Code Generation by Large Language Models: A Comparative Analysis of ChatGPT, Claude, and DeepSeek

Yousef Alraba'nah[1,2,*], Azzam Sleit[2], Iyas Qaddara[3], and Mohammad Hiari[4]
[1] Department of Software Engineering, Al-Ahliyya Amman University, Amman, Jordan
[2] Department of Computer Science, University of Jordan, Amman, Jordan
[3] Department of Computer Science, Al-Ahliyya Amman University, Amman, Jordan
[4] Department of Networks and Cybersecurity, Al-Ahliyya Amman University, Amman, Jordan
Email: y.alrabanah@ammanu.edu.jo (Y.A.), azzam.sleit@ju.edu.jo (A.S.), i.qaddara@ammanu.edu.jo (I.Q.), m.hyari@ammanu.edu.jo (M.H.)

Manuscript received September 4, 2025; revised October 18, 2025; accepted October 22, 2025
*Corresponding author

***Abstract*—As generative Artificial Intelligence (AI) models become increasingly integrated into software development workflows, understanding their efficiency and code quality is critical. This study offers a comprehensive comparison of three leading AI models—ChatGPT GPT-4-turbo, Claude Sonnet, and DeepSeek-V3—for automated code generation, focusing specifically on sorting algorithms. The models are evaluated across multiple metrics including execution time, memory usage, peak memory consumption, logical and physical file sizes, and code readability. Python implementations of Insertion Sort, Merge Sort, Quick Sort, and Heap Sort are generated by each model and benchmarked in a consistent Linux Docker environment. Results reveal that ChatGPT leads in overall efficiency, with the fastest average execution time, the lowest peak memory usage, and the highest readability scores. DeepSeek demonstrated competitive performance, especially in producing readable code, while Claude showed higher memory consumption and lower readability. This analysis provides practical insight into the trade-offs between code quality and system performance in AI-generated programming, offering valuable guidance for researchers and developers alike.**

***Index Terms*—ChatGPT, Claude sonnet, code generation, DeepSeek, large language model**

I. Introduction

The recent evolution of Artificial Intelligence (AI), especially in the domain of Large Language Models (LLMs), has sparked widespread interest in their applications across scientific and engineering disciplines [1]. LLMs such as OpenAI's ChatGPT, Anthropic's Claude, and DeepSeek which is a newer open-source model from China, have gained significant traction not only for their conversational fluency but also for their problem-solving and code generation abilities [2]. These models, powered by transformer-based neural architectures, are capable of parsing natural language instructions and converting them into syntactically correct and often logically sound programming code [3]. Their integration into software development workflows, education, and scientific computing heralds a new era of human-AI collaboration [4]. However, despite their growing influence, a systematic and comparative evaluation of these models in domain-specific coding tasks remains underexplored.

LLMs are trained on the large corpora of books, websites, programming documentation and open source code repositories. For this reason, they have achieved both the ability to understand and the ability to create source code in a number of different programming languages [5]. In early iterations of LLMs such as GPT-2 and GPT-3, this functionality was present in a very restricted manner, but the newest models ChatGPT-4, Claude 3.5/3.7 Sonnet and DeepSeek-R1 are significantly better at handling context, logical reasoning and execution reliability [6]. OpenAI's ChatGPT is now very popular and is well known for doing natural language tasks and software development. ChatGPT o3-mini and ChatGPT-4o are both science and coding optimized models. These models are able to generate efficient and human-readable code with practical values such as faster response time, low latency and advanced contextual reasoning, which are useful for Integrated Development Environments (IDEs), educational tools and professional workflows [7]. Anthropic's Claude is developed with a focus on safety, reliability and long context reasoning. For example, the Claude 3.7 Sonnet variant includes 'extended thinking mode' that will offer higher performance for tasks that are mathematical and logical such as code writing. Claude's architecture is designed to tradeoff between coherence and correctness, but his outputs have often been highly readable and it performs extremely well on instruction following tasks which is important when programming [8]. The other development is that DeepSeek is an open source AI. On competitive math and logic benchmarks, it has shown strong performance, including 79.8% on the AIME 2024, slightly beating ChatGPT o1. One of its appeals is that DeepSeek is cost efficient, transparent and open which has attracted developers and researchers in search of customizable, low-cost AI tools [9].

Although these advantages are promising, there are still significant challenges requiring the empirical evaluation. A language model can generate syntactically correct but semantically inaccurate code, has an inconsistent memory management and performance may vary based on prompt wording or the complexity of the problem being solved [10]. In addition, while widely used in industry and even demonstrated to be effective based on anecdotal evidence and case studies, there are few quantitative comparisons of performance differences, especially for fundamental algorithmic problems such as sorting.

In this work, we seek to fill that gap by assessing the performance of ChatGPT GPT-4-turbo, Claude Sonnet 4 and DeepSeek-V3 with specific focus on code generation performance when solving sorting problems, a central class of algorithmic tasks that define excellent benchmarks to gauge the quality of code generated, its efficiency and the logic applied. In particular, we analyze the code generated along four important dimensions: how readable the code is, how much it consumes in terms of memory and computation time and its logical size. As such, these dimensions collectively express how usable, efficient and scalable, in general, AI-generated code is for practical, real world programming contexts. We intend our findings to help serve both as a technical and practical understanding of the LLM's capabilities in low-level algorithmic challenges and add to the growing domain of AI assisted programming.

II. Related Works

Recent research has extensively explored the capabilities and comparative performance of LLMs such as DeepSeek, ChatGPT and Claude Sonnet in the domain of AI-driven code generation and programming assistance. A notable comparative study focusing on Python code generation using online judge challenges demonstrated that DeepSeek (version R1) achieved higher correctness and more frequent first-attempt acceptance, especially in algorithmic tasks. In contrast, ChatGPT (version o1) produced code with fewer errors related to memory and execution time and tended to write more concise programs, underscoring a trade-off between bug-free output and efficiency [11]. Expanding beyond code correctness, evaluations on scientific computing and scientific machine learning tasks reveal that reasoning-optimized models including DeepSeek R1, ChatGPT o3-mini-high, and Claude (3.7 Sonnet) excel at recognizing problem contexts and applying advanced mathematical reasoning. This suggests that domain-specific knowledge and reasoning capabilities are critical factors in effective AI-assisted scientific programming [12].

Furthermore, the benchmarking efforts on standardized dataset like HumanEval and MBPP show ChatGPT outperformed others in the code fluency and multi-language support and DeepSeek produces compact and more efficient code. This differs from Claude in that Claude is distinguished for writing maintainable and well documented code that reflects a variety of strengths suited to different use case [13].

The study of Huang *et al.* [14] has offered an empirical evaluation of LLMs (both open-source and closed-source) over 1,000 efficiency-critical programming problems drawn from LeetCode. The study showed that even top models such as GPT-4 generate code that is significantly less efficient in execution time and memory usage compared to canonical human solutions, sometimes by a large margin up to ~13.9× slower, ~43.9× higher memory consumption for certain problems. This work underscored that correctness benchmarks alone are insufficient to capture real performance differences. Qing *et al.* [15] extended the efficiency focus of prior work by supporting multiple programming languages (Python, C++, Java, JavaScript, Ruby, Go) and using competitive programming tasks with human-expert efficiency baselines. The findings revealed that while some LLMs generate functionally correct code, they often lag human experts in efficiency, and that performance varies significantly by language.

Palla and Slaby [16] presented a rigorous empirical evaluation of AI-generated code focusing on optimization and performance metrics under constrained hardware settings. This work investigated how model-generated solutions compare to human baselines in terms of runtime efficiency, memory consumption, and code compactness across multiple algorithmic tasks. Its findings underscored that even when correctness is achieved, performance degradations emerge due to non-optimal code structure or redundant computations.

Another study of Zheng *et al.* [17] examined LLMs code generation capability across a wide variety of application domains and programming languages. It covered domains such as web, mobile, IoT, robotics, cloud services, and enterprise applications, evaluating how well LLMs perform in domain-specific settings. The study highlighted domain variability but typically didn't drill down into algorithmic complexity, or metrics like memory usage or file size. In terms of readability, Sergeyuk [18] investigated how well code readability models align with human developers' judgments on AI-generated Java code. The work found that some models and metrics often miss developer notions of readability, and that concise, executable code is often seen as more readable. However, correlation is imperfect, pointing to limitations in existing readability metrics. More recently, Sabra *et al.* [19] evaluated the code produced by several modern LLMs via static analysis tools such as SonarQube on thousands of Java code assignments. The findings revealed that even when code passes functional correctness benchmarks, many security defects, code smells, and bugs persist, showing that correctness alone is insufficient for assessing production readiness. Complementing generation capabilities, ChatGPT's code refactoring skills have been empirically demonstrated to preserve functionality while enhancing structure and generating accurate documentation, highlighting its utility in code maintenance workflows [20]. Phogat *et al.* [21] conducted a comparative analysis of four prominent large language models—ChatGPT, DeepSeek, Claude, and Qwen—

focusing on their performance in code generation tasks. The study evaluated these models across various parameters, including accuracy, efficiency, and code quality. The findings revealed that DeepSeek outperformed the others in terms of computational efficiency, generating code that executed faster and consumed fewer resources. ChatGPT demonstrated superior accuracy and produced more human-readable code, making it suitable for applications requiring high precision and clarity. Claude exhibited strengths in generating well-structured and maintainable code, while Qwen, though competitive, showed variability in performance across different tasks.

Table I summarizes the related works including their approach, finding, advantages and limitations.

TABLE I: SUMMARY OF RELATED WORKS

Ref.	Approach / Findings	Advantages	Limitations
[11]	Compared ChatGPT and DeepSeek on code generation tasks using multiple programming benchmarks. DeepSeek achieved higher correctness, while ChatGPT provided cleaner and more efficient code.	Highlights performance trade-offs between correctness and efficiency.	Focused mainly on Python; lacks multi-domain and large-scale evaluation.
[12]	Investigated DeepSeek and ChatGPT performance on scientific computing and machine learning tasks. DeepSeek excelled in mathematical reasoning and numerical accuracy.	Demonstrates importance of reasoning optimization in domain-specific computing.	Limited to scientific domains; no human-centered readability evaluation.
[13]	Proposed a causality-based framework for benchmarking LLM-generated code, focusing on explainability and quality attribution.	Provides insights into why LLMs generate certain code patterns.	Evaluation limited to static code analysis; lacks runtime and memory tests.
[14]	Introduced EffiBench, a benchmark measuring execution efficiency, memory, and runtime of AI-generated code	Offers standardized metrics for computational efficiency.	Limited language diversity; lacks human evaluation of readability.
[15]	Proposed EffiBench-X, a multi-language benchmark evaluating efficiency and performance of LLM-generated code across several languages.	Expands efficiency testing to cross-language scenarios.	Focuses only on efficiency; omits code maintainability and style.
[16]	Conducted empirical evaluation of generative AI models (ChatGPT, Claude, DeepSeek) in Python code generation using quantitative metrics.	Comprehensive benchmarking with reproducible experimental design.	Restricted to Python; excludes qualitative readability assessment.
[17]	Developed DomainCodeBench to assess LLMs across domain-specific tasks, showing that general performance does not guarantee domain excellence.	Reveals mismatch between general benchmarks and domain-specific needs.	Lacks efficiency and readability measures.
[18]	Reassessed Java code readability using a human-centered evaluation approach combining metrics and developer feedback.	Integrates human perception into readability modeling.	Limited to Java and does not involve LLM-generated code.
[19]	Assessed AI-generated code for quality and security vulnerabilities across LLMs. ChatGPT produced safer code, while DeepSeek prioritized performance.	Combines performance and security perspectives.	Focuses on security; omits human readability and efficiency comparisons.
[20]	Empirically evaluated ChatGPT's ability to refactor and improve existing code structures while maintaining functionality.	Demonstrated ChatGPT's strength in improving readability and maintainability.	Focused only on ChatGPT; lacks multi-model comparison.
[21]	Compared ChatGPT, DeepSeek, Claude, and Qwen on reasoning, code generation, and contextual understanding using real-world benchmarks.	Broad evaluation across four models, highlighting relative strengths.	Limited to high-level comparisons; lacks deep runtime and memory profiling.

Despite extensive evaluations of LLMs like DeepSeek, ChatGPT, and Claude across various programming tasks, several gaps remain. Most existing studies focus on broad code generation benchmarks or domain-specific scientific computing, with limited attention to detailed analysis on fundamental algorithmic problems such as sorting. Moreover, while many works compare correctness and efficiency, fewer systematically assess readability, in tandem with execution metrics. These gaps highlight the need for a more comprehensive and fine-grained comparative study that integrates multiple evaluation dimensions to better inform practical AI code assistant selection. This study brings the following contributions, with the aim of addressing these gaps.

1) Provide a fine-grained comparison of three major LLMs ChatGPT (GPT-4-turbo), Claude Sonnet, and DeepSeek-V3 on classic sorting algorithms (Insertion Sort, Merge Sort, Quick Sort, and Heap Sort) to capture computational and structural performance differences.
2) Introduce a multi-dimensional evaluation across execution time, memory usage, peak memory consumption, logical and physical file size, and code readability, combining quantitative system metrics with qualitative readability analysis.
3) Benchmark all models under a controlled Linux Docker environment, ensuring fair and reproducible comparison across models and runs.
4) Establish correlations between readability, code length, and computational performance—providing deeper insights into the trade-off between human readability and machine efficiency.
5) Offer practical guidance for researchers and developers on selecting appropriate LLMs based on project needs—whether prioritizing runtime efficiency, memory economy, or code clarity.

III. METHODOLOGY

This section outlines the methodology designed to evaluate the performance of three LLMs —ChatGPT GPT-4-turbo, Claude Sonnet 4 and DeepSeek-V3 —on a controlled set of algorithmic code generation tasks. The study focuses on sorting algorithms due to their foundational importance in programming and their sensitivity to both logical structure and execution efficiency. The methodology is divided into several key components: model selection, dataset and prompt design, evaluation metrics, and experimental setup. Fig. 1 shows the methodology.

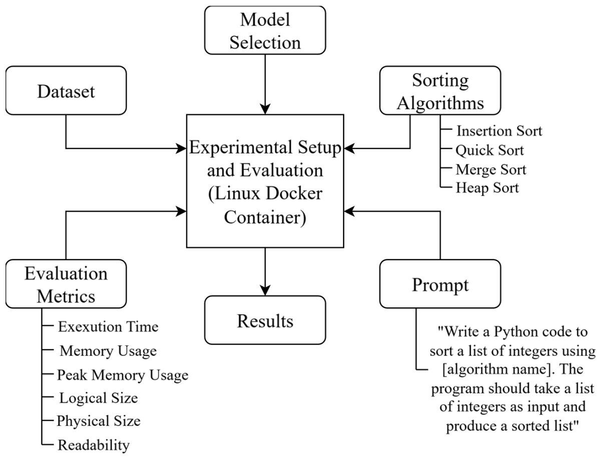
Model Selection
Dataset
Sorting Algorithms
Insertion Sort
Quick Sort
Merge Sort
Heap Sort
Experimental Setup and Evaluation (Linux Docker Container)
Evaluation Metrics
Exexution Time
Memory Usage
Peak Memory Usage
Logical Size
Physical Size
Readability
Results
Prompt
"Write a Python code to sort a list of integers using [algorithm name]. The program should take a list of integers as input and produce a sorted list"

Fig. 1. Research methodology.

A. Model Selection

To provide a comprehensive and fair evaluation of LLMs in code generation tasks, we selected three state-of-the-art models: ChatGPT, DeepSeek, Claude. These models were chosen due to their wide adoption, reported performance in recent evaluations, and complementary design objectives. Together, they represent a diverse set of approaches to AI-driven programming assistance-spanning from general-purpose reasoning models to lightweight, speed-optimized tools, making them suitable for contrasting different trade-offs in code generation. Each model was accessed through its respective official interface to ensure consistency in prompt delivery and result evaluation.

B. Dataset Description

To evaluate the performance and scalability of the selected LLMs in generating sorting algorithms, we constructed a synthetic dataset comprising six distinct input sets. The dataset consists of arrays of randomly generated integer values, with sizes incrementally increasing by a factor of ten to test algorithmic behavior under different computational loads. Specifically, the six subsets contain 10, 100, 1,000, 10,000, 100,000, and 1,000,000 integers, respectively. Each subset represents a separate test case to analyze how the generated code performs across small, moderate, and large-scale data. This structured design enables us to measure model efficiency, correctness, and resource utilization (execution time and memory) as input size grows, providing insight into the models' scalability and optimization quality.

C. Task Design and Prompting

Due to the necessity of consistency and control over the evaluation process, the study focused on sorting algorithm generation, which is a fundamental task in computer science that tests a model's capability of comprehending problem specifications, making logical reasoning and generating optimized code. The models were asked to generate an implementation of a complete sorting algorithm among insertion, merge, quick and heap sort algorithms. The generated code was then evaluated against the dataset to assess both correctness and performance. This task was chosen because sorting problems have well-defined expected outputs, making them ideal for benchmarking across correctness, speed, memory usage, and logical structure.

For fairness and reproducibility reason, the same problem statement and requirements were used across all models. The output was saved and the resulting runtime behavior was logged after execution in a controlled environment. Prompting plays a crucial role in determining the quality of output generated by LLMs. We used zero-shot prompting in most cases—providing only a concise problem description without any additional examples or templates. This setup simulates a real-world scenario where developers request helps from an assistant by describing a task in natural language. The general prompt format used was: "Write a Python code to sort a list of integers using [algorithm name]. The program should take a list of integers as input and produce a sorted list".

Prompts were carefully tested to ensure clarity and no bias toward a specific model. In addition, we ensured the prompts did not include performance hints or optimization suggestions, allowing the model to infer the most efficient implementation based on its training.

D. Evaluation Metrics

In this study, a suite of quantitative metrics is leveraged to comprehensively evaluate the performance and quality of the code generated by LLMs. These are metrics that aim to consider not only whether the output of a model is correct, but also how computationally efficient it is and the code level properties, in order to get the complete picture of what is the practical utility of each model.

Execution Time (ET): Execution time refers to the total duration the generated sorting algorithm takes to complete processing a given input dataset. It is measured in Seconds and represents the elapsed time from the start to the end of the function execution. Let A be a sorting algorithm generated by a LLM, and let D_n represent an input dataset of size n. The average time taken by algorithm A to sort dataset D_n over k=5 runs:

$$\mathrm{ET}(D_n, A) = \frac{1}{k}\sum_{i=1}^{k}\left(t_{\text{end}}^{(i)} - t_{\text{start}}^{(i)}\right) \quad (1)$$

where $t_{\text{start}}^{(i)}$ and $t_{\text{end}}^{(i)}$ are the start and end timestamps of the ith run.

Memory Usage (MU): Memory usage reflects the amount of system memory (in kilobytes) consumed by the program during its execution. It includes memory allocated for variables, data structures, and recursive stack frames where applicable. The average memory consumed during execution (in kilobytes):

$$\mathrm{MU}(D_n, A) = \frac{1}{k}\sum_{i=1}^{k}\left(m_{\text{end}}^{(i)} - m_{\text{start}}^{(i)}\right) \quad (2)$$

where $m_{\text{start}}^{(i)}$ and $m_{\text{end}}^{(i)}$ are memory usage measurements before and after execution for run i.

Peak Memory Usage (PMU): Maximum memory usage when using the algorithm is described by peak memory usage. In particular, this is important for recursive or memory intensive algorithms as temporary lists can greatly increase peak demand. The average of peak memory usage (in kilobytes) as recorded by a memory tracer:

$$\mathrm{PMU}(D_n, A) = \frac{1}{k}\sum_{i=1}^{k}\left(m_{\mathrm{peak}}^{(i)}\right) \tag{3}$$

where $m_{\mathrm{peak}}^{(i)}$ is the peak memory traced during run i.

Logical Size (LS): The logical size of a file refers to the actual size of the file's contents as reported by the operating system, excluding any file system overhead. It is measured in kilobytes (KB). The logical size (in kilobytes) is computed as follows:

$$\mathrm{LS}(A) = \frac{\mathrm{os.path.getSize}(A)}{1024} \tag{4}$$

where $\mathrm{os.path.getSize}(A)$ returns the size (in bytes) of the file specified by the path argument.

Physical Size (PS): refers to the actual disk space allocated for the file, which includes file system block overhead. The physical size (in kilobytes) is computed as follows:

$$PS(A) = blocks\ allocated \times Block\ size\ (KB) \tag{5}$$

whereas allocated blocks are the number of blocks assigned to store the file, and block size is the size of each storage block in kilobytes (typically 4 KB on many systems).

Readability Score (SC): Readability assesses how easily a human can understand the generated code. This includes factors like consistent formatting, use of meaningful variable and function names, clarity of logic flow, and presence of comments or documentation. Readability score (RS) is often normalized to a 10-point scale, such as:

$$\mathrm{RS}(A) \in [0,10] \tag{6}$$

IV. Experiments and Results

A. Experimental Setup

All experiments were conducted on a personal computer running Windows 10 Pro (64-bit) with the following hardware specifications: Processor is Intel Core i7-7700 CPU @ 3.60GHz, RAM is 8,192 MB (8 GB), and System Architecture is x64-based processor. To ensure consistency and isolation, a Linux-based Docker container was used to execute and evaluate the generated code. The Docker container was configured with the following constraints: operating system is minimal Linux image, allocated CPU is 1 core, allocated memory is 1 GB, and programming language is Python 3.10-slim.

All code generated by the LLMs was written and executed in Python, ensuring a standardized testing environment. The use of Docker allowed for reproducibility and minimized system-level interference when measuring performance metrics such as execution time and memory usage. All scripts were executed inside the container, and relevant metrics were collected during runtime using Python-based profiling tools. Specifically, the experiments leverage time python tool for precise execution time measurement, psutil to measure the resident set size (RSS) memory used by the process before and after execution, and tracemalloc to capture peak memory usage during the sorting task. Moreover, os tool is used for retrieving the physical file size of the script. Also, Garbage collection was temporarily disabled to ensure consistent memory profiling.

Each sorting algorithm was tested on six datasets of increasing size: 10, 100, 1,000, 10,000, 100,000, and 1,000,000 integers. For reliability, each algorithm was run five times on each dataset, and the average of the metrics, execution time, memory usage, peak memory usage and logical file size, was calculated and used in the final evaluation. This setup ensured a fair and systematic comparison of code generated by different LLMs under controlled computing conditions.

B. Results

1) Execution Time (ET) evaluation

To evaluate the efficiency of code generated by LLMs, we measured the execution time (in seconds) of four sorting algorithms—Insertion Sort, Merge Sort, Quick Sort, and Heap Sort—across six datasets of increasing size (DS1: 10 integers to DS6: 1,000,000 integers). Each algorithm was executed five times per dataset, and the average time was recorded. Table II shows the result of model evaluation in term of average execution time.

TABLE II: THE AVERAGE EXECUTION TIME OF MODELS (IN SECONDS)

Algorithm	Dataset (DS)	ChatGPT	Claude Sonnet	DeepSeek
Insertion Sort	DS 1	0.0000156	0.0000198	0.0000188
	DS 2	0.0007444	0.0007822	0.0006032
	DS 3	0.1427272	0.1346232	0.1430604
	DS 4	14.8687454	15.3280544	15.0527078
	DS 5	1521.8972000	1589.217278	1542.253533
	DS 6	> 50400	> 50400	> 50400
Merge Sort	DS 1	0.0000596	0.0000516	0.0000514
	DS 2	0.0006136	0.0005694	0.0006294
	DS 3	0.007821	0.0084864	0.007714
	DS 4	0.1054434	0.0994446	0.1025194
	DS 5	1.376218	1.3603642	1.2997082
	DS 6	17.0925022	16.2943608	16.6534512
Quick Sort	DS 1	0.0000416	0.0000358	0.0000396
	DS 2	0.000286	0.000246	0.0002678
	DS 3	0.0097646	0.0076344	0.0091446
	DS 4	0.1246186	0.1272454	0.1471322
	DS 5	1.232454	1.1939666	1.2563958
	DS 6	15.4996032	15.708026	15.3957354
Heap Sort	DS 1	0.000044	0.0000486	0.0000326
	DS 2	0.0005974	0.0005842	0.0005276
	DS 3	0.01195	0.0112248	0.0123984
	DS 4	0.1717594	0.1919182	0.2006176
	DS 5	2.2620296	2.3792628	2.3275016
	DS 6	29.89498	31.2612638	30.6523804

For Insertion Sort, which has a time complexity of $O(n^2)$, performance quickly degraded as dataset size increased. On small datasets such as DS1 (10 elements), ChatGPT produced the fastest execution with an average time of 0.0000156 s, followed by DeepSeek (0.0000188s) and Claude (0.0000198s). As input size grew, ChatGPT generally showed better scalability on large-range inputs like DS4 and DS5, but none could complete DS6 (1,000,000 elements) within the set threshold of 14 h (> 50400s). This clearly illustrates Insertion Sort's inefficiency for large-scale datasets regardless of how optimized the code is. In the case of Merge Sort, with its (*n*log*n*) efficiency, all models scaled much better. Claude consistently performed well, achieving the fastest completion time on DS6 at 16.29 s, ahead of DeepSeek (16.65s) and ChatGPT (17.09s). On smaller datasets (DS1 to DS3), the differences between the models were minimal, often within microseconds, indicating uniformly efficient code generation for this divide-and-conquer algorithm.

Quick Sort revealed Claude's advantage on the smallest inputs—e.g., DS1, where it executed in 0.0000358s, faster than ChatGPT (0.0000416s) and DeepSeek (0.0000396s). However, DeepSeek demonstrated the best performance on larger inputs, especially DS6, completing in 15.39s—faster than ChatGPT (15.49s) and Claude (15.70s). This suggests that while Claude optimized base cases well, DeepSeek produced code better suited for recursive efficiency. With Heap Sort, performance was generally slower compared to Merge and Quick Sort due to its more complex memory and tree-based operations. DeepSeek produced the fastest result on DS1 at 0.0000326s, but ChatGPT offered better performance on larger datasets. For example, on DS6, ChatGPT completed in 29.89, ahead of DeepSeek (30.65s) and Claude (31.26s), and on DS5, ChatGPT had the lowest time (2.26s).

Fig. 2 shows the average execution time of all algorithms for each model. Overall, ChatGPT recorded the lowest average execution time across all datasets and algorithms at 69.77 s, closely followed by DeepSeek at 70.67 s, and Claude at 72.75 s. This suggests that while ChatGPT's generated code is slightly more efficient in total, DeepSeek consistently delivered high performance, particularly for larger datasets. Claude showed strengths on smaller inputs but lagged slightly as input size increased.

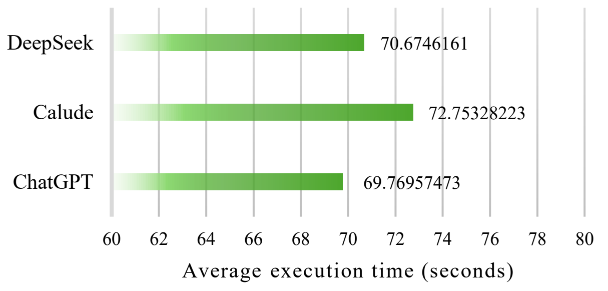

Fig. 2. The average execution time of models.

2) *Memory Usage (MU) evaluation*

Table III shows the result of models' evaluation in term of average memory usage. Memory consumption patterns varied across sorting algorithms and dataset sizes. For Insertion Sort, all models showed negligible memory usage (0 KB) for small datasets (DS1–DS3). As the input size grew, memory usage increased noticeably. For DS4 and DS5, ChatGPT consumed 764 KB and 13,260 KB respectively, while DeepSeek used slightly less with 751.2 KB and 13,056 KB. Claude consumed the most memory in DS5 (13,980 KB). Memory usage on DS6 is not reported, likely due to execution limits. Merge Sort exhibited higher memory demand for larger datasets, consistent with its divide-and-conquer nature. For DS4, memory use hovered around 1,000 KB across all models. On DS5, usage climbed above 17,000 KB, with Claude consuming the most (17,340 KB) and DeepSeek the least (17,316 KB). On DS6, all models required over 160,000 KB, with DeepSeek again being the most efficient (159,047.2 KB), compared to ChatGPT (163,967.2 KB) and Claude (174,936 KB).

TABLE III: THE AVERAGE MEMORY USAGE OF MODELS (IN KB)

Algorithm	Dataset (DS)	ChatGPT	Claude Sonnet	DeepSeek
Insertion Sort	DS 1	0	0	0
	DS 2	0	0	0
	DS 3	0	0	0
	DS 4	764	811.2	751.2
	DS 5	13260.00	13980	13056
	DS 6	-	-	-
Merge Sort	DS 1	0	0	0
	DS 2	0	0	0
	DS 3	0	0	0
	DS 4	1060	999.2	1048
	DS 5	16723.2	17340	17316
	DS 6	163967.2	174936	159047.2
Quick Sort	DS 1	0	0	0
	DS 2	0	0	0
	DS 3	0	0	0
	DS 4	776	764	800
	DS 5	13115.2	13896	13115.2
	DS 6	132840	140861.6	132864
Heap Sort	DS 1	0	0	0
	DS 2	0	0	0
	DS 3	0	0	0
	DS 4	788	776	764
	DS 5	11196.738	13868.8	13082.4
	DS 6	132844.8	140855.2	132920.8

Quick Sort followed similar trends. It required minimal memory on smaller datasets, then ramped up significantly. For DS5 and DS6, DeepSeek and ChatGPT performed similarly at 13,115.2 KB and around 132,840 KB respectively, while Claude used the most memory with 13,896 KB on DS5 and 140,861.6 KB on DS6. Heap Sort was relatively memory-efficient on all datasets. All models recorded 0 KB usage on DS1–DS3. For DS4, memory ranged from 764 KB (DeepSeek) to 788 KB (ChatGPT). On DS5, ChatGPT used the least memory (11,196.738 KB), while Claude consumed the most (13,868.8 KB). For DS6, ChatGPT is the most memory-efficient with 132,844.8 KB.

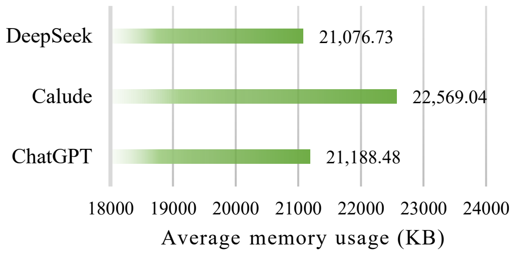

Fig. 3. The average memory usage of models.

Fig. 3 shows the average of memory usage of all algorithms for each model. Across all sorting algorithms and datasets, DeepSeek demonstrated the most memory-efficient behavior, with an average usage of approximately 21,076.73 KB. ChatGPT followed closely, consuming around 21,188.48 KB on average, while Claude showed the highest average memory usage at 22,569.04 KB. These results indicate that DeepSeek is slightly more optimized in terms of memory management, especially on larger datasets. Claude's higher memory footprint may be attributed to additional processing overhead or more complex data structures used in its generated code. Overall, the differences, while small, could become significant in large-scale applications where memory efficiency is critical.

3) Memory Peak (MP) evaluation

Table IV shows the result of models' evaluation in term of average memory peak. In smaller datasets (DS 1–DS 3), all three models performed similarly, with minimal differences in memory peaks, typically under 100 KB. However, as dataset sizes increased, disparities became more apparent. For instance, in DS 5 and DS 6, ChatGPT maintained noticeably lower memory peaks, such as 3560.47 KB for Insertion Sort on DS 5, while Claude used 4342.12 KB, and DeepSeek matched ChatGPT. For Merge Sort on DS 6, ChatGPT's peak was 51,996.18 KB, while Claude's climbed to 59,808.99 KB and DeepSeek followed closely at 59,808.83 KB, again showcasing Claude's higher memory footprint. Similar trends appeared with Quick Sort and Heap Sort, especially in DS 6, where ChatGPT's peek and DeepSeek's peaks remained significantly lower—35,659 KB and 35,646 KB, respectively—compared to Claude's over 43,000 KB in both cases. Overall, this comparison reveals that ChatGPT not only uses less peak memory but does so consistently across sorting techniques, suggesting better scalability and efficiency. DeepSeek generally mirrors ChatGPT's memory behaviour, while Claude exhibits the highest peaks, particularly with large input sizes.

TABLE IV: THE AVERAGE MEMORY PEAK OF MODELS (IN KB)

Algorithm	Dataset (DS)	ChatGPT	Claude Sonnet	DeepSeek
Insertion Sort	DS 1	51.054	51.534	51.054
	DS 2	51.752	52.95	51.762
	DS 3	79.838	88.082	79.872
	DS 4	400.476	479.036	400.502
	DS 5	3560.47	4342.12	3560.47
	DS 6	-	-	-
Merge Sort	DS 1	52.904	52.928	52.934
	DS 2	55.63	55.672	55.702
	DS 3	92.97	93.034	93.034
	DS 4	530.658	609.048	608.898
	DS 5	5213.318	5994.868	5994.71
	DS 6	51996.182	59808.998	59808.834
Quick Sort	DS 1	53.262	52.568	53.254
	DS 2	57.044	56.354	57.032
	DS 3	89.64	89.006	89.646
	DS 4	415.592	453.846	415.586
	DS 5	3576.858	4318.438	3576.874
	DS 6	35659.566	43432.968	35659.546
Heap Sort	DS 1	52.4	51.762	52.384
	DS 2	54.448	54.528	54.44
	DS 3	83.972	91.094	83.978
	DS 4	406.374	483.826	406.376
	DS 5	3567.682	4348.262	3567.69
	DS 6	35646.888	43458.698	35646.9

Fig. 4 shows the average memory peaks of all algorithms for each model. In evaluating peak memory usage, ChatGPT showed the lowest average peak memory consumption at approximately 6,162.99 KB, indicating better memory handling under stress or during recursive and large-scale operations. DeepSeek followed with a slightly higher average of 6,540.06 KB, showing relatively efficient performance but with slightly more fluctuation in memory demand. Claude, however, recorded the highest average peak memory usage at 7,326.94 KB, suggesting a tendency to allocate more memory at the highest points of execution. This pattern was especially noticeable on larger datasets (DS 5 and DS 6), where Claude's memory peaks significantly surpassed those of ChatGPT and DeepSeek. These results highlight ChatGPT's advantage in managing peak memory demands more conservatively, making it potentially more suitable for environments with strict memory constraints.

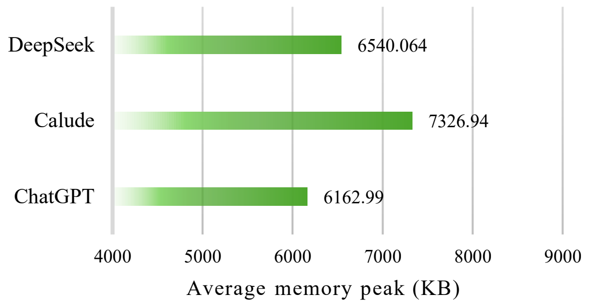

Fig. 4. The average memory peak of models.

4) Logical Size (LS) and Physical Size (PS) evaluation

Table V shows the result of model evaluation in term of logical size and physical size. In terms of logical code size, ChatGPT consistently produces the most concise implementations across all four sorting algorithms, followed closely by DeepSeek, with Claude generating the most verbose code. For instance, ChatGPT's Insertion Sort has a logical size of 1.72 KB, while Claude's is 2.42 KB, and DeepSeek's is 1.93 KB. The same pattern is seen in Merge Sort (2.03 KB for ChatGPT vs. 2.97 KB for Claude and 2.83 KB for DeepSeek), Quick Sort (2.09 KB vs. 3.14 KB and 2.63 KB), and Heap Sort (2.35 KB vs. 3.21 KB and 2.53 KB).

TABLE V: THE AVERAGE LOGICAL SIZE AND THE AVERAGE PHYSICAL SIZE OF MODELS (IN KB)

Algorithm	ChatGPT		Claude Sonnet		DeepSeek	
	LS	PS	LS	PS	LS	PS
Insertion Sort	1.72	4.00	2.42	4.00	1.93	4.00
Merge Sort	2.03	4.00	2.97	4.00	2.83	4.00
Quick Sort	2.09	4.00	3.14	4.00	2.63	4.00
Heap Sort	2.35	4.00	3.21	4.00	2.53	4.00

Despite differences in logical size, the physical size remains constant at 4.00 KB for all models across all algorithms, likely due to file system or formatting constraints. This indicates that while the actual stored size does not vary, ChatGPT achieves better code compactness and efficiency in logical structure, which can impact readability, maintainability, and execution in memory-sensitive environments. Fig. 5 shows the average of logical and physical size of all algorithms for each model. Where ChatGPT achieves the lowest logical size with 2.04, followed by DeepSeek with 2.48, while Claude achieves larger size with 2.93.

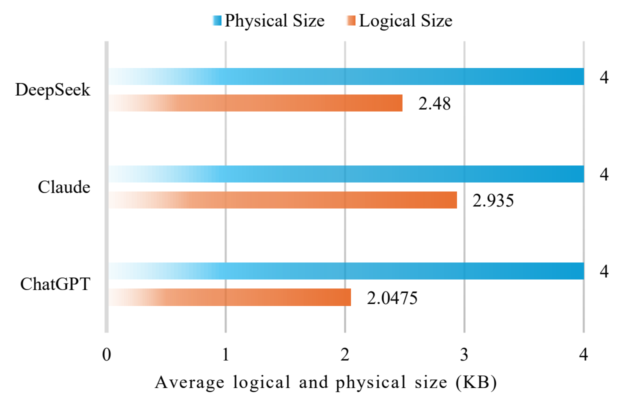

Fig. 5. The average logical size and the average physical size of models.

5) *Readability Score (RS) evaluation*

In this study, Pylint tool is used evaluate the code readability. Pylint is a static code analysis tool designed to evaluate Python code without executing it. It performs in-depth inspection by parsing the code into an abstract syntax tree (AST) and analyzing its structure for potential errors, code style issues, and complexity. Pylint checks for a wide range of issues, including syntax errors, comment, undefined variables, unused imports, bad indentation, naming convention violations, and code smells such as overly complex functions or deeply nested structures. One of Pylint's key features is its scoring mechanism, which rates the quality of the code on a scale from −10.0 to 10.0, where 10.0 indicates a perfect score. This score is based on the number and severity of issues found. Pylint is highly configurable, users can customize which rules are enabled or disabled. It classifies messages into categories such as convention, warning, error, and refactor, helping developers distinguish between minor style preferences and serious bugs.

Table VI shows the result of models evaluation in code readability. The readability of generated sorting algorithms was assessed using a 10-point scale, focusing on clarity, structure, naming conventions, and overall human comprehension. ChatGPT consistently produced highly readable code across all algorithms, scoring between 7.78 and 7.89. Its Merge Sort and Quick Sort implementations were particularly well-structured and easy to follow. Claude, while generally clear, showed more variability, with scores ranging from 5.91 (Heap Sort) to 7.00 (Quick Sort), indicating less consistency in formatting and code legibility. DeepSeek exhibited the widest range in readability, scoring lowest on Insertion Sort (6.25) but highest on Quick Sort (9.29), where its code was notably clean and logically arranged. Overall, ChatGPT maintained superior readability across the board, while DeepSeek showed excellence in specific cases like Quick Sort but inconsistency elsewhere.

TABLE VI: THE READABILITY SCORE OF MODELS

Algorithm	ChatGPT	Claude Sonnet	DeepSeek
Insertion Sort	7.78	6.67	6.25
Merge Sort	7.89	6.4	6.36
Quick Sort	7.86	7	9.29
Heap Sort	7.78	5.91	8.33

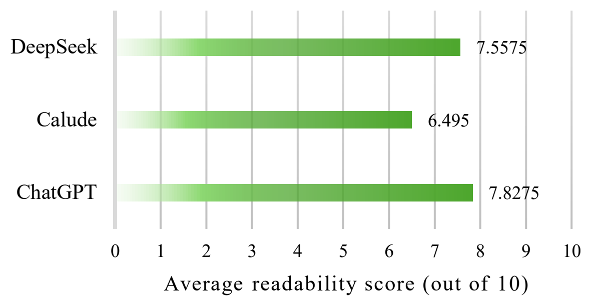

Fig. 6. The average readability score of models.

Fig. 6 shows the average of code readability of all algorithms for each model. Where ChatGPT achieves the highest readability with 7.82, followed by DeepSeek with 7.55, while Claude achieves lowest score with 6.49.

C. *Summary*

To provide a comprehensive comparison of the generated sorting algorithms across the three LLMs, we summarize the average values of all key evaluation metrics in Table VII (rounded to two decimal places). These metrics include execution time, memory usage, peak memory usage, logical and physical code size, and code readability. This consolidated view enables an at-a-glance assessment of the overall efficiency, resource consumption, and code quality produced by each model.

TABLE VII: RESULTS SUMMARY

Metric	ChatGPT GPT-4-turbo	Claude Sonnet 4	DeepSeek-V3
ET (S)	69.77	72.75	70.67
MU (KB)	21,188.48	22,569.04	21,076.73
MPU (KB)	6,163.00	7,326.94	6,540.06
LS (KB)	2.05	2.94	2.48
PS (KB)	4.00	4.00	4.00
RS (0–10)	7.83	6.50	7.56

In terms of execution time, ChatGPT was the fastest on average (69.77 s), followed closely by DeepSeek (70.67 s), while Claude lagged slightly behind (72.75 s). Regarding average memory usage, DeepSeek demonstrated the most efficient memory handling with 21,076.73 KB, marginally outperforming ChatGPT (21,188.48 KB), and significantly better than Claude (22,569.04 KB). A similar trend was observed in peak memory consumption, where ChatGPT recorded the lowest peak at 6,163.00 KB, followed by DeepSeek at 6,540.06 KB, and Claude at 7,326.94 KB. In terms of code size, all models produced scripts with a consistent physical size of 4.00 KB, reflecting uniform disk allocation. However, logical size—which better reflects the actual amount of written content—varied, with ChatGPT producing the most concise code (2.05 KB), followed by DeepSeek (2.48 KB), and Claude generating the most verbose scripts (2.94 KB). Finally, the readability scores, assessed on a 10-point scale, show that ChatGPT's code was the easiest to understand (7.83), with DeepSeek also performing well (7.56), while Claude's outputs were rated least readable (6.50). Overall, ChatGPT led in most categories, with DeepSeek close behind and Claude trailing in efficiency and readability.

V. Conclusion

This comparative study illuminated the varying strengths and limitations of three leading generative AI models—ChatGPT-4-turbo, Claude Sonnet4, and DeepSeek-V3—in generating Python implementations of classical sorting algorithms. ChatGPT consistently demonstrated balanced performance, combining speed, efficient memory usage, and high code readability. DeepSeek, while slightly trailing in execution speed and memory efficiency, produced the most readable and concise code in complex cases like Quick Sort and Heap Sort. Claude, on the other hand, showed a noticeable lag in both memory performance and code quality. These findings highlight that while all three models are capable of generating functional code, ChatGPT currently offers the most well-rounded output for practical programming tasks, especially where performance and readability are equally critical. As generative models continue to evolve, such empirical evaluations will be crucial for developers and researchers seeking optimal AI coding companions.

While this study provides valuable insights into the capabilities of ChatGPT GPT-4-turbo, Claude Sonnet 4, and DeepSeek-V3 for code generation, several limitations should be acknowledged. First, the evaluation focused solely on sorting algorithms implemented in Python, which may not generalize to other algorithmic domains, languages, or code complexities. Additionally, the readability assessment, although quantified on a 10-point scale, remains partially subjective despite efforts to standardize criteria such as formatting, naming conventions, and clarity of logic. Moreover, certain qualitative aspects like error handling of the generated code were not explored in this study.

Future work can address these gaps by expanding the scope of algorithm types (e.g., graph algorithms, dynamic programming), incorporating multiple programming languages, and using automated readability tools or larger expert panels for more objective evaluations. Integrating correctness checking across edge cases, and examining security, maintainability, and cross-platform performance can also provide deeper understanding. Furthermore, applying this comparative framework to newer or fine-tuned models could offer timely benchmarks in this rapidly evolving space.

Conflict of Interest

The authors declare no conflict of interest.

Author Contributions

Alraba'nah and Sleit conducted the research and analyzed the data. Alraba'nah proposed the methodology. Sleit provided guidance and supervision throughout the research process and analysis methodology. Alraba'nah did the experiment and wrote the paper. Qaddara and Hiari reviewed and edited the work. All authors had approved the final version

Funding

This work was supported by Al-Ahliyya Amman University.

References

[1] A. M. Arabiat, "Intelligent model for detecting GAN-generated images based on multi-classifier and advanced data mining techniques," *International Journal of Electrical and Electronic - Engineering & Telecommunications*, vol. 14, no. 3, pp. 147–157, 2025. doi: 10.18178/ijeetc.14.3.147-157

[2] K. Chaitanya and K. J. Rolla, "The evolution and impact of large language models in artificial intelligence," *Algorithms in Advanced Artificial Intelligence*, CRC Press, pp. 410–417, 2024. doi: 10.1201/9781003529231-61

[3] S. Jaradat, N. Acharya, S. Shivshankar, T. I. Alhadidi, amd M. Elhenawy, "AI for data quality auditing: detecting mislabeled work zone crashes using large language models," *Algorithms*, vol. 18, no. 6, 317, 2025. doi: 10.3390/a18060317

[4] Y. Annepaka and P. Pakray, "Large language models: A survey of their development, capabilities, and applications," *Knowledge and Information Systems*, vol. 67, no. 3, pp. 2967–3022, 2025. doi: 10.1007/s10115-024-02310-4

[5] S. Kukreja, T. Kumar, A. Purohit, A. Dasgupta, and D. Guha, "A literature survey on open source large language models," in *Proc. 2024 7th Int. Conf. on Computers in Management and Business*, 2024, pp. 133–143. doi: 10.1145/3647782.3647803

[6] Y. Chang, X. Wang, J. Wang, Y. Wu, L. Yang, K. Zhu, *et al.*, "A survey on evaluation of large language models," *ACM Transactions on Intelligent Systems and Technology*, vol. 15, no. 3, pp. 1–45, 2024. doi: 10.1145/3641289

[7] A. Soliman, S. Shaheen, and M. Hadhoud, "Leveraging pre-trained language models for code generation," *Complex & Intelligent Systems*, vol. 10, no. 3, pp. 3955–3980, 2024. doi: 10.1007/s40747-024-01373-8

[8] L. Caruccio, S. Cirillo, G. Polese, G. Solimando, S. Sundaramurthy, and G. Tortora, "Claude 2.0 large language model: Tackling a real-world classification problem with a new iterative prompt engineering approach," *Intelligent Systems with Applications*, vol. 21, art no. 200336, 2024. doi: 10.1016/j.iswa.2024.200336

[9] K. Bhattacharya, S. Bhattacharya, N. Bhattacharya, and N. Bhattacharya, "DeepSeek versus ChatGPT in surgical practice," *Indian Journal of Surgery*, pp. 1–4, 2025. doi: 10.1007/s12262-025-04368-y

[10] M. Izadi, J. Katzy, T. Van Dam, M. Otten, R. M. Popescu, and A. Van Deursen, "Language models for code completion: A practical evaluation," in *Proc. IEEE/ACM 46th Int. Conf. on Software Engineering*, 2024, pp. 1–13. doi: 10.1145/3597503.3639138

[11] M. M. H. Manik, "ChatGPT vs. DeepSeek: A comparative study on AI-based code generation," arXiv preprint, arXiv:2502.18467, 2025.

[12] Q. Jiang, Z. Gao, and G. E. Karniadakis, "DeepSeek vs. ChatGPT: A comparative study for scientific computing and scientific machine learning tasks," arXiv preprint, arXiv:2502.17764, 2025.

[13] Z. Ji, P. Ma, Z. Li, and S. Wang, "Benchmarking and explaining large language model-based code generation: A causality-centric approach," arXiv preprint, arXiv:2310.06680, 2023.

[14] D. Huang, Y. Qing, W. Shang, H. Cui, and J. M. Zhang, "EffiBench: Benchmarking the efficiency of automatically generated code," *Advances in Neural Information Processing Systems*, vol. 37, pp. 11506–11544, 2024.

[15] Y. Qing, B. Zhu, M. Du, Z. Guo, T. Y. Zhuo, Q. Zhang, and L. A. Tuan, "EffiBench-X: A multi-language benchmark for measuring efficiency of LLM-generated code," arXiv preprint arXiv:2505.13004, 2025.

[16] D. Palla and A. Slaby, "Evaluation of generative AI models in Python code generation: A comparative study," *IEEE Access*, vol 13, pp. 65334–65347, 2025.

[17] D. Zheng, Y. Wang, E. Shi, X. Liu, Y. Ma, H. Zhang, and Z. Zheng, "Top general performance = top domain performance? DomainCodeBench: A multi-domain code generation benchmark," arXiv preprint, arXiv:2412.18573, 2024.

[18] A. Sergeyuk, O. Lvova, S. Titov, A. Serova, F. Bagirov, E. Kirillova, and T. Bryksin, "Reassessing Java code readability models with a human-centered approach," in *Proc. the 32nd IEEE/ACM International Conference on Program Comprehension*, Apr. 2024, pp. 225–235. doi: 10.1145/3643916.3644435

[19] A. Sabra, O. Schmitt, and J. Tyler, "Assessing the quality and security of AI-generated code: A quantitative analysis," arXiv preprint, arXiv:2508.14727, 2025.

[20] K. DePalma, I. Miminoshvili, C. Henselder, K. Moss, and E. A. AlOmar, "Exploring ChatGPT's code refactoring capabilities: An empirical study," *Expert Systems with Applications*, vol. 249, 123602, 2024. doi: 10.1016/j.eswa.2024.123602

[21] R. Phogat, D. Arora, P. S. Mehra, J. Sharma, and D. Chawla, "A comparative study of large language models: ChatGPT, DeepSeek, Claude and Qwen," in *Proc. 2025 3rd Int. Conf. on Device Intelligence, Computing and Communication Technologies*, Mar. 2025, pp. 609–613. doi: 10.1109/DICCT64131.2025.10986449

Yousef Alraba'nah received his B.Sc. degree in software engineering from Zarqa university (Jordan) in June 2012. In February 2013, Yousef obtained fully-funded scholarship from Zarqa University to complete the M.Sc. degree in computer science. He gradułted in June 2015 with excellent degree. He currently works as a lecturer in the Faculty of Information Technology at Al-Ahliyya Amman University, Jordan. His main research interests include distributed systems, networks security, and machine learning.

Azzam Sleit is the former minister of information and communications technology (2013–2015). He is currently working as a professor of computer science, King Abdulla II School for Information Technology, University of Jordan, where he functioned as the dean (2015–2016) and the assistant president/director of the computer center (2007–2009). Dr. Sleit holds the B.Sc, M.Sc. and Ph.D. degrees in computer science. He received his Ph.D. in 1995 from Wayne State University, Michigan. Dr. Sleit was the chief information officer at Hamad Medical/Ministry of Public Health, Qatar. Before joining Hamad Medical, Dr. Sleit was the vice president of Strategic Group & Director of Professional Services of Triada, USA, where he introduced the NGram Technology and Associative Memory Structures. Dr. Sleit authored more than one hundred refereed research papers related to cloud computing, imaging databases, data mining, health and management information systems and software engineering, published in reputable journals and conferences.

Iyas Qaddara is a lecturer in Al-Ahliyya Amman university. He received his first degree in software engineering from Al-Ahliyya Amman University, Jordan in March 2020 and master degree in computer science from Al Balqa Applied University, Jordan in June 2022. His research are of interest include machine learning, parallel computing, computer graphics and virtual reality.

Mohammad Hiari is a lecturer in Al-Ahliyya Amman University. He received his first degree in software engineering from Philadelphia University, Jordan, in August 2004 and master degree in computer science from Al Balqa Applied University, Jordan in February 2016. His research area of interest includes VoIP and cybersecurity data mining and optimization.