. It is popular among job seekers and coding enthusiasts for technical interviews and coding competitions. The platform offers nearly 3,000 problems in six categories

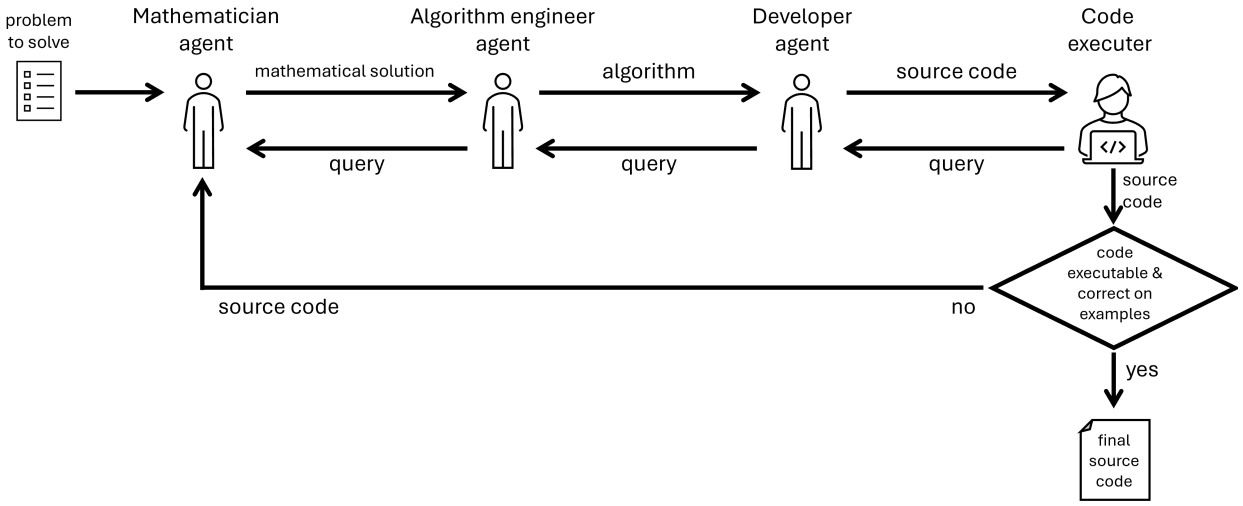

Fig. 1: Multi-agent system for program code generation to solve the LeetCode problems.

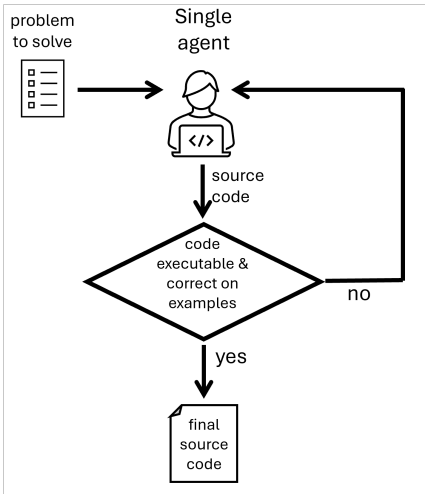

Fig. 2: Single-agent LLM for program code generation to solve the LeetCode problems.

(*algorithms*, *database*, *Pandas*, *JavaScript*, *Shell*, *concurrency*) and three difficulty levels (*easy*, *medium*, *hard*). Each coding problem on LeetCode includes a (1) task description, (2) input-output examples, and (3) constraints, such as variable ranges. After users submit their code, LeetCode reports runtime and memory usage, and if the code is not executable, an error message is displayed.

As shown in Table I, for our experiments we selected six LeetCode algorithm coding problems in Python: two *easy*, two *medium*, and two *hard*. The algorithm category was chosen to evaluate efficiency and maintainability, as poor solutions often result in high memory usage and difficult-to-understand code. To minimize overlap with AI training data, we used the latest coding problems from LeetCode. For comparison, we selected the highest-rated, human-generated Python code from LeetCode, written by expert programmers.

C. GPT-4o mini

For our experiments, we used GPT-4o, an optimized version of GPT-4, which processes up to 128k tokens more efficiently[2]. It handles multimodal input, including text and images, for tasks like image interpretation. Although no specific studies on its use in program code generation exist for GPT-4o mini, GPT-4 was the top performer in the program code generation evaluation of 26 popular LLMs conducted by [18].

IV. Experiments and Results

In this section, we will begin by analyzing which generated program codes are *correct*, meaning they effectively solve the corresponding coding problems. Next, we will assess the quality of the *correct* program codes using our evaluation criteria: *lines of code*, *cyclomatic complexity*, *time complexity*, *space complexity*, *runtime*, *memory usage*, and *maintainability index*. Finally, we will identify which of the *incorrect* program codes—due to their *maintainability* and proximity to *correct* program code—have the potential to be easily modified manually and then used quickly and without much effort to solve the corresponding problems.

A. Correct Solutions

Table II demonstrates which generated program codes for our six tasks were both *executable* and *correct* (indicated with ✓), i.e. did solve the coding problem. The entry "—" indicates program code which was *incorrect*, i.e. did not solve the coding problem.

While the program codes generated by all our 4 analyzed LLM setups are *executable*, we observe that only 11 (46%) of the 24 LLM-generated program codes are *correct*. Out of our 6 programming tasks, our *single LLM with role description* (*single (roles)*), our *multi-agent system without role description* (*multi (no roles)*) and our *multi-agent system with role description* (*multi (roles)*) were able to generate

[2]https://platform.openai.com/docs/models/gpt-4o-mini

TABLE I: Selected Coding Problems for our Evaluation

#	Coding Problem	Difficulty Level
1	Max Pair Sum in an Array	Easy
2	Faulty Keyboard	Easy
3	Minimum Absolute Difference Between Elements With Constraint	Medium
4	Double a Number Represented as a Linked List	Medium
5	Apply Operations to Maximize Score	Hard
6	Maximum Elegance of a K-Length Subsequence	Hard

TABLE II: Correct Solutions.

	easy		medium		hard		total
	#1	#2	#3	#4	#5	#6	
single (no roles)	✓	✓	—	—	—	—	2
single (roles)	✓	✓	✓	—	—	—	3
multi (no roles)	✓	✓	✓	—	—	—	3
multi (roles)	✓	✓	—	✓	—	—	3
human	✓	✓	✓	✓	✓	✓	6

3 *correct* program codes (50%) in total. Our *single LLM without role description* (*single (no roles)*) produced only 2 *correct* program codes (33%). Program codes written by human programmers (*human*) are consistently *correct*, regardless of problem difficulty.

All 4 LLM setups generated *correct* code for *easy* problems. However, none successfully solved the 2 *hard* coding problems. The *single (roles)*, *multi (no roles)*, and *multi (roles)* setups each solved 1 *medium* coding problem. Overall, the multi-agent systems show potential, but detailed role descriptions did not consistently enhance performance across problem difficulties.

B. Lines of Code

Table III shows the number of code lines in the *correct* 11 program codes which solve the coding problem plus the number of code lines in our human-written reference program codes (*human*).

TABLE III: Lines of Code.

	easy		medium		hard	
	#1	#2	#3	#4	#5	#6
single (no roles)	16	8	—	—	—	—
single (roles)	13	8	14	—	—	—
multi (no roles)	14	8	**13**	—	—	—
multi (roles)	24	8	—	41	—	—
human	**9**	**7**	**13**	**7**	**43**	**16**
best vs. *human* (Δ in %)	-31	-13	0	-486	—	—

None of our LLM setups was able to solve the coding problem with code that contains less *lines of code* than *human*. However, for task#3, the program code produced by *multi (no roles)* matched the number of *lines of code* in the *human* solution, demonstrating comparable code compactness in that specific case. Additionally, for task#2, all four LLM setups produced code that was only one line longer than the reference solution. However, the *correct* code generated by *multi (roles)* is nearly six times longer than the code produced by *human*.

C. Cyclomatic Complexity

Table IV presents the *cyclomatic complexity* of the 11 *correct* program codes generated by the LLM setups, alongside the *cyclomatic complexity* of the human-written reference codes (*human*). *Cyclomatic complexity* measures the number of independent paths through the code, with higher values indicating lower code quality.

TABLE IV: Cyclomatic Complexity.

	easy		medium		hard	
	#1	#2	#3	#4	#5	#6
single (no roles)	6	3	—	—	—	—
single (roles)	6	3	6	—	—	—
multi (no roles)	6	3	6	—	—	—
multi (roles)	10	3	—	10	—	—
human	**3**	**2**	**5**	**4**	**42**	**16**
best vs. *human* (Δ in %)	-50	-50	-20	-150	—	—

The results show that the *cyclomatic complexity* is consistent across all LLM setups, except for *multi (roles)* in task #1. None of the LLM setups generated code with lower *cyclomatic complexity* than the *human* reference.

D. Time Complexity

Table V illustrates the *time complexity* in the 11 *correct* program codes plus the *time complexity* in our human-written reference program codes (*human*). The *time complexity* quantifies the upper bound of time needed by an algorithm as a function of the input [26]. The lower the order of the function, the better the *complexity*.

TABLE V: Time Complexity.

	easy		medium		hard	
	#1	#2	#3	#4	#5	#6
single (no roles)	**nlogn**	n^2	—	—	—	—
single (roles)	**nlogn**	n^2	**n**	—	—	—
multi (no roles)	**nlogn**	n^2	**n**	—	—	—
multi (roles)	**nlogn**	n^2	—	**n**	—	—
human	nm	**kn**	nlogn	**n**	**nlogn**	**nlogn**
best vs. *human* (*h*)	LLM	h	LLM	—	h	h

The results indicate that *time complexity* remains consistent across all LLM setups. Among the 11 *correct* program codes, 6 codes (36%) from an LLM setup exhibit lower *time complexity* than the *human* reference. In 2 cases (18%), the *human*-written code outperforms the LLM-generated code in terms of *time complexity*. One LLM-generated code (9%)—*multi (roles)*—match the *human* code in *time complexity*.

E. Space Complexity

Table VI presents the *space complexity* of the 11 *correct* LLM-generated program codes, alongside the *space complexity* of the human-written reference codes (*human*). Like *time complexity*, *space complexity* measures the upper bound of memory required by an algorithm as a function of input size [27]. Lower-order functions indicate better *complexity*.

TABLE VI: Space Complexity.

	easy		medium		hard	
	#1	**#2**	**#3**	**#4**	**#5**	**#6**
single (no roles)	**n**	**n**	—	—	—	—
single (roles)	**n**	**n**	**n**	—	—	—
multi (no roles)	**n**	**n**	**n**	—	—	—
multi (roles)	**n**	**n**	—	**n**	—	—
human	**n**	**n**	**n**	**n**	**n**	**n**
best vs. *human* (*h*)	—	—	—	h	h	h

The results for *time complexity* indicate that *cyclomatic complexity* remains consistent across all LLM setups. All LLM setups perform equally, each maintaining *O(n)*. This consistency is also observed in the *human* reference code.

F. Runtime

Table VII demonstrates the *runtime* of the 11 *correct* program codes plus the *runtime* of our human-written reference program codes (*human*) on LeetCode in milliseconds. The lower the *runtime* of a program code, the better.

TABLE VII: Runtime.

	easy		medium		hard	
	#1	**#2**	**#3**	**#4**	**#5**	**#6**
single (no roles)	118	36	—	—	—	—
single (roles)	**107**	41	984	—	—	—
multi (no roles)	123	**33**	**924**	—	—	—
multi (roles)	120	45	—	309	—	—
human	114	47	961	**235**	**5k**	**1k**
best vs. *human* (Δ in %)	+6	+30	+4	-21	—	—

Out of the 11 *correct* program codes, 6 cases (55%) feature an LLM setup that produced code with a shorter *runtime* than the *human* reference. In contrast, 5 LLM-generated codes (45%) are outperformed by *human* in this metric. Comparing the LLM setups demonstrates that *multi (no roles)* delivers the best performance in 2 tasks (#2 and #3), while *single (roles)* excels in 1 task (#1). However, *single (no roles)* and *multi (roles)* fail to surpass any of the other LLM setups.

G. Memory Usage

Table VIII lists the *memory usage* of the 11 *correct* program codes plus the *memory usage* of our human-written reference program codes (*human*) on LeetCode in megabytes. The lower the *memory usage* of a program code, the better.

The results indicate that *memory usage* is consistent across the program codes generated by all LLM setups and the *human* reference. None of the LLM-generated codes exhibits lower *memory usage* than the human-written code. In terms of this metric, no particular LLM setup demonstrates superior or inferior performance.

	easy		medium		hard	
	#1	**#2**	**#3**	**#4**	**#5**	**#6**
single (no roles)	**17**	**17**	—	—	—	—
single (roles)	**17**	**17**	**32**	—	—	—
multi (no roles)	**17**	**17**	**32**	—	—	—
multi (roles)	**17**	**17**	—	**21**	—	—
human	**17**	**17**	**32**	**21**	**40**	**53**
best vs. *human* (Δ in %)	0	0	0	0	—	—

TABLE VIII: Memory Usage.

H. Maintainability Index

Table IX demonstrates the *maintainability index* of the 11 *correct* program codes along with the *maintainability index* of our human-written reference program codes (*human*). A higher *maintainability index* indicates better code quality.

TABLE IX: Maintainability Index.

	easy		medium		hard	
	#1	**#2**	**#3**	**#4**	**#5**	**#6**
single (no roles)	54	**64**	—	—	—	—
single (roles)	56	**64**	55	—	—	—
multi (no roles)	55	63	**56**	—	—	—
multi (roles)	48	63	—	42	—	—
human	**61**	**64**	**56**	**63**	**39**	**54**
best vs. *human* (Δ in %)	-8	—	0	-33	—	—

Examining the results of the *maintainability index*, we see that while *human*-generated code consistently performs best, both *single (no roles)* and *single (roles)* reach a comparable score of 64 for task #2. *multi (no roles)* matches the human result for task #3 (56). However, *multi (roles)* falls short in *maintainability* across all tasks.

I. Potential of Incorrect LLM-Generated Program Code

After analyzing the 11 *correct* program codes, our goal was to evaluate which of the 13 *incorrect* program codes have the potential to be easily modified manually and then used quickly and without much effort to solve the corresponding coding problems. To ensure a fair comparison that is independent of programmers' experience, we report the *TTC* using Halstead's estimates of the *implementation time*, which is only dependent on the operators and operands in the program code—not on the expertise of a programmer. Consequently, to estimate the *TTC* in seconds, we developed the following formula:

$$TTC = |T_{correct} - T_{incorrect}| + T_{maintain}$$

where $T_{correct}$ is Halstead's *implementation time* [28, pp. 57–59] of the *correct* program code in seconds and $T_{incorrect}$ is Halstead's *implementation time* [28, pp. 57–59] of the *incorrect* program code in seconds. The absolute difference is employed to account for scenarios where $T_{correct}$ may be lower than $T_{incorrect}$ due to the necessity of removing parts of the program code to achieve the *correct* version. As Healstead's *implementation time* only addresses the time for the effort of implementing and understanding the program based on the operators and operands but not time to maintain the code—which is crucial for correcting program code—we

TABLE X: Potential of *Incorrect* LLM-generated Program Code.

	#	MI	$T_{incorrect}$ | $T_{correct}$	Estimated time to program in seconds TTC	Δ $T_{correct}$–TTC (%)
multi (no roles)	3	55	1,054 | 881	1,046	–15.74
single (no roles)	4	55	669 | 685	581	**+17.93**
single (roles)	4	55	646 | 663	539	**+23.11**
multi (no roles)	4	55	699 | 714	585	**+22.04**
single (no roles)	5	48	1,694 | 6,270	6,384	–1.79
single (roles)	5	54	1,296 | 6,270	6,082	**+3.10**
multi (no roles)	5	54	1,245 | 6,270	6,092	**+2.93**
multi (roles)	5	44	2,885 | 6,270	7,129	–12.05
single (no roles)	6	49	2,204 | 1,341	3,179	–57.82
single (roles)	6	46	3,320 | 3,425	3,954	–13.38
multi (no roles)	6	48	1,685 | 1,341	2,158	–37.85
multi (roles)	6	52	1,849 | 1,341	2,244	–40.23

additionally calculated the time to maintain the program with the help of the *maintainability index MI*. The *MI* is based on *lines of code*, *cyclomatic complexity* and Halstead's *volume*. $T_{maintain}$ in seconds is estimated using the following formula:

$$T_{maintain} = \frac{T_{incorrect}}{MI/100} - T_{incorrect}$$

where MI represents the *maintainability index*, which ranges from 0 to 100, as referenced in [29]. To normalize MI to a scale from 0 to 1, it is divided by 100. This adjustment allows $T_{incorrect}$ to be multiplied by a factor that increases as the maintainability of the program code decreases.

Table X demonstrates the *MI*, $T_{incorrect}$, $T_{correct}$, *TTC* as well as the relative difference between $T_{correct}$ and TTC (Δ $T_{correct}$–TTC (%)) for our 13 *incorrect* program codes and their corresponding *correct* program codes. We observe that for 5 program codes *TTC* < $T_{correct}$, i.e. the *time to correct* (*TTC*) the *incorrect* program code takes less time than the *implementation time* of the *correct* program code $T_{correct}$.

For these 5 program codes, time savings range from 2.93% to 23.11% when correcting the LLM-generated program code instead of developing it from scratch. Specifically, for task #4, the code produced by *single (no roles)* has the lowest *TTC*, making it 23% faster than implementing the correct code from scratch. In task #5, the code generated by *single (roles)* also achieves the lowest *TTC*, being 3% faster than creating the correct code from the ground up. For the other tasks and LLM setups, $T_{correct}$ is lower than *TTC*, indicating that it makes more sense to manually implement the *correct* program from scratch.

V. Conclusion and Future Work

LLMs have significantly transformed coding practices, enabling both experts and novices to engage more efficiently in programming tasks. Despite this potential, challenges remain regarding the production of *correct*, *efficient*, and *maintainable* code. We have investigated the performance of single LLMs compared to multi-agent systems of LLMs in generating Python code, focusing on various metrics such as *lines of code*, *cyclomatic complexity*, *runtime*, *memory usage*, *Halstead complexity*, and *maintainability index*.

We evaluated six LeetCode algorithm problems across three difficulty levels—*easy*, *medium*, and *hard*—comparing LLM-generated solutions against human-written code. The results revealed that both the *single LLM with role descriptions* and the *multi-agent systems* demonstrated a 50% success rate in producing *correct* code, while the *single LLM without role descriptions* achieved only a 33% success rate. All setups solved the *easy* problems. But none successfully tackled the *hard* problems, highlighting limitations in the LLMs' capabilities.

Our results show that while multi-agent systems hold promise, providing detailed *role descriptions* did not consistently enhance performance. In terms of *efficiency*, *multi-agent setups* produced comparable or better solutions for some tasks, particularly in *runtime*. However, none of the LLM-generated codes managed to outperform the *human*-written counterparts in metrics such as *memory usage* and *maintainability index*.

But manually correcting *incorrect* code generated by LLM setups can offer significant time savings over coding from scratch: For instance, one *incorrect* code produced by the *single LLM without role descriptions* was close enough to the *correct* solution that manual correction resulted in a 23% time reduction. This emphasizes the potential of LLMs to contribute effectively in real-world coding scenarios, especially when minor corrections are required.

Ultimately, our work highlights the importance of further refining LLM frameworks and exploring the balance between predefined roles and flexibility to enhance code *quality* and *maintainability* in software development.

References

[1] C. Pelau, D.-C. Dabija, and I. Ene, "What Makes an AI Device Human-like? The Role of Interaction Quality, Empathy and Perceived Psychological Anthropomorphic Characteristics in the Acceptance of Artificial Intelligence in the Service Industry," *Computers in Human Behavior*, vol. 122, p. 106855, 2021. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0747563221001783

[2] M. Dibitonto, K. Leszczynska, F. Tazzi, and C. M. Medaglia, "Chatbot in a Campus Environment: Design of LiSA, a Virtual Assistant to Help Students in Their University Life," in *Human-Computer Interaction. Interaction Technologies*, M. Kurosu, Ed. Cham: Springer International Publishing, 2018, pp. 103–116.

[3] D. Arteaga, J. J. Arenas, F. Paz, M. Tupia, and M. Bruzza, "Design of Information System Architecture for the Recommendation of Tourist Sites in the City of Manta, Ecuador through a Chatbot," *2019 14th Iberian Conference on Information Systems*

and Technologies (CISTI), pp. 1–6, 2019. [Online]. Available: https://api.semanticscholar.org/CorpusID:198964734

[4] D. Adiwardana, M.-T. Luong, D. R. So, J. Hall, N. Fiedel, R. Thoppilan, Z. Yang, A. Kulshreshtha, G. Nemade, Y. Lu, and Q. V. Le, "Towards a Human-like Open-Domain Chatbot," 2020.

[5] V. Taecharungroj, ""What Can ChatGPT Do?" Analyzing Early Reactions to the Innovative AI Chatbot on Twitter," *Big Data and Cognitive Computing*, vol. 7, no. 1, 2023. [Online]. Available: https://www.mdpi.com/2504-2289/7/1/35

[6] E. Loh, "ChatGPT and Generative AI Chatbots: Challenges and Opportunities for Science, Medicine and Medical Leaders," *BMJ Leader*, 2023. [Online]. Available: https://bmjleader.bmj.com/content/early/2023/05/02/leader-2023-000797

[7] E. Mollick, "ChatGPT Is a Tipping Point for AI," *Harvard Business Review*, 2022, accessed: 11-10-2023.

[8] Z. Alizadehsani, E. G. Gomez, H. Ghaemi, S. R. González, J. Jordan, A. Fernández, and B. Pérez-Lancho, "Modern Integrated Development Environment (IDEs)," in *Sustainable Smart Cities and Territories*, J. M. Corchado and S. Trabelsi, Eds. Cham: Springer International Publishing, 2022, pp. 274–288.

[9] A. Kaur, A. Jadhav, M. Kaur, and F. Akter, "Evolution of Software Development Effort and Cost Estimation Techniques: Five Decades Study Using Automated Text Mining Approach," *Mathematical Problems in Engineering*, vol. 2022, p. 5782587, 2022.

[10] I. Bluemke and A. Malanowska, "Software Testing Effort Estimation and Related Problems: A Systematic Literature Review," *ACM Comput. Surv.*, vol. 54, no. 3, apr 2021. [Online]. Available: https://doi.org/10.1145/3442694

[11] S. A. Butt, S. Misra, G. Piñeres-Espitia, P. Ariza-Colpas, and M. M. Sharma, "A Cost Estimating Method for Agile Software Development," in *Computational Science and Its Applications – ICCSA 2021*, O. Gervasi, B. Murgante, S. Misra, C. Garau, I. Blečić, D. Taniar, B. O. Apduhan, A. M. A. C. Rocha, E. Tarantino, and C. M. Torre, Eds. Cham: Springer International Publishing, 2021, pp. 231–245.

[12] B. Zhang, P. Liang, X. Zhou, A. Ahmad, and M. Waseem, "Practices and Challenges of Using GitHub Copilot: An Empirical Study," in *International Conferences on Software Engineering and Knowledge Engineering*. KSI Research Inc., jul 2023. [Online]. Available: https://doi.org/10.18293%2Fseke2023-077

[13] Q. Wu, G. Bansal, J. Zhang, Y. Wu, B. Li, E. E. Zhu, L. Jiang, X. Zhang, S. Zhang, A. Awadallah, R. W. White, D. Burger, and C. Wang, "AutoGen: Enabling Next-Gen LLM Applications via Multi-Agent Conversation," in *COLM 2024*, August 2024.

[14] S. Hong, M. Zhuge, J. Chen, X. Zheng, Y. Cheng, J. Wang, C. Zhang, Z. Wang, S. K. S. Yau, Z. Lin, L. Zhou, C. Ran, L. Xiao, C. Wu, and J. Schmidhuber, "MetaGPT: Meta Programming for A Multi-Agent Collaborative Framework," in *The Twelfth International Conference on Learning Representations*, 2024. [Online]. Available: https://openreview.net/forum?id=VtmBAGCN7o

[15] D. Chen, H. Wang, Y. Huo, Y. Li, and H. Zhang, "GameGPT: Multi-agent Collaborative Framework for Game Development," 2023. [Online]. Available: https://arxiv.org/abs/2310.08067

[16] C. Qian, W. Liu, H. Liu, N. Chen, Y. Dang, J. Li, C. Yang, W. Chen, Y. Su, X. Cong, J. Xu, D. Li, Z. Liu, and M. Sun, "ChatDev: Communicative Agents for Software Development," in *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, L.-W. Ku, A. Martins, and V. Srikumar, Eds. Bangkok, Thailand: Association for Computational Linguistics, Aug. 2024, pp. 15 174–15 186. [Online]. Available: https://aclanthology.org/2024.acl-long.810

[17] W. Tao, Y. Zhou, Y. Wang, W. Zhang, H. Zhang, and Y. Cheng, "MAGIS: LLM-Based Multi-Agent Framework for GitHub Issue Resolution," 2024. [Online]. Available: https://arxiv.org/abs/2403.17927

[18] J. Liu, C. S. Xia, Y. Wang, and L. Zhang, "Is Your Code Generated by ChatGPT Really Correct? Rigorous Evaluation of Large Language Models for Code Generation," 2023.

[19] M. Chen, J. Tworek, H. Jun, Q. Yuan, H. P. de Oliveira Pinto, J. Kaplan, H. Edwards, Y. Burda, N. Joseph, G. Brockman, A. Ray, R. Puri, G. Krueger, M. Petrov, H. Khlaaf, G. Sastry, P. Mishkin, B. Chan, S. Gray, N. Ryder, M. Pavlov, A. Power, L. Kaiser, M. Bavarian, C. Winter, P. Tillet, F. P. Such, D. Cummings, M. Plappert, F. Chantzis, E. Barnes, A. Herbert-Voss, W. H. Guss, A. Nichol, A. Paino, N. Tezak, J. Tang, I. Babuschkin, S. Balaji, S. Jain, W. Saunders, C. Hesse, A. N. Carr, J. Leike, J. Achiam, V. Misra, E. Morikawa, A. Radford, M. Knight, M. Brundage, M. Murati, K. Mayer, P. Welinder, B. McGrew, D. Amodei, S. McCandlish, I. Sutskever, and W. Zaremba, "Evaluating Large Language Models Trained on Code," 2021.

[20] B. Yetiştiren, I. Özsoy, M. Ayerdem, and E. Tüzün, "Evaluating the Code Quality of AI-Assisted Code Generation Tools: An Empirical Study on GitHub Copilot, Amazon CodeWhisperer, and ChatGPT," 2023.

[21] B. Yetistiren, I. Ozsoy, and E. Tuzun, "Assessing the Quality of GitHub Copilot's Code Generation," *Proceedings of the 18th International Conference on Predictive Models and Data Analytics in Software Engineering*, 2022. [Online]. Available: https://api.semanticscholar.org/CorpusID:253421992

[22] N. Nguyen and S. Nadi, "An Empirical Evaluation of GitHub Copilot's Code Suggestions," in *2022 IEEE/ACM 19th International Conference on Mining Software Repositories (MSR)*, 2022, pp. 1–5.

[23] B. Idrisov and T. Schlippe, "Program Code Generation with Generative AIs," *Algorithms*, vol. 17, no. 2, 2024. [Online]. Available: https://www.mdpi.com/1999-4893/17/2/62

[24] T. Guo, X. Chen, Y. Wang, R. Chang, S. Pei, N. V. Chawla, O. Wiest, and X. Zhang, "Large Language Model based Multi-Agents: A Survey of Progress and Challenges," 2024. [Online]. Available: https://arxiv.org/abs/2402.01680

[25] LeetCode, "LeetCode QuickStart Guide," https://support.leetcode.com/hc/en-us/articles/360012067053-LeetCode-QuickStart-Guide, 2023, accessed: 16-10-2024.

[26] T. Cormen, C. Leiserson, R. Rivest, and C. Stein, *Introduction to Algorithms*, 4th ed. MIT Press, 2022.

[27] Baeldung, "Understanding Space Complexity," *Baeldung on Computer Science*, 2021. [Online]. Available: https://www.baeldung.com/cs/space-complexity

[28] M. H. Halstead, *Elements of Software Science*. Elsevier, 1977.

[29] Microsoft, "Visual Studio—Maintainability Index," https://docs.microsoft.com/en-us/visualstudio/code-quality/code-metrics-maintainability-index-range-and-meaning, 2021, 16-10-2024.