# Performance evaluation and analysis with code benchmarking and generative AI

Andrii Berko[1,†], Vladyslav Alieksieiev[1,†] and Artur Dovbysh[1,*,†]

[1] *Lviv Polytechnic National University, 12, S.Bandery str., Lviv, 79014, Ukraine*

**Abstract**

This paper explores code benchmarking techniques and their integration with advanced generative Artificial Intelligence (AI) models, emphasizing the need for continuous performance optimization in data-driven industries. Benchmarking is essential for evaluating and comparing hardware and software performance, identifying bottlenecks, and developing improvement strategies. The study reviews benchmarking theory and practice, detailing key parameters, steps, challenges, and solutions for researchers and practitioners. It examines benchmarks for various sorting algorithms, highlighting the implications for algorithm selection and implementation. Innovatively, the study uses AI models like GPT-3.5-turbo and Gemini 1.5 Pro to analyze algorithmic benchmarks, categorizing efficiency and redefining performance evaluation. The effectiveness of this approach is assessed using the F-score metric, providing insights into AI model performance. The research demonstrates the potential of integrating benchmarking techniques with generative AI, marking a significant advancement in automated code analysis and offering valuable implications for software development and AI applications.

**Keywords**

Benchmarking, performance analysis, generative AI models, Machine Learning, testing, statistics

## 1. Brief overview of performance benchmarking terms

Performance benchmarking is a critical exercise that provides key insights into the operating efficiency and overall competitiveness of a business within an industry. It is the practice of comparing the performance processes, operations, or strategies against those of relevant and comparable items, often referred to as a "leaders", to identify strengths, weaknesses, and opportunities for improvement.

This article section aims to offer a brief overview of the most used terms in the realm of performance benchmarking. For those who are new to the concept, understanding these terms can help in deciphering the complex language of benchmarking, making it easier to implement within a business scenario. The goal here is to broaden your knowledge and

*MoMLeT-2024: 6th International Workshop on Modern Machine Learning Technologies, May, 31 - June, 1, 2024, Lviv-Shatsk, Ukraine*

* Corresponding author.

† These authors contributed equally.

artur.v.dovbysh@lpnu.ua (A. Dovbysh); vladyslav.i.alieksieiev@lpnu.ua (V. Alieksieiev); andrii.y.berko@lpnu.ua (A. Berko)

0009-0004-1912-4887 (A. Dovbysh); 0000-0003-0712-0120 (V. Alieksieiev); 0000-0001-6756-5661 (A. Berko)

equip you with the necessary vocabulary that would facilitate an effective and efficient benchmarking process.

### 1.1. Basic terms and benchmark-related statistical metrics

Starting from the very basics, in computer science, a **benchmark** is the act of running a computer program, a set of programs, or other operations, in order to assess the relative performance of an object, normally by running a number of standard tests and trials against it. Following, computer **performance** is the amount of useful work accomplished by a computer system. Outside of specific contexts, computer performance is estimated in terms of accuracy, efficiency and speed of executing computer program instructions.

Now let's recall primary statistical metrics used in benchmarking: mean, standard deviation and median [1].

To explain why we focus that much on these specific metrics in benchmarking context - consider the following arguments. Firstly, the mean offers a straightforward measure of central tendency, giving a clear indication of the average value within a dataset. This makes it a fundamental metric for understanding typical performance or characteristics. Secondly, standard deviation complements the mean by providing a measure of the spread or dispersion of data points around the mean. It offers insights into the variability within the dataset, crucial for assessing consistency and identifying outliers. Lastly, the median provides a robust alternative to the mean, particularly in datasets with skewed distributions or outliers. It represents the middle value when data points are sorted, making it less sensitive to extreme values compared to the mean. Together, these metrics offer a comprehensive view of data distribution, making them essential tools for benchmark analysis, where accurate comparisons and reliable insights are paramount [1].

**Mean** is the average value of all data points in a dataset. In the realm of code benchmarking, the mean is often used as a representative value to indicate average performance. A lower mean can generally signify better performance. However, it's crucial to interpret the mean alongside other measures, as one abnormally high or low value in the dataset (outlier) could significantly skew the mean, presenting a misleading picture of performance.

**Standard Deviation (StdDev)** deviation measures the dispersion of a dataset relative to its mean. If the standard deviation of a code's performance is low, it means that the results are close to the average, signifying consistent performance. If the standard deviation is high, then the results vary considerably, indicating inconsistent performance. Therefore, in code benchmarking, a lower standard deviation is typically more desirable as it implies reliability and consistency in the code's performance.

**Median** is the middle value in a dataset when the values are arranged in ascending or descending order. In code benchmarking, median values can be particularly relevant because, unlike the mean, they are not affected by outliers. Hence, the median can provide a more accurate representation of the 'typical' performance of a code. This metric can be very useful in situations where there is significant variability, providing a more stable central tendency. Therefore, when we talk about improving performance, we often look at reducing the 'median' time. Basically, the median is the middle element in a sorted dataset [1-3].

In sum, the mean, standard deviation, and median are all statistical measures that provide complementary insight into the structure and tendencies of your data when benchmarking the performance of code. By understanding all three, you can gain a much more comprehensive understanding of your code's efficiency and consistency.

## 1.2. Importance of performance testing and benchmarking and problems

Performance testing and benchmarking are critical aspects in the development process of systems, applications, and networks. They serve fundamental roles in ensuring that these digital assets meet their intended functionality and service quality. Performance testing is a methodical practice that helps developers examine how a system behaves or responds under different circumstances, including workload, operational speed, and stability thresholds. It can help identify bottlenecks, understand system limitations and confirm whether a system can meet its expected performance criteria.

On the other hand, benchmarking is a way to compare the efficiency, speed, and quality of the system, software, or hardware against industry standards or competitor products. Tools and metrics used in benchmarking enable developers and stakeholders to gain valuable comparisons and insights into where their product stands in the competitive landscape. It provides an objective way to identify areas for improvement and explore opportunities for enhancement. Ultimately, both performance testing and benchmarking contribute to delivering a high-quality, efficient, and user-friendly product to end-users.

Performance testing and benchmarking are not just about finding faults and addressing them, they're also about preemptively creating a better user experience. In today's digital age, users have a short tolerance for slow and non-responsive applications. Systems that have been thoroughly performance tested and benchmarked tend to have fast load times, better functionality, and can seamlessly manage high traffic. These characteristics help in boosting user engagement, which in turn positively impacts customer loyalty and overall business success.

Moreover, these processes are also vital in risk mitigation. System crashes and failures resulting from previously undetected issues can lead to considerable financial and reputational damage. Performance testing helps uncover those potential problems before deployment, decreasing the chance of unforeseen system downtime. Benchmarking, in the meanwhile, ensures the product is matching or outperforming market standards, enforcing the reliability and credibility of the product among its users [4, 5].

Furthermore, consistent performance testing and benchmarking allow developers to stay informed about innovative techniques, tools, or methods that leading competitors implement to maintain their exceptional performance levels. It provokes the drive for continuous system improvement in a competitive environment. Ultimately, the combination of performance testing and benchmarking form a basis for delivering superior quality products, which meet user needs and standout in the competitive digital market. These tools offer a clear pathway for product enhancements, as they not only reveal opportunities for improvement but also facilitate a data-driven approach towards seizing these opportunities. In sum, the importance of performance testing and benchmarking cannot be underestimated, given that they are essential contributors to the development of an efficient, competitive, and user-friendly digital product.

## 2. Benchmarks analysis with AI models

One of the main problems I faced and face in modern work with benchmarks is the fact that many people simply do not understand how to work with these metrics.

That is, programmers can easily write and carry out appropriate performance measurements in the form of benchmarks, but they usually do not know how to work with the results. One of the tasks of this paper is to check whether modern models of artificial intelligence can help us with this task.

For this, it was decided to conduct an experiment with a clear goal to verify AI models capabilities in this area [6].

The experiment aims to evaluate ten well-known sorting algorithms, ranging from inefficient ones such as Bubble Sort to more efficient ones like Quick Sort and Tree Sort, among others [7]. This selection is grounded on several factors:

- Each algorithm's clear efficiency rate
- Applicability for benchmarking in various contexts
- Ability to support the required level of abstraction to represent both efficient and inefficient code

Here is a list of selected algorithms with their time complexities [8]:

**Table 1**
**Sorting algorithms with their time complexity**

| Algorithm | Time Complexity | Algorithms Code |
|---|---|---|
| Bubble Sort | $O(n^2)$ | T1 |
| Selection Sort | $O(n^2)$ | T2 |
| Quick Sort | $O(n \log n)$ | T3 |
| Merge Sort | $O(n \log n)$ | T4 |
| Insertion Sort | $O(n^2)$ | T5 |
| Heap Sort | $O(n \log n)$ | T6 |
| Bucket Sort | $O(n + k)$ | T7 |
| Radix Sort | $O(nk)$ | T8 |
| Cube Sort | $O(n \log n)$ | T9 |
| Tree Sort | $O(n \log n)$ | T10 |

Next, we conducted benchmark collections for each algorithm using various datasets, each comprising arrays of different sizes denoted as 'n'. These scenarios included:

1. 100 elements
2. 10000 elements
3. 200000 elements

This approach ensured accurate benchmark measurements across a range of dataset sizes.

Subsequently, we proceeded with the selection of AI models. While numerous open-source and freely available models exist, accessible through platforms like "HuggingFace" [9] our selection was based on the following criteria:

- Availability of documentation
- API accessibility
- Support
- Model size

Thus, we opted for two opposing models from OpenAI and Google: GPT 3.5-turbo and Gemini 1.5 Pro, respectively. It is worth mentioning that we utilized "raw" models without any fine-tuning processes for the benchmark analysis scenario.

Following this, benchmark analyses were conducted with the assistance of both models, wherein each AI assistant classified algorithms as either 'E-efficient' or 'I-inefficient' Additionally, it should be mentioned that the models solely provided "raw" benchmark outputs without specific algorithm names. Consequently, all tested methods were renamed using random identifiers such as T1, T2, ..., T10.

Finally, to evaluate the efficiency of each model in this analysis, we calculated the F-score value and provided feedback [10].

Let's delve into each step in more detail.

### 2.1. Basic terms and benchmark-related statistical metrics

For the task of amassing algorithm benchmarks, we opted to utilize the Benchmark .NET library [9], in conjunction with the corresponding algorithms executed using the C# language. This preference was guided by my past experience as a .NET developer, marking C# as my proficient programming language.

We set up three iterations of tests, each encompassing an assortment of datasets consisting of arrays of varying sizes, represented by the variable 'n'. The three iterations comprise 100 elements, 10,000 elements, and 200,000 elements, respectively.

#### 2.1.1. Benchmark collection on 100 randomly generated elements

In the initial stage of our experimentation, a deliberate decision was made to randomly generate a compact dataset. This dataset, characterized by its diminutive size and inherent randomness, presents a unique testing ground where a wide array of outcomes can manifest. Surprisingly, inefficient algorithms, which may typically be dismissed in larger and more structured datasets due to their suboptimal performance, can paradoxically showcase efficacy within the confines of this modest dataset. The nuanced interplay between algorithm efficiency and dataset size unveils an intriguing phenomenon where inefficiency, under certain circumstances, can translate into heightened performance outcomes on smaller datasets.

One key reason why inefficient algorithms may exhibit favorable performance on smaller datasets is their ability to navigate the reduced complexity and variability inherent in such datasets. Inefficient algorithms, often characterized by their suboptimal computational efficiency or algorithmic design, might inadvertently align with the simplified features and

reduced intricacies of compact datasets. Unlike their performance on larger datasets, where inefficiencies lead to significant performance bottlenecks, these algorithms can capitalize on the streamlined nature of smaller datasets to produce viable results. This phenomenon underscores the importance of contextualizing algorithmic efficiency within the specific characteristics and constraints of the dataset under consideration, highlighting the potential adaptability and unexpected advantages that inefficient algorithms can offer in certain scenarios [11].

Following the execution of the benchmarking procedure, the ensuing data outputs were assembled in Table 1. To visually display the correlation of benchmarks the respective bar chart was built (Figure 1).

**Table 2**
**Sorting algorithms benchmarking results on 100 elements array (us = 0.000001 sec)**

| Method | Array | Mean | StdDev | Median |
|---|---|---|---|---|
| T1 | Int32[100] | 108.34 us | 180.52 us | 24.400 us |
| T2 | Int32[100] | 114.01 us | 189.24 us | 28.050 us |
| T3 | Int32[100] | 35.78 us | 106.47 us | 2.100 us |
| T4 | Int32[100] | 62.91 us | 141.53 us | 18.650 us |
| T5 | Int32[100] | 132.14 us | 224.84 us | 60.300 us |
| T6 | Int32[100] | 42.85 us | 127.07 us | 2.600 us |
| T7 | Int32[100] | 89.07 us | 225.09 us | 18.250 us |
| T8 | Int32[100] | 35.88 us | 106.51 us | 2.000 us |
| T9 | Int32[100] | 428.81 us | 1,081.19 us | 81.950 us |
| T10 | Int32[100] | 80.43 us | 239.06 us | 4.500 us |

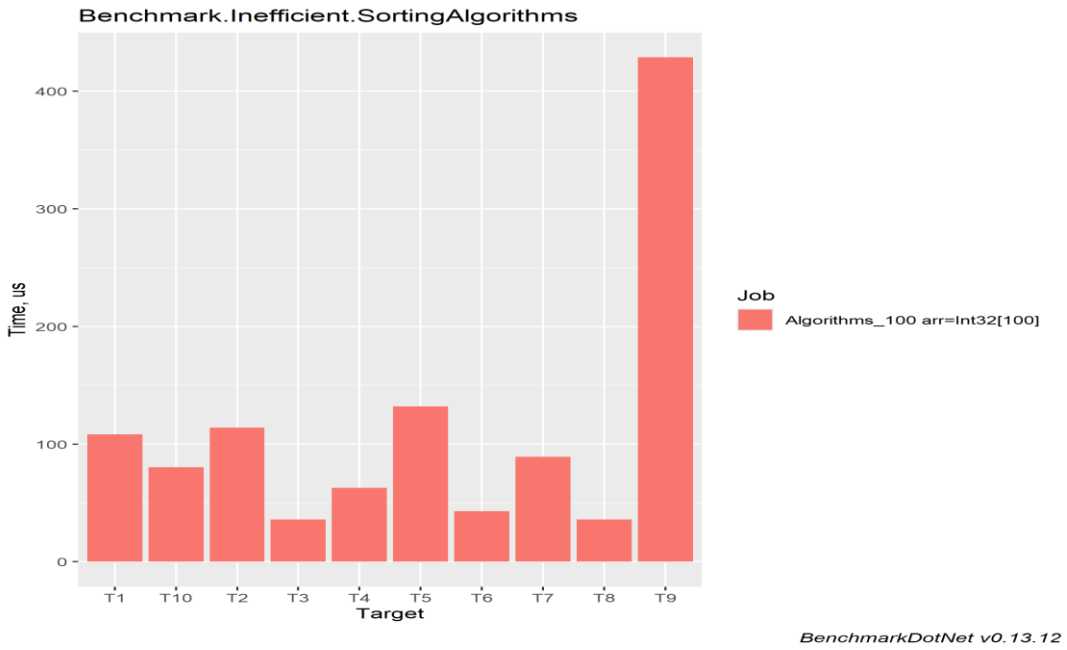


**Figure 1**: Benchmarks Mean outputs for sorting algorithms on 100 elements array

The mean execution times range from 35.78 us to 428.81 us. T3(Quick Sort) has the lowest mean execution time while T9 has the highest mean execution time. That is expected as far as Cube(T9) has its own complexity issues which are very visible on small dataset. It should be more efficient on bigger datasets. The standard deviation of execution times ranges from 106.47 us to 1,081.19 us. T3 has the lowest standard deviation while T9(Cube Sort) has the highest. Interestingly, even algorithms labeled as 'inefficient' demonstrate satisfactory performance with small datasets. Conversely, algorithms expected to be effective yielded contrasting results. Even Bubble Sort(T1) and Selection Sort(T2) were quite efficient on such small distance. They are in the top 5 according to the mean execution time.

### 2.1.2. Benchmark collection on 10000 randomly generated elements

Essentially, we have generated 10,000 random integer values that are distributed over an interval [-2147483648, 2147483647] (int.MinValue, int.MaxValue in .NET respectively) [12]. This scenario aims to simulate a common occurrence characterized by a substantial volume of data, albeit without exceeding the usual workload thresholds for the algorithms involved. After collecting benchmarks on a 10K array of random integer values, we obtained the following outputs:

**Table 3**
**Sorting algorithms benchmarking results on 10000 elements array (us = 0.000001 sec)**

| Method | Array | Mean | StdDev | Median |
|---|---|---|---|---|
| T1 | Int32[10000] | 57,869.7 us | 1,994.9 us | 57,162.45 us |

| | | | | |
|---|---|---|---|---|
| T2 | Int32[10000] | 45,603.0 us | 2,984.7 us | 44,987.60 us |
| T3 | Int32[10000] | 142.4 us | 338.0 us | 29.95 us |
| T4 | Int32[10000] | 2,126.8 us | 2,189.3 us | 1,238.85 us |
| T5 | Int32[10000] | 48,049.3 us | 31,207.5 us | 31,704.05 us |
| T6 | Int32[10000] | 548.0 us | 920.3 us | 132.85 us |
| T7 | Int32[10000] | 974.7 us | 1,517.3 us | 540.00 us |
| T8 | Int32[10000] | 182.6 us | 460.2 us | 38.35 us |
| T9 | Int32[10000] | 32,819.3 us | 7,868.0 us | 29,060.65 us |
| T10 | Int32[10000] | 315.0 us | 646.0 us | 106.15 us |

And visual representation is given via bar chart (Figure 2).

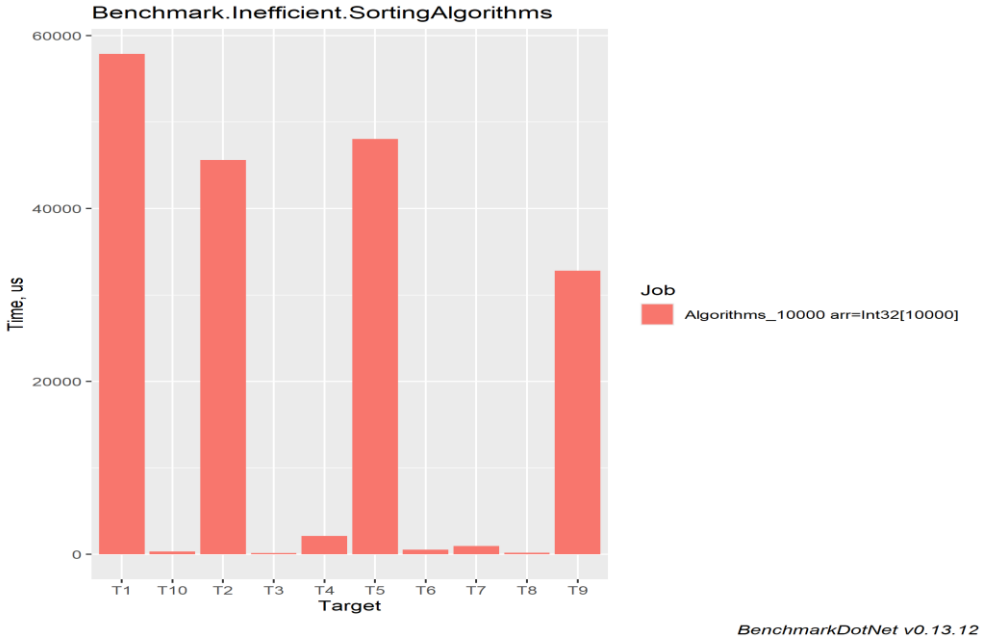


**Figure 2**: Benchmarks Mean outputs for sorting algorithms on 10000 elements array

Starting with the Mean, which is the average time taken, the most efficient method was T3(Quick Sort), taking an average of only 142.4 microseconds (us), followed by T10(Tree Sort) and T8(Radix Sort), taking 315.0 us and 182.6 us respectively. On the end of the spectrum, T1 (Bubble Sort) had the worst performance with its Mean at 57,869.7 us, trailing behind T2(Selection Sort) and T5(Insertion Sort) with means of 45,603.0 us and 48,049.3 us, respectively.

In addition, let's consider Standard Deviation (StdDev) which is an important metric which indicates variance in performance. A higher StdDev means the method's performance varies greatly. T5(Insertion Sort) showed the most inconsistency with a StdDev of 31,207.5 us, significantly higher than the next highest, T4, with a StdDev of 2,189.3 us. On the other hand, T1(Bubble Sort) proved to be the most consistent with a StdDev of only 1,994.9 us.

Persistent correlation errors are discernible even with the utilization of adequately large datasets. Notably, the shortcomings inherent in inefficient algorithms, exemplified by Bubble Sort (T1), Selection Sort (T2), and T5 (Insertion Sort), manifest conspicuously.

Conversely, proficient algorithms exhibit a discernible divergence in execution times, as indicated by the Mean execution time metric. The performance of Cube Sort (T9) presents an intriguing aspect, demonstrating a noteworthy enhancement in efficiency, albeit within a conditional context vis-à-vis its competitive counterparts. Further investigation into the underlying factors contributing to these observed efficiencies could provide valuable insights into algorithmic optimizations and computational efficiency.

Finally, let's take a look of the biggest dataset results.

### 2.1.3. Benchmark collection on 200000 randomly generated elements

In this particular scenario, a substantial workload is anticipated for each algorithm under examination. To provide context, the generation of 200,000 random integers imposes a significant computational burden, as evidenced by the duration of approximately 30 minutes per benchmark execution on a laptop equipped with 64GB of RAM and a highly efficient CPU architecture. It is crucial to note that this time interval is distinct from the sorting algorithm's execution time. The forthcoming results are anticipated with great interest, as they will shed light on the performance characteristics of the algorithms under such demanding computational conditions.

After running benchmarks on an enormous dataset comprising 200K elements of randomly generated integers, we acquired the outputs presented in Table 4 and resulting the following bar chart presentation (Figure 3).

From the benchmark results table, we can see that the mean execution times for the different methods vary significantly.

T3(Quick Sort) is the most efficient method with a mean execution time of only 291.7 us. This is followed by T8(Radix Sort) with a mean execution time of 409.9 us and T6(Heap Sort) with a mean execution time of 3,450.2 us.

On the other hand, T1(Bubble Sort) has the highest mean execution time of 17,480,568.3 us, followed by T2(Selection Sort) and T5(Insertion Sort) with a mean execution time of 8,694,258.2 us. It was quite expected result, because these algorithms are we well knows inefficient sorting algorithms.

When looking at the standard deviation (StdDev) values, we can see that T3(Quick Sort) has the lowest standard deviation of 219.3 us, indicating that the execution times for this method are consistent. T8(Radix Sort) and T10(Tree Sort) also has a relatively low standard deviation of 507.2 us. These results prove that these algorithms are not only considered to be most efficient ones, but they actually are.

**Table 4**
**Sorting algorithms benchmarking results on 200000 elements array (us = 0.000001 sec)**

| Method | Array | Mean | StdDev | Median |
|---|---|---|---|---|
| T1 | Int32[200000] | 17,480,568.3 us | 1,178,671.9 us | 17,234,825.9 us |
| T2 | Int32[200000] | 11,477,353.2 us | 567,577.3 us | 11,348,932.8 us |
| T3 | Int32[200000] | 291.7 us | 219.3 us | 214.6 us |
| T4 | Int32[200000] | 29,027.5 us | 9,875.0 us | 28,415.8 us |

| | | | | |
|---|---|---|---|---|
| T5 | Int32[200000] | 8,694,258.2 us | 1,337,944.7 us | 8,410,619.9 us |
| T6 | Int32[200000] | 3,450.2 us | 784.9 us | 3,237.9 us |
| T7 | Int32[200000] | 30,471.5 us | 3,352.4 us | 30,000.7 us |
| T8 | Int32[200000] | 409.9 us | 507.2 us | 244.0 us |
| T9 | Int32[200000] | 780,770.4 us | 124,539.2 us | 819,716.8 us |
| T10 | Int32[200000] | 1,754.2 us | 801.1 us | 1,375.3 us |

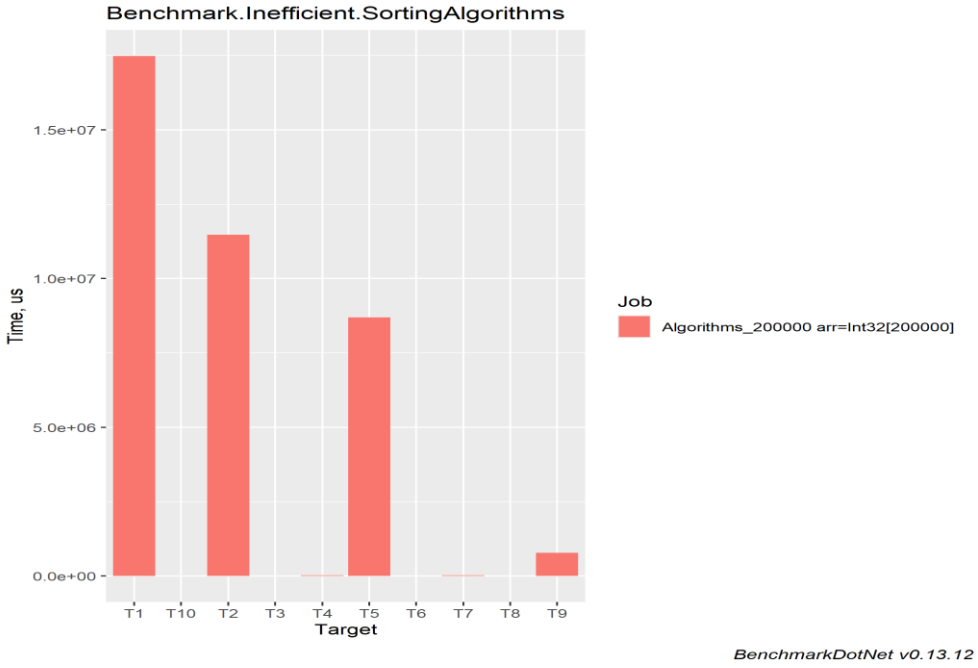


**Figure 3**: Benchmarks Mean outputs for sorting algorithms on 200000 elements array

In contrast, T1(Bubble Sort) has a very high standard deviation of 1,178,671.9 us, suggesting that the execution times for this method vary significantly.

Overall, based on the benchmark results, T3, T8, and T6 appear to be the most efficient methods in terms of mean, standard deviation, and median execution times. These methods consistently perform well in terms of speed and variability in execution times.

In addition, T9(Cube Sort) finally showed its capabilities. This algorithm is among top performer algorithms and proved itself as an efficient one.

On the other hand, methods like T1 and T5 appear to be less efficient, with higher mean execution times and standard deviations. This suggests that these methods may not be as reliable or consistent in their performance compared to the more efficient methods.

Now let's check whether modern AI models are able to process "raw" benchmark data. Since not all people know statistics, and even fewer people know how to analyze the data that we collect when measuring benchmarks - the help of AI models will be very appropriate. The task of the next section is to test the capabilities of AI models in this direction.

## 2.2. AI-based analysis of performance statistical data

First, let me briefly describe the AI models we have chosen for this experiment.

OpenAI's GPT-3.5-turbo (GPT - Generalized Pre-trained Transformer) is part of the third generation of large transformer-based language models with 175 billion parameters. It adheres to the autoregressive language model, which can process and generate text based on the preceding context.

GPT-3.5-turbo is often referred to as a "chat model", as it is formatted to generate responses in a conversational context. The model is capable of composing emails, writing code, answering questions, creating written content, tutoring in a variety of subjects, translating languages, simulating characters for video games, and much more.

To interact with GPT-3.5-turbo, one sends a series of messages, and the model returns a model-generated message. A conversation typically begins with a system message which sets the behavior of the assistant, followed by alternating user and assistant messages. The model completes its predictive task based on the entire conversation history [13, 14, 15].

GPT's contender is this battle is Google's Gemini 1.5 Pro model.
The "Gemini AI" model, also known as Bard, refers to an experimental conversational AI developed by Google. Gemini is designed to provide users with high-quality, contextually relevant information and insights. It draws from Google's vast data resources, enhancing user interactions by delivering relevant and precise answers.

Gemini aims to leverage Google's LaMDA (Language Model for Dialogue Applications) technology, featuring capabilities in understanding and generating human-like text-based responses. This positions it as a competitor to other advanced AI models like OpenAI's ChatGPT.

Gemini is part of Google's broader efforts to integrate advanced AI into everyday user interactions, helping to simplify information retrieval, enhance learning, and provide a more interactive and intuitive user experience through natural language processing. The development and refinement of Gemini emphasizes Google's commitment to leading in AI-driven technologies and their applications across various sectors [16].

In short, these are the most modern generative AI models that exist in public access and can be used for various purposes.

Additionally, based on the AI models documentations [13, 14, 15, 16, 17], let's review face-to-face comparison of these models (see Table 5).

The comparison between the OpenAI and Google models unveils distinct characteristics and functionalities that distinguish these two prominent AI providers in the realm of natural language processing. OpenAI's model, boasting an input context window size of 4096 tokens and a corresponding maximum output token capacity of 4096 tokens, positions itself as a contender in the field with a balanced focus on contextual understanding and output generation. In contrast, Google's offering showcases a significantly larger input context window of 32.8K tokens, enhancing its capacity to process extensive textual data, coupled with a maximum output token allowance of 8192 tokens, thereby catering to the requirements of complex text generation tasks.

**Table 5**
**GPT 3.5-turbo versus Gemini 1.5 Pro models comparison**

| Characteristic | GPT 3.5 Turbo | Gemini 1.5 Pro |
|---|---|---|
| Model provider | Open AI | Google |
| Input context window | 4096 tokens | 32.8K tokens |
| Maximum output tokens | 4096 tokens | 8192 tokens |
| Release date | November 28th, 2022 | December 13th, 2023 |
| Input pricing | $ 0.5 per 1M tokens | $ 0.13 per 1M characters |
| Output pricing | $ 1.5 per 1M tokens | $ 0.38 per 1M characters |

Furthermore, the release dates of these models shed light on their respective timelines of entry into the market. OpenAI introduced its model earlier on November 28th, 2022, providing an earlier foothold in the evolving landscape of AI-driven solutions for natural language processing. On the other hand, Google unveiled its model at a later date on December 13th, 2023, indicating a subsequent entrance into the competitive arena of AI model deployment. This temporal discrepancy may influence user preferences based on the models' maturity, refinement, and adaptability to evolving industry standards and demands.

In terms of pricing structure, OpenAI and Google present divergent approaches in their fee arrangements for input and output processing. OpenAI offers input pricing at $0.5 per 1 million tokens, reflecting a cost-effective model for users aiming to leverage token-based input mechanisms. Conversely, Google adopts a pricing scheme of $0.13 per 1 million characters for input data, catering to users inclined towards character-based input strategies. Similarly, the output pricing contrasts between the providers, with OpenAI charging $1.5 per 1 million tokens and Google offering output processing at a rate of $0.38 per 1 million characters. These pricing differentials underscore the nuanced considerations that users must weigh when selecting a model provider based on their specific budgetary constraints and processing requirements [19].

### 2.2.1. GPT 3.5 Turbo

We've attached the benchmarks for the models' context using CSV files and outlined our target scenarios. To enhance the accuracy and intrigue of our experiment, we decided to task the AI model with classification based on:

1. Benchmarks outputs for 100 elements - Algorithms_Benchmarks_100.csv
2. Benchmarks outputs for 10,000 elements - Algorithms_Benchmarks_10000.csv
3. Benchmarks outputs for 200,000 elements - Algorithms_Benchmarks_200000.csv
4. Utilizing all available outputs

***Note***. For those, who want to repeat the experiment all mentioned above files could be found by the following link: https://github.com/d0vbysh/MoMLeTDS2024_ModelsContext

The directive given to the model was: "Having attached benchmark files for ten methods representing different algorithms to this chat context, could you please classify these methods as 'I-inefficient' or 'E-efficient'? Please provide the answer in a simple table format

like T1 - I, T2 - E, etc. Additionally, I kindly ask you to perform this evaluation four times—once using only the 'Algorithms_Benchmarks_100.csv', once using 'Algorithms_Benchmarks_10000.csv', once using 'Algorithms_Benchmarks_200000.csv', and once using all available files in the context."

The output from the model was as follows:

"Based on the provided data, we can classify the algorithms as:"

**Table 6**
**GPT 3.5-turbo benchmark analysis outputs**

| Method | Answer – 100 | Answer - 10K | Answer - 200K | Answer with full context |
|---|---|---|---|---|
| T1 | I | I | I | I |
| T2 | I | E | E | I |
| T3 | E | I | I | I |
| T4 | I | E | E | E |
| T5 | I | E | E | E |
| T6 | E | E | E | E |
| T7 | I | E | E | I |
| T8 | E | E | E | E |
| T9 | I | I | I | I |
| T10 | I | E | E | E |

As you can see, the GPT 3.5 Turbo produced quite accurate classification results with a small error in the answers.

Let's calculate the F-score for the model classification results.

Firstly, the F-score, also known as the F1-score or F-measure, is a measure of a model's accuracy that considers both precision(P) and recall(R). It's used to express the balance between precision (which is the proportion of true positive results among positive results predicted by a model) and recall (which is the ability of a model to find all the relevant cases in a dataset). The F-score is particularly useful when you want to compare two classifiers directly, as it combines these two metrics into a single number [20, 21].

The formula for the F-score is:

$$F1 = 2\frac{P \cdot R}{P+R},$$

where:

- Precision is the number of true positive results divided by the number of all positive results, including those not identified correctly. Its formula is
$$P = \frac{TP}{(TP+FP)}.$$

- Recall is the number of true positive results divided by the number of all samples that should have been identified as positive. Its formula is
$$R = \frac{TP}{(TP+FN)}.$$

In these formulas:

- $TP$ is the number of true positives (the number of items correctly identified as positive),
- $FP$ is t he number of false positives (items incorrectly identified as positive),
- $FN$ is the number of false negatives (items incorrectly identified as negative).

Let's proceed with the simple example of F score calculation for the GPT model classification results based on 100 elements array benchmarks.

According to the model's outputs we have two correctly identified Inefficient algorithms (TN), and the same amount of correctly classified Efficient algorithms (TP). On the other hand, we have five wrongly recognized algorithms as Inefficient (FN) and one inaccurate classification of algorithms as Efficient (FP). Following these numbers, let's get P and R:

$$P = \frac{TP}{TP + FP} = \frac{2}{2 + 1} = 0.6667;\ R = \frac{TP}{TP + FN} = \frac{2}{2 + 5} = 0.2857$$

And having P and R we can calculate the F score value:

$$F1 = 2\frac{PR}{P+R} = 0.4.$$

Having all these data in place, we received F score for the GPT 3.5 Turbo model (see Table 7).

**Table 7**
**GPT 3.5-turbo F score**

| Metric | Answer – 100 | Answer - 10K | Answer - 200K | Answer with full context |
|---|---|---|---|---|
| P | 0.6667 | 0.7143 | 0.8571 | 1 |
| R | 0.2857 | 0.7143 | 0.8571 | 0.6667 |
| F score | 0.4 | 0.7143 | 0.8571 | 0.8 |

The model is performing very well in correctly identifying both positive and negative instances, with few false positives and false negatives. This high F-score in "All Context" scenario suggests strong overall performance and reliability of the model. Only in the case of 100 elements array, model was not that efficient. I assume the reason for that is that the dataset is quite tiny. In addition, we must to recall that models process only raw benchmarking data without any additional context of it. I assume that such results are really nice and definitely this AI model must be considered as a solid option to be used in scenario of performance analysis.

Let's evaluate Gemini 1.5 Pro model to have some data for efficiency comparison.

### 2.2.2. Gemini 1.5 Pro

Repeating the whole experiment with the GPT's opponent we have obtained the following results (Table 8) and it's F scores (Table 9).

The Gemini model demonstrates commendable performance, particularly evident in scenarios with smaller data sets. Notably, the model exhibits exceptional accuracy when operating on a dataset containing 10,000 elements. However, as the dataset size increases to 200,000 elements, encompassing all execution results, the accuracy diminishes significantly. Despite this drawback, Gemini endeavors to elucidate its decision-making

process in the realm of classification. Through a comprehensive analysis of the available data, the model provides insightful arguments: “Algorithms characterized by low standard deviation, such as T3 and T8, are deemed preferable when consistency in performance is paramount. On the other hand, algorithms like T4 and T7 may find applicability in niche scenarios where the input size remains relatively modest. The variable efficiency of T9 renders it a specialized algorithm, potentially well-suited for specific data types or distributions. Conversely, algorithms such as T1, T2, and T5 are advised against for large datasets or time-critical applications due to their persistent inefficiency.”

**Table 8**
**Gemini 1.5 Pro benchmark analysis outputs**

| Method | Answer – 100 | Answer - 10K | Answer - 200K | Answer with full context |
|---|---|---|---|---|
| T1 | E | I | I | I |
| T2 | I | I | I | I |
| T3 | E | I | I | I |
| T4 | E | E | E | E |
| T5 | I | E | E | E |
| T6 | E | E | E | E |
| T7 | I | E | E | E |
| T8 | E | E | E | E |
| T9 | I | E | I | I |
| T10 | I | E | E | E |

**Table 9**
**Gemini 1.5 Pro F score**

| Metric | Answer – 100 | Answer - 10K | Answer - 200K | Answer with full context |
|---|---|---|---|---|
| P | 0.6 | 0.8571 | 0.8333 | 0.8333 |
| R | 0.4286 | 0.8571 | 0.7143 | 0.7143 |
| F score | 0.5 | 0.8571 | 0.7692 | 0.7692 |

This detailed examination of various benchmarks exemplifies the model's capacity to generate insights without necessitating any fine-tuning. Such capabilities bode well for the continued evolution and enhancement of the Gemini model within the scientific community.

Further expansion of the text could delve into the implications of these findings for the field of machine learning, the significance of accurate classification algorithms in various industries, and potential avenues for future research to optimize the performance of models like Gemini in diverse real-world applications.

## 3. Summary

Drawing on rigorous experiments, we conclude that artificial intelligence models can be considered viable tools to assist with our performance testing routines. Both models deliver impressive and accurate results, without the need for model fine-tuning or consideration of additional context pertaining to algorithm characteristics - they work solely using raw benchmark data. And this is great. I am sure it field to be explored and developed. Introducing fine-tuning for models will be a game changer. However, it is out of scope for this paper work.

In my professional experience, not all developers fully grasp how to implement this method, though a notable majority are certainly employing it. Gemini 1.5 demonstrates greater precision over a broader range, whereas GPT3.5-turbo is particularly proficient at processing smaller datasets.

No matter which model you opt for, out investigation strongly posits that with thorough and patient preparation, we have the potential to achieve even greater results in terms of code performance analysis.

Potentially, we can achieve autodetection of the problematic code, based on trained model experience. This underlines a new possibility in how we approach and comprehend the efficiency of coding.

The future of performance testing and statistics with AI models holds significant promise and potential for innovation in the field of technology. As artificial intelligence continues to advance, the integration of AI models in performance testing is poised to revolutionize the analysis and optimization of code performance. By harnessing the power of AI algorithms, developers can expect a paradigm shift in how performance testing is conducted, moving towards more efficient and automated processes.

One key aspect that the future holds for AI models in performance testing is the utilization of machine learning techniques to enhance the accuracy and reliability of testing results. Through the application of sophisticated algorithms, AI models can autonomously detect performance bottlenecks, identify areas of inefficiency, and provide actionable insights for optimization. This capability not only streamlines the testing process but also augments the overall quality of code performance analysis.

Furthermore, the future trajectory of AI models in performance testing envisions the potential for autodetection of problematic code segments based on the accumulated experience and training of the models. By leveraging this predictive capacity, developers can proactively address coding inefficiencies, enhance code quality, and ultimately improve the overall performance of software applications. This transformative approach heralds a new era in software development, where AI-driven insights play a pivotal role in optimizing code efficiency and performance.

As this field continues to evolve, the collaborative efforts of researchers, developers, and data scientists will be crucial in unlocking the full potential of AI models in performance testing and statistics. By fostering interdisciplinary collaborations and embracing innovative technologies, the future landscape of performance testing with AI models holds a wealth of possibilities for advancing the efficiency, reliability, and scalability of software systems.

The objectives set for this paper have been effectively achieved, delivering clear and promising results. As we conclude this project, we find ourselves with a deeper understanding and vision for further investigation. We eagerly anticipate what the new phase of our investigation holds as we endeavor to uncover more insights in this captivating and complex field. We are confident that our ongoing work will extend the knowledge we have established within this fascinating topic.

## References


[1] Andrey Akinshin, Pro .NET Benchmarking: The Art of Performance Measurement 1st ed, 2019, ISBN-10 1484249402: 97-345.

[2] Thomas Nield, Essential Math for Data Science. Take Control of Your Data with Fundamental Linear Algebra, Probability, and Statistics. 1st Edition, 2019, ISBN-13 9781098102937: 42-271.

[3] Samuel Kounev, Klaus-Dieter Lange, Jóakim von Kistowski, Systems Benchmarking: For Scientists and Engineers 1st. Edition, 2020, ISBN-13 978-3030417048

[4] Brendan Gregg, Systems Performance (Addison-Wesley Professional Computing Series) 2nd Edition, 2020, ISBN-13 978-0-13-682015-4.

[5] K. Singh, K. Singh Dhindsa, B. Bhushan, Performance Analysis of Agent Based Distributed Defense Mechanisms Against DDOS Attacks. Intern. J. of Computing, 17(1), 15-24. (2018). URL: https://doi.org/10.47839/ijc.17.1.945

[6] Olivier Caelen, Marie-alice Blete, Developing Apps With GPT-4 and ChatGPT: Build Intelligent Chatbots, Content Generators, and More 1st Edition, 2023, ISBN-13 978-1098152482

[7] Aditya Bhargava, Grokking Algorithms, Second Edition 2nd Edition, 2024, ISBN-13 978-1633438538.

[8] Jay Wengrow, A Common-Sense Guide to Data Structures and Algorithms, 2020, ISBN-13 9781680507225

[9] HugginFace forum. URL: https://huggingface.co/posts

[10] Benchmark.NET documentation.URL: https://benchmarkdotnet.org

[11] George Heineman, Learning Algorithms: A Programmer's Guide to Writing Better Code. 1st Ed, ISBN-13 9781492091066

[12] .NET docs. URL: https://github.com/dotnet/docs

[13] Ali Aminian and Alex Xu, Machine Learning System Design Interview, 2023, ISBN-13 978-1736049129

[14] Open AI platform documentation. URL: https://platform.openai.com/docs/overview

[15] Open AI API documentation. URL: https://platform.openai.com/docs/api-reference

[16] Open AI Chat GPT documentation. URL: https://openai.com/chatgpt

[17] Gemini API documentation. URL: https://ai.google.dev/gemini-api/docs

[18] Gemini documentation. URL: https://deepmind.google/technologies/gemini/#gemini-1.5

[19] AI models analysis. URL: https://artificialanalysis.ai/models

[20] Nathalia Nascimento, Paulo Alencar, Donald Cowan "Comparing Software Developers with ChatGPT: An Empirical Investigation" arXiv, May. 25, 2023, doi: https://doi.org/10.48550/arXiv.2305.11837

[21] I. Gorbenko, A. Kuznetsov, Y. Gorbenko, S. Vdovenko, V. Tymchenko, and M. Lutsenko, Studies On Statistical Analysis and Performance Evaluation for Some Stream Ciphers,

Intern. J. of Computing, 18(1), 82-88. (2019). URL: https://doi.org/10.47839/ijc.18.1.1277