

# Are LLMs the **END** of Programming?

Ilan Barr, Nancy Patel, Yifan Yuan

## Abstract

This study evaluates the performance of Large Language Models (LLMs) against human coders in solving programming challenges across three programming languages and problem categories using a dataset from LeetCode. We compare five LLMs—GPT-3.5, GPT-4, Claude Haiku, Claude Sonnet, and Claude Opus—focusing on solution acceptance rates, runtime efficiency, memory usage, and error analysis. Preliminary findings show that LLMs generally surpass human coders, particularly after adjusting human performance metrics to account for biases in data collection. The analysis also reveals that problem characteristics such as complexity and description length significantly influence LLM performance. While LLMs demonstrate substantial potential in enhancing programming efficiency, our results suggest a more complementary role alongside human expertise, particularly in complex problem-solving scenarios.

## 1. INTRODUCTION

The rapid evolution of Large Language Models (LLMs) has brought an onslaught of automation capabilities, from audio transcription, text parsing, and generative features. With such capabilities, these LLMs are rapidly integrating into multiple professions, such as providing sports commentary in real-time, summarizing legal documents, and even creating music. Our project is interested in the integration of LLMs in a different professional landscape: software engineering. Through our study, we aim to delve into the relative strengths and weaknesses of LLMs versus humans and examine the feasibility of LLMs working in conjunction with human software engineers and potentially even substituting them altogether in the future.

### Related Work

This March, the entrance of DevinAI shocked the software engineering world [1]. The team behind DevinAI claims it is the “first fully autonomous AI software engineer”, capable of building apps end-to-end and autonomously finding and fixing bugs. However, metrics released for DevinAI conclude that it can only solve approximately 14% of the issues it is given. Thus, some critics believe that human-AI pair programming will be the more realistic future of software engineering [2].

“Is GitHub copilot a substitute for human pair-programming?” investigates the productivity and quality of human-human pair programming versus human-AI programming using GitHub Copilot. Imai finds that although the human-AI pairing generated more lines of code than the human-human pairing in the same amount of time, the code quality was lower [3]. Ma et. al. observe in their paper “Is AI the better programming partner?” that pair programming (both human-human and human-AI) is less productive when there are mismatched expertise levels between the two parties. Thus, the authors hypothesize that, for human-AI pairing to be effective, AIs should be able to adapt to different expertise [4].

To determine the relative strengths and weaknesses of the popular AI Chat-GPT, Nascimento et. al assessed LLM-generated code versus software-engineer-generated code on popular technical coding interview site, Leetcode. The study finds that GPT-4 outperforms novice programmers, solely when tackling coding problems of easy and medium difficulty. However, the paper did not have enough evidence to assert that GPT-4 surpasses experienced programmers in coding problems of any difficulty (easy, medium, or hard). For their study, Nascimento et. al focus on 1 LLM (GPT-4), 1 programming language (C++), and 1 question category (Array) [5].

We seek to expand on the empirical study conducted by Nascimento et. al. by testing 5 LLMs, 3 programming languages, and 3 question categories. By exploring how human-authored versus AI-generated solutions for specific programming paradigms compare against each other, we aim to continue developing the groundwork for more informed human-in-the-loop approaches to software engineering.

## 2. PRELIMINARIES

Before describing our experiments, we briefly discuss the setting in which we collected all of our human- and LLM-generated code. Our collection process took place between April 1-25, 2024, before GPT-4 rolled out its cross-chat memory control [6]. As a result, we expect that each LLM-generated code snippet we collected is independent of the other LLM-generated code snippets (i.e. each LLM response is unaffected by other responses in separate chats).

### 3. METHODOLOGY

#### 3.1 Question Collection

Our experiments begin by developing a dataset of 9 questions from LeetCode. We chose LeetCode for our dataset over other coding benchmarks, like SWE-Bench or HumanEval, because the latter datasets do not offer human-completed solutions (with total acceptance rates) for comparison [10, 11]. For our dataset, we picked three questions each from three of the most popular programming categories, Dynamic Programming (#5, #10, #53), Matrix (#542, #1210, #1970), and Sorting (#217, #630, #2402). Note that many of these questions were “multi-tagged.” For example, #1970 is assigned to the Matrix category, but also belongs to the Array, Depth First Search (DFS), Breadth First Search (BFS), Hash Map, and Union Find categories. In total, there is 1 easy-difficulty question, 3 medium-difficulty questions, and 5 hard-difficulty questions. Because Nascimento et. al. found that GPT-4 only outperformed novice programmers on easy and medium-difficulty questions, we are interested in delving deeper into LLM performance on hard-difficulty coding problems. As a result, our question selection is skewed toward harder problems.

#### 3.2 Programming Language Selection

For each question, we collect 3 human submissions and 3 AI-generated codes (for each of our 5 LLMs) in each of the following programming languages: Python, C++, and Go. We chose these languages because we are interested in seeing if LLMs can consistently replicate their performance on languages with different structures, memory capabilities, and programming paradigms. Python is the most popular of the three languages because of its simplicity: it is dynamically typed, has automatic garbage collection, and follows an object-oriented paradigm. C++ is second-most popular: it is object-oriented like Python but does not support automatic garbage collection. Lastly, Go is the least popular: it is statically typed like C++, but unlike C++ and Python, it is a procedural programming language [8].

#### 3.3 Large Language Model Selection

We chose to use the following 5 LLMs: GPT-3.5, GPT-4, Claude Haiku, Claude Sonnet, and Claude Opus. This choice was inspired by the benchmark comparisons released by DevinAI which compares DevinAI against GPT-3.5, GPT-4, and Claude 2 [1]. According to Anthropic (the team behind Claude 2 and 3), the LLMs are ranked as following for HumanEval: Opus (84.9%), Haiku (75.9%), Sonnet (73.0%), GPT-4 (67.0%), and GPT-3.5 (48.1%) [7]. We expect to see similar results in section 4.

#### 3.4 Code Collection

**3.4.1 Human-Generated Solutions** On LeetCode, there is a “Discussion” section accompanying each problem, where users post their solutions. For our study, we used this section to collect human-generated code using the process outlined as follows. First, we filter the “Discussion” section based on the programming language of interest (Python,

C++, or Go). It is important to note that posts are sorted by rating and views, with the highest-rated and most-viewed posts displayed first. Thus, to minimize any sampling bias, we randomly pick each submission using a random number generator.

**3.4.2 AI-Generated Solutions** To ensure a robust evaluation of artificial intelligence models, our methodology involved a carefully designed setup for collecting AI-generated solutions. The process is outlined as follows:

1. **Question Modification:** Each problem selected from LeetCode was initially modified to create a novel challenge not directly encountered during the training of the language models. This modification was aimed at assessing the genuine problem-solving abilities of the models under conditions that prevented any benefit from previous exposure to similar problems. If a modification led to all language models (LLMs) consistently failing to pass any tests—indicating an excessively stringent alteration—the original version of the problem was used to maintain a balance of challenge and feasibility.
2. **Isolated Problem-Solving Sessions:** Each LLM was tasked with solving these problems in a separate, controlled environment. This isolation ensured that the performance data from one model did not influence another, thereby maintaining the integrity of the experimental conditions.
3. **Uniform Problem Presentation:** The problems were presented to each LLM using a standardized prompt format. As demonstrated in Figure 1, the prompt included a clear articulation of the problem requirements, constraints, and expected outputs, along with a code template for implementation. The standardized prompt helped to ensure that all models were provided with the same level of detail and clarity, minimizing variability in understanding or interpretation that could affect the performance outcomes.

#### 3.5 Research Metrics

In their paper “Perfection Not Required?” Weisz et. al ask software engineers under what conditions they would accept LLM-generated code [9]. The study finds that software engineers tend to accept code through verification, not explanation. In other words, functional metrics were more important to these engineers than non-functional metrics (i.e. explanation or readability).

Thus, we chose to measure human and AI-generated solutions using the following functional metrics: number of passing tests, runtime performance, and memory usage. To test each of these metrics, we take each solution and execute it within a simulated LeetCode environment. Once submitted, the site would provide three pieces of information: the number of test cases passed, the runtime (and the percentage of other submissions it beat), and the memory usage (and the percentage of other submissions it beat).

In some instances, the executed code did not provide any of these metrics because of errors like “Compile Error”,

```

Please read the following algorithm problem. Understand its requirements, examples, constraints,
and the expected input/output formats. Then, develop a solution in C++, explaining your thought
process and any assumptions you make. If possible, also discuss the time and space complexity of
your solution.

Requirement: Given an m x n binary matrix mat, return the distance of the nearest 0 for each cell.

The distance between two adjacent cells is 1.

Example 1:
Input: mat = [[0,0,0],[0,1,0],[0,0,0]]
Output: [[0,0,0],[0,1,0],[0,0,0]]

Example 2:
Input: mat = [[0,0,0],[0,1,0],[1,1,1]]
Output: [[0,0,0],[0,1,0],[1,2,1]]

Fill this function below:

class Solution {
public:
    vector<vector<int>> updateMatrix(vector<vector<int>>& mat) {
    }
};

```

Figure 1: Example Prompt for AI LeetCode Task

"Runtime Error", and "Time Length Exceeded". The one exception is the "Wrong Answer" error, which would offer the number of passing tests, but not runtime performance, and memory usage. A more detailed error analysis is offered in section 4.4.

Lastly, we note that there is potential survivorship bias in the human-generated responses because users only post correct submissions to the "Discussion" section. To mitigate this bias, we collect each problem's acceptance rate to re-standardize the percentage of human-generated codes that are accepted by LeetCode.

## 4. RESULTS

### 4.1 Overall Performance Evaluation

Across all question categories (Dynamic Programming, Matrix, and Sorting), there are varied success rates for each LLM. Notably, Claude Opus achieved the highest overall solution passing rate at 84.19%, followed by GPT-4 with a 75.44% passing rate as shown in Figure 2.

Further, within each category, performance was significantly influenced by the nature of the selected questions. In each category, questions that had fewer submissions and longer descriptions on LeetCode generally had lower passing rates, suggesting that the complexity of problem descriptions and their popularity among coders might affect LLM performance more than the problem category itself. However, one should note that problem complexity/length are often correlated with problem category on LeetCode. For example, there are more easy-difficulty and lower-complexity problems in the Array category than in the Dynamic Programming category.

### 4.2 Comparison of LLMs

Across all categories, GPT-4 consistently outperformed other models in terms of runtime, beating 68.56% of the sub-

missions on LeetCode. GPT-4 also showed superior memory management, beating 61.49% of the solutions on LeetCode. In contrast, despite Opus's strong passing rates, it lagged in runtime and memory efficiency, with only 59.85% and 54.34% of submissions beaten, respectively, which is the lowest among all models.

### 4.3 Human vs. LLM Performance

For the solutions we collected, human participants appeared to perform exceptionally well with a 96.23% passing rate. However, these figures were derived from a pool of already successful solutions, introducing a survivorship bias that inflated human performance metrics.

As mentioned in section 3.5, to address this bias and provide a fair comparison, we recalculated human performance metrics by adjusting for the inherent difficulty of the problems, as reflected by the average passing rates on LeetCode. Upon this adjustment, the real passing rate for human solutions was significantly reduced to 44.81%. This re-calibrated figure starkly contrasts the LLMs' performance; even the lowest-performing LLM, Claude Haiku, demonstrated a solution passing rate of 67.33%, markedly higher than the adjusted human rate.

Furthermore, our experimental results show that AI models substantially outperformed human programmers in terms of runtime and memory efficiency. For instance, GPT-4 consistently exceeded human performance, with its code beating 68.56% of LeetCode submissions in terms of runtime. GPT-4 generated code also beat 61.49% of LeetCode submissions in terms of memory efficiency. In comparison, human solutions only managed to beat 54.98% of submissions in runtime and 53.44% in memory performance on average. Even the least efficient LLM model in these respects, Opus, performed better than human coders, with runtime and memory metrics beating 59.85% and 54.34% of LeetCode submissions, respectively.

Initially, we had planned to conduct a t-test analysis to determine if the mean passing rates of human solutions versus LLM-generated solutions are statistically significant. However, during our data preparation phase, we noted that LeetCode does not release standard deviations for each of its problems' passing rates. Attempting to estimate them using the data from submissions would result in an extremely low standard deviation for human submissions because we are collecting from a (correct) subset of total submissions. Thus, conducting such a test would have meaningless results because of the skewed representation of data. Hence, we compare empirical averages for our analyses instead.

### 4.4 Error Analysis

Next, we conduct an error analysis, documenting which errors are most frequent among each of the LLMs. Our error analysis reveals that across all LLMs, the most frequent error types are "Wrong Answer" and "Runtime Error". As shown in Figure 2, both GPT-4 and Opus models exhibited a higher proportion of "Wrong Answers" and "Runtime Errors" compared to the other error categories. This observation suggests that the code generated by these models possessed the correct syntax and successfully passed the compilation phase.

		ALL Languages							
		Solution Accepted			Error				
Category	LLM	Passed Tests	Runtime Performance	Memory Performance	Total Error	Total Wrong Answers	Total Compile Errors	Total Runtime Errors	Total Time Limit Exceeded
ALL	GPT3.5 Summary	72.54%	64.31%	56.02%	32	22	4	8	0
	GPT4 Summary	75.44%	68.56%	61.49%	23	12	3	5	0
	Haiku Summary	67.33%	73.06%	57.43%	33	11	7	9	0
	Sonnet Summary	71.74%	68.37%	59.63%	35	22	4	6	2
	Opus Summary	84.19%	59.85%	54.34%	19	13	1	4	1
	Human Summary	96.23% x 46.57%	54.98%	53.44%	5	2	1	1	1
	Cross Model Summary	74.25%	66.83%	57.78%	142	80	19	30	12

Figure 2: Summary of Performance Metrics Across ALL Languages and All categories

However, the presence of "Wrong Answers" and "Runtime Errors" indicates the existence of logical inconsistencies or the failure to consider edge cases within the code, leading to the unsuccessful passing of test cases.

In contrast, the Claude Haiku model demonstrated a different error profile, with a higher incidence of "Compile Errors" and "Time Limit Exceeded" (TLE) errors. Compile errors arise when the code violates the syntax or semantics of the programming language, preventing successful compilation. The higher frequency of "compile errors" in the code generated by Claude Haiku suggests potential issues in code generation, such as incorrect syntax, improper variable declarations, or incompatible data types. On the other hand, TLE errors occur when the execution of the code exceeds the allocated time limit. The presence of TLE errors in the Claude Haiku model indicates potential inefficiencies in the generated code, such as sub-optimal algorithms or excessive computational complexity.

Once again, it is worth noting that error analysis for human participants was not applicable in this study, as their solutions were sourced from a pool of previously successful submissions. This approach ensures the correctness and efficiency of human-generated code, serving as a benchmark for evaluating the performance of AI models.

#### 4.5 LLM Performance by Problem Category

The performance of the LLMs varied across the Dynamic Programming (DP), Matrix, and Sorting categories as shown in Figure 3.

In the DP category, all models demonstrated nearly perfect solution acceptance rates, suggesting high competencies in applying DP principles.

The Matrix category, however, proved challenging. The average passing rate for LLMs was 49.05%, with the Haiku model performing the worst at 30.78%. Matrix problems involve complex operations on two-dimensional arrays, indicating limitations in AI models' ability to comprehend and manipulate matrix data structures effectively.

The LLMs generally performed well in the Sorting category, with GPT-4 excelling at an 82.11% passing rate and efficient runtime and memory performance. The strong performance suggests AI models' proficiency in understanding and implementing optimal sorting techniques.

#### 4.6 LLM Performance by Problem Characteristics

Further analysis revealed that Language Models (LLMs) performance was strongly correlated with problem charac-

teristics such as description length and submission rates on LeetCode, rather than problem categories themselves.

Matrix and Sorting problems in our experiment had lower submission rates, as shown in Figure 2 where one of the questions we selected (#1210) for Matrix only had approximately 19,000 submissions. Questions in Matrix and Sorting categories tend to have longer descriptions (over 300 words) as well. Meanwhile, DP problems had higher submissions (9M) and concise descriptions (under 50 words). Despite the inherent complexity of DP problems, LLMs performed better at DP problems, potentially because of the greater availability of training data, higher submission rates, and reduced complexity in understanding problem statements from shorter descriptions.

#### 4.7 LLM Performance by Programming Language

Our research yielded intriguing findings regarding the performance of Large Language Models (LLMs) across different programming languages. As depicted in Tables 3, 4, and 5 in the Appendix, LLMs generally exhibited the better performance in Go compared to Python and C++. Notably, for many questions, the average LLM performance surpassed more than 70% of human solutions on LeetCode in terms of runtime and memory efficiency using Go.

However, while LLMs may have a slight advantage in generating efficient code in Go, their overall performance remains relatively consistent across Python and C++. The similarity in performance across languages suggests that LLMs might have the ability to learn and adapt to the specific syntax, semantics, and best practices of different programming languages.

## 5. DISCUSSION

### 5.1 Evaluation of Experimental Design and Data Collection

Our experimental design involved testing 5 Large Language Models (LLMs)— GPT-3.5, GPT-4, Claude Haiku, Claude Sonnet, and Claude Opus— across 3 programming languages and 3 distinct coding problem categories on LeetCode. This broad spectrum allowed us to assess the robustness and versatility of LLMs in varied coding environments and problem complexities. The strength of this approach lies in its comprehensive coverage, which provides a nuanced understanding of LLM capabilities beyond simple task execution, examining their performance in terms of runtime efficiency, memory usage, and error frequency.

		ALL Languages								
		Solution Accepted			Error					
Category	LLM	Passed Tests	Runtime Performance	Memory Performance	Total Error	Total Wrong Answers	Total Compile Errors	Total Runtime Errors	Total Time Limit Exceeded	Total Memory Limit Exceeded
DP	GPT3.5 Summary	100%	66.51%	57.14%	0	0	0	0	0	0
	GPT4 Summary	89%	66.47%	62.00%	1	0	0	1	0	0
	Haiku Summary	100%	55.52%	43.32%	1	0	1	0	0	0
	Sonnet Summary	99%	53.48%	52.75%	3	3	0	0	0	0
	Opus Summary	96%	60.31%	50.00%	1	0	1	0	0	0
	Human Summary	100% x 37.53%	58.49%	49.30%	3	1	0	1	1	0
Matrix	Cross Model Summary	96.73%	60.86%	53.04%	6	3	2	1	0	0
	GPT3.5 Summary	44.16%	50.50%	51.16%	15	7	4	4	0	0
	GPT4 Summary	55.22%	61.27%	41.00%	14	10	1	3	0	0
	Haiku Summary	30.78%	66.14%	69.55%	20	5	3	6	6	0
	Sonnet Summary	40.11%	76.27%	60.41%	19	9	3	4	2	1
	Opus Summary	75%	51.18%	47.51%	9	7	0	2	0	0
Sorting	Human Summary	96.33% x 53.3%	54.40%	54.14%	2	1	1	0	0	0
	Cross Model Summary	49.05%	65.07%	53.93%	77	38	11	19	8	1
	GPT3.5 Summary	73.45%	75.91%	59.76%	17	15	0	2	0	0
	GPT4 Summary	82.11%	75.93%	81.46%	8	2	2	1	3	0
	Haiku Summary	71.22%	77.53%	59.43%	12	6	3	3	0	0
	Sonnet Summary	76.11%	75.35%	65.73%	13	10	1	2	0	0
	Opus Summary	81.56%	68.06%	65.52%	9	8	0	2	1	0
	Human Summary	92.36% x 48.87%	52.04%	56.88%	0	0	0	0	0	0
	Cross Model Summary	76.89%	74.56%	64.80%	59	39	6	10	4	0

Figure 3: Summary of Performance Metrics Across ALL Languages and Different categories

However, while LeetCode offers a controlled environment to systematically evaluate the coding capabilities of LLMs, it does not fully represent the breadth and depth of challenges faced by software engineers (SWEs) in real-world scenarios. LeetCode problems are typically algorithmic puzzles designed to test specific programming skills and theoretical knowledge, which are only a subset of the skills required in software development. In professional settings, SWEs must also handle tasks such as debugging legacy code, integrating APIs, managing databases, working within complex software architectures, and collaborating in team environments. These tasks often involve dealing with ambiguous requirements, making architectural decisions, and understanding the business logic that goes beyond the scope of algorithmic problem-solving typically found on LeetCode.

The reliance on LeetCode as a test platform may therefore limit our understanding of how well LLMs can perform in real-world software development tasks. The platform’s focus on individual, often isolated challenges may overlook how LLMs handle larger, more integrated projects that require a combination of coding proficiency, system design, and maintenance capabilities. Moreover, the discrete nature of LeetCode problems does not account for aspects of software development like version control, code review, scalability considerations, and security implications, which are crucial for a comprehensive assessment of an SWE’s work.

In light of these limitations, future research should consider incorporating more diverse and representative tasks that encompass a wider range of software engineering challenges. This could involve using project-based assessments, integrating tasks from open-source projects, or creating simulated software development environments that reflect the complex nature of modern software projects. Such approaches would provide a more accurate gauge of the potential for LLMs to contribute to or enhance the software development lifecycle, thereby offering deeper insights into their practical utility and limitations in the field of software

engineering.

## 5.2 Future Implications

As we have seen in this study, LLMs can reliably outperform humans on LeetCode in a variety of tasks. LLM development is still in its early stages, and companies are pouring billions of dollars a year into the development of more sophisticated LLMs. We expect the trend will continue and the gap between human and LLM software engineers will continuously expand in the coming years.

While the humans submitting LeetCode problems may not be a fair representation of professional software engineers, as we mentioned in section 5.1, if LLMs have not already reached this professional level, they will soon. There is a lot of uncertainty right now about what the future of software development will look like as LLMs continue to improve in that area. Assuming LLMs can code up to par with a professional software engineer, LLMs will be able to do so at a fraction of the cost. Thus, as we have seen in other industries that have been usurped by technology and automation, the jobs surrounding the space will inevitably change.

## 5.3 Implications for Programming Education

For over 70 years, the evolution of programming languages has been marked by an ongoing quest to simplify the common problems in programming. From the creation of high-level programming languages in the 1950s, aimed at bridging the gap between human reasoning and machine operations, to the development of user-friendly programming interfaces in recent decades, there has been a continuous effort to make programming simpler and easier to understand. Despite these advancements, programming remains a skill that requires significant knowledge, logical thinking, and talent.

With the advent of Large Language Models (LLMs) as we studied in this experiment, we may be witnessing a pivotal shift. Unlike traditional programming, where mastery over syntax and semantics is essential, LLMs allow individuals

to engage in "conversational programming." With this practice, users can code in their spoken language, with LLMs then translating those instructions into functional code. Our study already highlights the vast capabilities of these existing models to generate solutions for complex problems with efficiency and accuracy surpassing human performance.

Given these developments and studies, a fundamental question arises: is it still necessary to teach traditional programming, or should educational systems shift towards teaching students how to effectively interact with LLMs like the GPTs? As LLMs advance rapidly, fueled by the significant investments and research, driven by enormous financial benefits mentioned in section 5.2, traditional coding tasks that involve writing detailed lines of code might become less prevalent.

However, this does not render the teaching of programming obsolete. Instead, it suggests a transformation in the curriculum. Here, we suggest that future programming education should focus more on the following areas:

- **AI Literacy:** Understanding how AI models work, including their strengths, limitations, and ethical implications. As LLMs become integral tools in programming, education in these technologies must go beyond mere usage instructions. It should aim to build a deep understanding of how AI works, its limitations, and its broader impacts. This will ensure that future professionals are not only proficient in using AI but are also capable of critically assessing, managing, and directing these tools responsibly and ethically in the evolving landscape of software development.
- **Problem-Solving and Design Thinking:** In an era where AI can handle an increasing array of programming tasks, the human role in software development is evolving towards more complex problem formulation and system design. This shift necessitates a deeper focus on problem-solving and design-thinking skills within programming education, which are crucial for leveraging AI capabilities effectively.
- **Human-AI Collaboration:** As AI technologies become integral to software development processes, the ability to manage projects that leverage both human and AI contributors is becoming increasingly important. Human-AI collaboration is not merely about using AI as a tool but integrating AI into the software development team in a way that amplifies both the AI's capabilities and the human team's skills.

## 6. CONCLUSION

The results of this study on the application of Large Language Models (LLMs) in programming present a transformative vision for the future of software development and education. While our experiments reveal that LLMs can outperform human developers in efficiency and problem-solving within the constrained environment of LeetCode, these results do not immediately translate to a comprehensive takeover of programming jobs by AI. Instead, the results highlight the potential for AI to augment human abilities, particularly in well-defined problem spaces, where the

problem descriptions are concise and straightforward. As AI continues to evolve and as investment in this technology grows, the role of software developers is likely to shift towards more complex and creative problem-solving tasks. Education systems should strategically plan for the upcoming change within the next five to ten years, emphasizing the development of skills that enable effective human-AI collaboration. Classes, like CSE518A: Human-in-the-Loop Computation at Washington University in St. Louis, are great examples of such courses. Ultimately, as the landscape of technology progresses, the fusion of human ingenuity with advanced AI capabilities may unlock new realms of possibility in software development and beyond.

## 7. REFERENCES

- [1]<https://www.cognition-labs.com/introducing-devin>
- [2]<https://societysbackend.com/p/devin-has-exposed-software-engineers>
- [3]<https://dl.acm.org/doi/abs/10.1145/3510454.3522684>
- [4]<https://arxiv.org/abs/2306.05153>
- [5]<https://arxiv.org/abs/2305.1183>
- [6]<https://openai.com/blog/memory-and-new-controls-for-chatgpt>
- [7]<https://www.vellum.ai/blog/llm-benchmarks-overview-limits-and-model-comparison>
- [8]<https://www.statista.com/statistics/793628/worldwide-developer-survey-most-used-languages/>
- [9]<https://arxiv.org/abs/2104.03820>
- [10]<https://www.swebench.com/>
- [11]<https://paperswithcode.com/dataset/humaneval>

## 8. APPENDIX

Refer to the next few pages for more information on our full data collection and example LLM-generated codes.









		Python3										C++										Go									
		Solution Accepted					Error					Solution Accepted					Error					Solution Accepted					Error				
Category	Question#	LLM	Number of Tests	Passed Tests	Runtime Performance	Memory Performance	Error Type	Passed Tests	Runtime Performance	Memory Performance	Error Type	Passed Tests	Runtime Performance	Memory Performance	Error Type	Passed Tests	Runtime Performance	Memory Performance	Error Type	Passed Tests	Runtime Performance	Memory Performance	Error Type								
LLM Matrix Summary																															
217 Easy, 61.6% Acceptance Rate	GP13.5	75	75	Beats 18.80%	Beats 62.77%	NA	75	Beats 43.01%	Beats 44.51%	NA	75	85.15%	45.36%	NA																	
	GP13.5	75	75	Beats 73.08%	Beats 8.18%	NA	75	Beats 43.01%	Beats 25.33%	NA	75	87.57%	52.70%	NA																	
	GP13.5	75	75	Beats 34.43%	Beats 25.03%	NA	75	Beats 72.08%	Beats 63.67%	NA	75	93.28%	52.70%	NA																	
	GP13.5 Summary	75	75 (100%)	AVG 41.38%	AVG 31.39%	0 Error	100%	AVG 52.47%	Beats 44.4%	0 Error	100%	100%	88.67%	50.25%	0 Error																
	GP14	75	75	12.16%	23.85%	NA	75	82.89%	71.57%	NA	75	95.59%	80.37%	NA																	
	GP14	75	75	40.38%	31.90%	NA	75	82.89%	44.49%	NA	75	73.69%	85.94%	NA																	
	GP14	75	75	80.52%	88.49%	NA	75	59.58%	44.49%	NA	75	88.80%	87.34%	NA																	
	GP14 Summary	75	100%	44.35%	79.75%	0 Error	100%	81.78%	53.52%	0 Error	100%	100%	78.71%	81.22%	0 Error																
	Claude Haiku	75	75	86.71%	32.02%	NA	75	59.58%	44.53%	NA	75	88.28%	78.02%	NA																	
	Claude Haiku	75	75	72.85%	85.45%	NA	75	92.57%	25.30%	NA	75	78.32%	92.00%	NA																	
	Claude Haiku	75	75	63.81%	85.45%	NA	75	56.51%	44.53%	NA	75	81.03%	35.12%	NA																	
	Haiku Summary	75	100%	87.85%	54.31%	0 Error	100%	89.55%	38.12%	0 Error	100%	100%	79.87%	87.71%	0 Error																
	Claude Sonnet	75	75	82.92%	85.35%	NA	75	34.69%	25.30%	NA	75	85.43%	78.88%	NA																	
	Claude Sonnet	75	75	98.07%	85.35%	NA	75	59.58%	83.33%	NA	75	13.98%	75.10%	NA																	
	Claude Sonnet	75	75	75.31%	85.35%	NA	75	75.38%	71.57%	NA	75	85.43%	82.80%	NA																	
	Sonnet Summary	75	100%	84.77%	85.35%	0 Error	100%	86.54%	53.4%	0 Error	100%	100%	87.81%	72.28%	0 Error																
	Claude Opus	75	75	48.39%	31.97%	NA	75	69.20%	83.37%	NA	75	82.99%	49.33%	NA																	
	Claude Opus	75	75	97.41%	31.97%	NA	75	58.52%	44.54%	NA	75	85.73%	53.80%	NA																	
	Claude Opus	75	75	98.08%	85.45%	NA	75	30.31%	44.54%	NA	75	90.12%	84.97%	NA																	
	Opus Summary	75	100%	80.83%	45.13%	0 Error	100%	52.07%	50.82%	0 Error	100%	100%	88.28%	58.03%	0 Error																
	Human 1	75	75	54.57%	19.76%	NA	0	NA	NA	Wrong Answer	75	75.49%	77.67%	NA																	
	Human 2	75	75	42.92%	31.93%	NA	0	NA	NA	Wrong Answer	75	68.52%	90.27%	NA																	
	Human 3	75	75	87.20%	85.41%	NA	75	30.38%	72.69%	NA	75	51.82%	12.77%	NA																	
	Human Summary	75	100%	61.50%	59.01%	0 Error	53%	30.38%	72.69%	2 Error	100%	100%	65.34%	80.24%	0 Error																
	Exact Model Summary	75	100%	85.74%	55.13%	0 Error	100%	58.52%	48.07	0 Error	100%	100%	79.23%	85.48%	0 Error																
	630 Hard, 40.2% Acceptance Rate	GP13.5	97	75	NA	NA	Wrong Answer	75	NA	NA	Wrong Answer	82	NA	NA	Wrong Answer																
		GP13.5	97	75	NA	NA	Wrong Answer	75	NA	NA	Wrong Answer	97	90.91%	77.27%	NA																
		GP13.5	97	75	NA	NA	Wrong Answer	75	NA	NA	Wrong Answer	97	90.91%	77.27%	NA																
		GP13.5 Summary	97	78%	NA	NA	3 Error	77%	NA	NA	3 Error	87%	AVG 90.91%	AVG 77.27%	2 Error																
		GP14	97	97	53.79%	82.44%	NA	97	87.78%	88.84%	NA	97	95.98%	100%	NA																
		GP14	97	97	98.15%	58.67%	NA	97	97.74%	70.68%	NA	97	100%	100%	NA																
		GP14	97	97	31.05%	88.67%	NA	97	92.42%	88.84%	NA	97	100%	100%	NA																
GP14 Summary		97	100%	AVG 73.88%	AVG 88.93%	0 Error	100%	AVG 92.98%	AVG 81.45%	0 Error	100%	100%	AVG 85.16%	AVG 96.36%	0 Error																
Claude Haiku		97	97	88.93%	41.38%	NA	97	83.52%	88.70%	NA	97	80.00%	100%	NA																	
Claude Haiku		97	55	NA	NA	Wrong Answer	97	71.54%	88.70%	NA	0	NA	NA	Wrong Answer																	
Claude Haiku		97	55	NA	NA	Wrong Answer	97	84.42%	88.82%	NA	0	90%	30%	NA																	
Haiku Summary		97	77%	AVG 98.93%	AVG 41.38%	2 Error	100%	AVG 78.18%	AVG 88.07%	0 Error	87%	AVG 85.00%	AVG 85.00%	1 Error																	
Claude Sonnet		97	45	NA	NA	Wrong Answer	97	91.17%	88.85%	NA	37	NA	NA	Wrong Answer																	
Claude Sonnet		97	80	NA	NA	Wrong Answer	97	87.98%	98.85%	NA	0	NA	NA	Wrong Answer																	
Claude Sonnet		97	97	94.03%	59.08%	NA	97	88.30%	99.28%	NA	97	88.67%	50.00%	NA																	
Sonnet Summary		97	78%	AVG 94.03%	AVG 59.08%	2 Error	100%	AVG 88.48%	AVG 94.32%	0 Error	48%	AVG 88.67%	AVG 50.00%	2 Error																	
Claude Opus		97	97	79.44%	97.43%	NA	97	93.05%	70.68%	NA	97	20.00%	40.00%	NA																	
Claude Opus		97	97	50.32%	82.44%	NA	97	84.68%	70.68%	NA	97	80.00%	100%	NA																	
Claude Opus		97	97	85.01%	97.43%	NA	97	75.75%	70.68%	NA	97	20.00%	100%	NA																	
Opus Summary		97	100%	AVG 71.90%	AVG 92.43%	0 Error	100%	AVG 77.82%	AVG 70.88%	0 Error	100%	100%	AVG 40.00%	AVG 80.00%	0 Error																
Human 1		97	97	20.95%	41.04%	NA	97	80.34%	70.11%	NA	97	90.00%	100%	NA																	
Human 2		97	97	56.78%	58.75%	NA	97	5.17%	8.00%	NA	97	90.00%	40.00%	NA																	
Human 3		97	97	20.00%	13.87%	NA	97	88.97%	70.11%	NA	97	20.00%	10.00%	NA																	
Human Summary		97	100%	AVG 30.80%	AVG 37.80%	0 Error	100%	AVG 50.83%	AVG 49.04%	0 Error	100%	100%	AVG 88.67%	AVG 50.00%	0 Error																
Exact Model Summary		97	86%	83.75%	53.11%	7 Error	100%	58.52%	48.07	3 Error	82%	72.30%	73.71%	5 Error																	
2402 Hard, 44.8% Acceptance Rate		GP13.5	81	28	NA	NA	Wrong Answer	59	NA	NA	Wrong Answer	0	NA	NA	Wrong Answer																
		GP13.5	81	0	NA	NA	Runtime Error	8	NA	NA	Wrong Answer	60	NA	NA	Wrong Answer																
		GP13.5	81	72	NA	NA	Runtime Error	0	NA	NA	Wrong Answer	85	NA	NA	Wrong Answer																
		GP13.5 Summary	81	46%	NA	NA	3 Error	24%	NA	NA	3 Error	59%	NA	NA	3 Error																
		GP14	81	81	82.23%	94.43%	NA	82	NA	NA	Runtime Error	60	NA	NA	Wrong Answer																
		GP14	81	0	NA	NA	Runtime Error	0	NA	NA	Runtime Error	0	NA	NA	Wrong Answer																
		GP14	81	71	NA	NA	Wrong Answer	62	NA	NA	Runtime Error	0	NA	NA	Wrong Answer																
	GP14 Summary	81	69%	AVG 82.23%	AVG 94.43%	2 Error	51%	NA	NA	3 Error	25%	NA	NA	3 Error																	
	Claude Haiku	81	0	NA	NA	Runtime Error	60	NA	NA	Wrong Answer	0	NA	NA	Wrong Answer																	
	Claude Haiku	81	0	NA	NA	Runtime Error	71	NA	NA	Wrong Answer	0	NA	NA	Wrong Answer																	
	Claude Haiku	81	0	NA	NA	Runtime Error	60	NA	NA	Wrong Answer	60	NA	NA	Wrong Answer																	
	Haiku Summary	81	0%	NA	NA	3 Error	75%	NA	NA	3 Error	25%	NA	NA	3 Error																	
	Claude Sonnet	81	59	NA	NA	Wrong Answer	0	NA	NA	Runtime Error	50	NA	NA	Wrong Answer																	
	Claude Sonnet	81	88	NA	NA	Wrong Answer	0	NA	NA	Runtime Error	47	NA	NA	Wrong Answer																	
	Claude Sonnet	81	60	NA	NA	Wrong Answer	60	NA	NA	Wrong Answer	85	NA	NA	Wrong Answer																	
	Sonnet Summary	81	72%	NA	NA	3 Error	25%	NA	NA	3 Error	88%	NA	NA	3 Error																	
	Claude Opus	81	0	NA	NA	Runtime Error	60	NA	NA	Wrong Answer	50	NA	NA	Wrong Answer																	
	Claude Opus	81	0	NA	NA	Runtime Error	58	NA	NA	Wrong Answer	48	NA	NA	Wrong Answer																	
	Claude Opus	81	60	NA	NA	Runtime Error	0	NA	NA	Wrong Answer	49	NA	NA	Wrong Answer																	
	Opus Summary	81	29%	NA	NA	3 Error	49%	NA	NA	3 Error	80%	NA	NA	3 Error																	
	Human 1	81	81	21.92%	50.38%	NA	81	31.28%	31.08%	NA	81	15.21%	83.54%	NA																	

Figure 6: Appendix for All Sorting Questions Collected

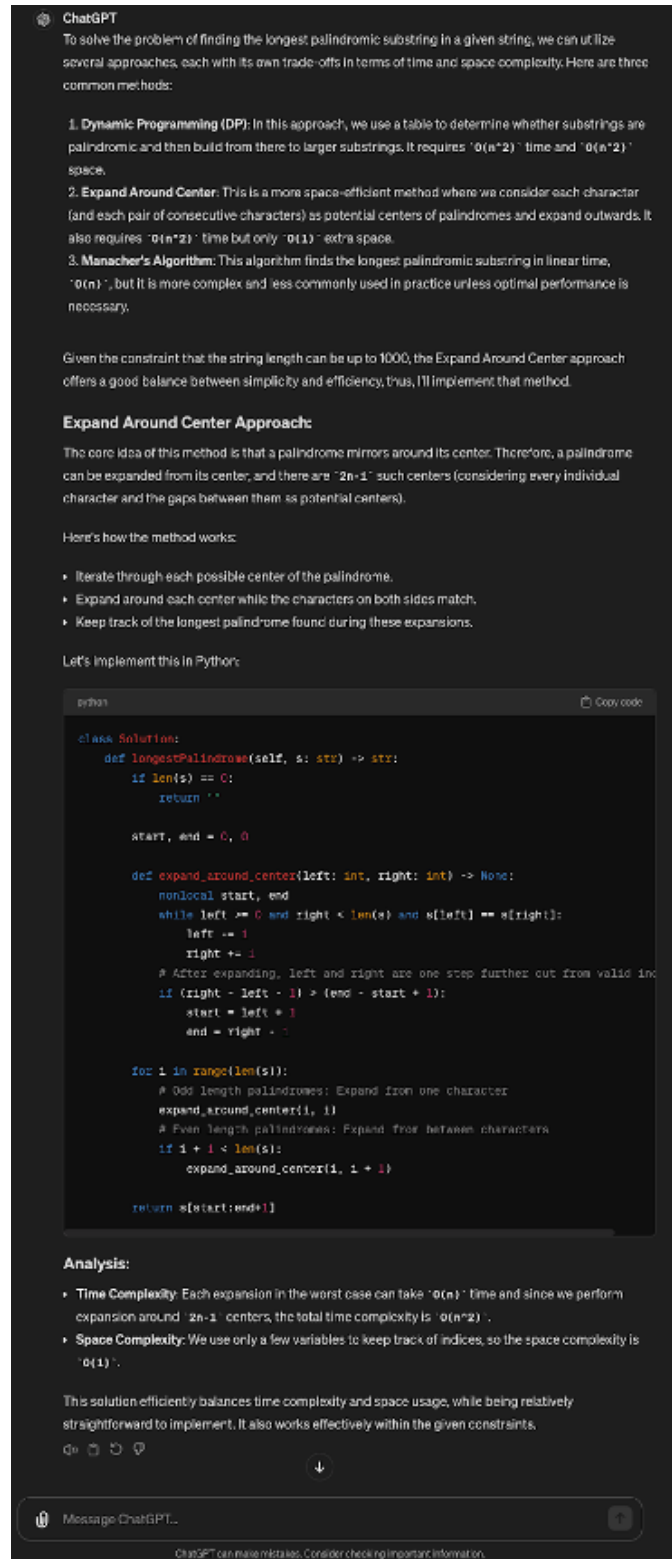


Figure 7: Appendix for Example ChatGPT Answer

To solve this problem, we need to find all the possible palindromic substrings in the given string `s` and return the longest one. A palindromic string is a string that reads the same backward as forward, like "racecar" or "level".

Here's a Python solution using the expanding around the center approach:

```
python
class Solution:
    def longestPalindrome(self, s: str) -> str:
        def expand_around_center(left, right):
            while left >= 0 and right < len(s) and s[left] == s[right]:
                left -= 1
                right += 1
            return s[left + 1:right]

        if not s:
            return ""

        longest_palindrome = ""

        # Expand around each character and pairs of characters
        for i in range(len(s)):
            # Odd length palindromes
            tmp = expand_around_center(i, i)
            if len(tmp) > len(longest_palindrome):
                longest_palindrome = tmp

            # Even length palindromes
            tmp = expand_around_center(i, i + 1)
            if len(tmp) > len(longest_palindrome):
                longest_palindrome = tmp

        return longest_palindrome
```

#### Explanation:

1. The `expand_around_center` function takes two indices `left` and `right` as input and expands outward from the center until it finds the maximum palindrome around that center.
2. In the `longestPalindrome` function, we first handle the edge case where the input string `s` is empty by returning an empty string.
3. We initialize an empty string `longest_palindrome` to store the longest palindromic substring found so far.
4. We iterate through each character in the string `s` using a loop.
5. For each character, we consider two cases:
  - Odd length palindromes: We call `expand_around_center` with `i` as both `left` and `right` indices, effectively treating the current character as the center of the palindrome.
  - Even length palindromes: We call `expand_around_center` with `i` and `i + 1` as the `left` and `right` indices, respectively, treating the current character and the next character as the center of the palindrome.
6. After expanding around each center, we update `longest_palindrome` with the longer palindromic substring found.
7. Finally, we return `longest_palindrome`, which contains the longest palindromic substring in `s`.

#### Assumptions:

- The input string `s` consists of only digits and English letters, as per the constraints.
- The length of the input string `s` is within the specified range of  $1 \leq s.length \leq 1000$ .

#### Time Complexity:

The time complexity of this solution is  $O(n^2)$ , where  $n$  is the length of the input string `s`. This is because in the worst case, we need to expand around each character and each pair of characters, which takes  $O(n)$  time for each expansion. Since we have at most  $2n$  centers ( $n$  characters and  $n - 1$  pairs of characters), the overall time complexity is  $O(n^2)$ .

#### Space Complexity:

The space complexity of this solution is  $O(1)$ , as we are not using any additional data structures that scale with the input size. We are only using a constant amount of extra space to store temporary variables.

⚠️ Claude does not have the ability to run the code it generates yet.

📄 Copy 🔄 Retry 🗑️



Claude can make mistakes. Please double-check responses.

Figure 8: Appendix for Example Claude Answer