DocAlchemy (turning code into gold)

Shuni Bickel
316123207
shunibickel@mail.tau.ac.il

Eyal Grinberg
207129792
eyalgrinberg@mail.tau.ac.il

Abstract

Large Language Models (LLMs) have significantly advanced natural language processing tasks, including code generation and understanding. Benchmarks have been crucial in evaluating and improving these models. While code generation benchmarks have driven progress, enhancing LLMs' abilities to explain code and maintain readability is equally important. We propose a novel benchmark to assess docstring generation. Our results highlight the limitations of current models and demonstrate the potential of our benchmark to guide future research in docstring generation.

1 Introduction

In the past few years, LLMs have leaped forward, demonstrating remarkable capabilities in various NLP tasks, including code generation and understanding. One driving force for these advancements is well-designed benchmarks that evaluate model performance and identify areas for improvement. Code generation benchmarks, such as HumanEval (Chen et al., 2021), have been instrumental in pushing the boundaries of what these models can achieve.

However, to maximize the utility of LLMs as coding assistants, it is equally important to enhance their abilities to generate comprehensive and accurate docstrings. Docstrings play a crucial role in explaining code, ensuring readability, and maintaining overall code quality. Despite the progress in code generation, the evaluation of docstring generation remains underexplored. One main issue with evaluating docstring generation is that it is harder to automatically evaluate in comparison to generated code.

In this paper, we propose a novel benchmark specifically designed to assess the effectiveness of docstring generation by LLMs. Our benchmark incorporates a diverse range of evaluation metrics, providing a robust and comprehensive assessment of model performance. It was crucial to ensure that the benchmark is practical for improving models, challenging for the models to perform well on, and enables automatic assessment. Through our experimental results, we reveal the limitations of current state-of-the-art models and demonstrate the potential of our proposed benchmark to drive future research and development in docstring generation. By addressing these gaps, we aim to contribute to the ongoing efforts to enhance LLMs' capabilities as comprehensive coding assistants.

2 Related Work

The importance of benchmarks in evaluating and advancing Natural Language Processing (NLP) tasks is well-established. The article "Research community dynamics behind popular AI benchmarks" (Martínez-Plumed et al., 2021) published in Nature Machine Intelligence highlights how well-defined benchmarks foster collaboration and accelerate progress within research communities. A crucial aspect of an effective benchmark is its ability to challenge models and prevent them from achieving perfect scores too easily.

In the context of code documentation generation, prior work by Dvivedi et al. in "A comparative analysis of Large Language Models for code documentation generation" (Dvivedi et al., 2023) explores the potential of LLMs for this task. Their work provides valuable insights, but manual evaluation of docstring generations, while thorough, becomes impractical for the purpose of creating a benchmark. The sheer volume of docstring generations needed for statistically significant results makes manual assessment infeasible. This is where automatic evaluation comes into play. Benchmarks like SWE-Bench (Jimenez et al., 2024), which evaluates bug-fixing capabilities by

automatically comparing generated code patches to actual solutions using unit-tests, demonstrate the effectiveness of automation.

Our work builds upon these contributions. We acknowledge the importance of evaluating LLM performance in docstring generation, but we propose a different approach that addresses the limitations mentioned above. Similar to the SWE-Bench benchmark, we aim to develop a benchmark that facilitates automatic measurement of docstring quality. However, our approach goes beyond basic automation by incorporating the growing capabilities of LLMs themselves. We propose an ensemble of evaluation methods, including both traditional automated tests and tests incorporating dynamic generations using LLMs. This approach ensures comprehensiveness and leverages the strengths of both established techniques and cutting-edge advancements in NLP.

3 Methodology

This section describes the methodology employed to investigate a potential benchmarking method for evaluating Large Language Models (LLMs) in their ability to generate docstrings according to Google conventions. The methodology involved several key stages:

3.1 Dataset Creation

- A dataset of 50 code chunks was manually created (to avoid contamination as much as possible), encompassing various code types:
  - 20 'basic' python functions, involving string manipulation, arithmetics, lists etc.
  - 10 recursion python functions.
  - Functions involving common libraries - 5 questions using Pandas and 5 questions using Numpy.
  - 10 Python classes with varying complexity.
- Corresponding docstrings were written for each code chunk as a "golden standard example", adhering to Google's docstring formatting guidelines.
- The code chunks and docstrings were stored in designated Jupyter Notebook (code_chunks.ipynb) and a CSV file (Benchmark_database.csv).

3.2 Baseline

To establish a baseline for docstring generation performance, we've chosen a pre-trained model based on the T5-base architecture specifically trained for the task of generating Python code documentation[1]. While it may be considered a less complex model compared to some models we will evaluate, its targeted pre-training for our specific objective makes it a suitable starting point for evaluation.
The generated docstrings were assessed, and the T5 model was found to perform poorly, failing to produce docstrings in the desired format, as expected.

3.3 Docstring Generation using chosen LLMs

To explore the trade-off between cost and performance, we selected three LLMs for evaluation: GPT-3.5 Turbo, Gemini 1.0 Pro, and Claude instant 1. These models represent a balance between affordability and established capabilities, while newer options like GPT-4 and Gemini-Ultra were not considered due to cost constraints.

- The generate_docstrings.ipynb notebook employed the langchain library to interact with the chosen LLMs' APIs for docstring generation.
- Recognizing the inherent differences between function and class docstrings, prompts were specifically tailored for both types to give a helpful context to each generation type.
- Gemini, due to limitations in its system prompt option, required a concatenated system and user prompt for optimal performance. This is documented within the notebook.
- The generated docstrings for all code chunks were stored in data_full_docstrings_generated.csv.

3.4 Evaluation

Our design principles for the docstring benchmark focused on two key aspects:

[1] https://huggingface.co/SEBIS/code_trans_t5_base_code_documentation_generation_python

- **Challenging LLMs:** The benchmark should push the capabilities of evaluated LLMs. If all models excel, it won't effectively guide LLM improvement.

- **Automated Evaluation:** We aimed for metrics that code pipelines could readily assess. Manual evaluation for each iteration and model would be impractical and hinder the benchmark's purpose.

Guided by these principles, we selected the following metrics to evaluate the quality of generated docstrings:

3.4.1 Comparative Evaluation Metrics

To evaluate the generated docstrings against the provided golden examples, we employed four algorithmic metrics: ROUGE-1, ROUGE-2, ROUGE-L F1-scores (combining precision and recall, (Lin, 2004)), and BLEU (Papineni et al., 2002) score.

3.4.2 LLM Evaluation Metrics

To assess qualities beyond the scope of traditional metrics, we leveraged GPT-4's capabilities as an LLM (we've used gpt-4-1106-preview). We presented the model with pairs of functions and their corresponding generated docstrings, prompting it to evaluate each docstring based on five key criteria, based on the criteria defined in "A comparative analysis of Large Language Models for code documentation generation":

- Accuracy: How well does the docstring describe the code's functionality?

- Completeness: Does the docstring include all necessary information about the code?

- Relevance: Does the docstring stay focused and avoid irrelevant information?

- Understandability: How clear and easy is it to comprehend the docstring?

- Readability: Does the docstring adhere to formatting and structure conventions?

GPT-4 then provided a Python list containing a quality score for each of these criteria.

3.4.3 Formatting Check Metric

We assessed the adherence of generated docstrings to formatting conventions using the ruff[2] formatter. Google's Python docstring conventions served as the foundation for our configuration, covering most PEP 257 formatting rules and additional useful checks (e.g., parameter documentation). However, we configured (A.1) ruff to disregard specific checks that produced unstable outputs in our pipeline, such as rules related to the docstring's starting position on the first or second line (likely due to the model or processing steps). Since ruff lacks a Python/Jupyter-friendly API, we employed a command-line interface (CLI) to analyze each model's 50 code-docstring pairs stored in separate Jupyter Notebooks. The results were saved to text files, which were then processed to incorporate a new column in our data frame containing the number of ruff errors for each model, corresponding to the respective functions.

3.4.4 Unit Test Evaluation

To further evaluate the docstrings' effectiveness, the following process was implemented:

- We designed unit tests for each of the 50 manually written functions and classes.

- Subsequently, GPT-4 (gpt-4-1106-preview) was tasked with generating code implementations based solely on the generated docstrings for each function class.

- These generated code snippets were then executed against the pre-defined unit tests.

- The resulting error rates, captured for each generated function and corresponding model, served as an additional quality metric.

Our rationale behind this approach is that GPT-4's well-established code generation capabilities allow us to assess the quality of the docstrings by measuring how effectively they guide the generation of functional code.

3.5 Composite Score

As a part of our approach to evaluating the generated docstrings using all of the metrics and methods presented above, we suggested an aggregated

[2] https://docs.astral.sh/ruff/

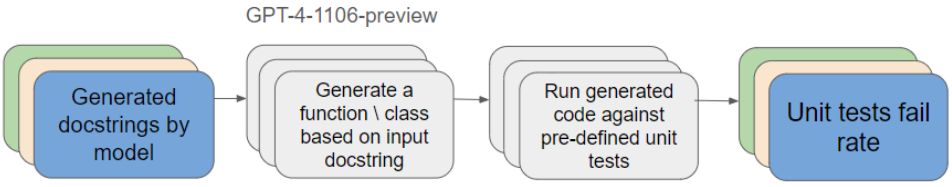

Figure 1: Unit test fail rate evaluation flow illustration

method to create a single composite score to measure the general quality of the results. The base principles that lead us while creating this metric were:

- **Weighted Means for Similar Metrics:** We calculated a weighted mean for comparable metrics like ROUGE and BLEU scores, as they evaluate similarity to an example in similar ways, and another weighted mean for the five scores obtained from GPT-4 prompts (Accuracy, Completeness, Relevance, Understandability, Readability). Within each group, weights were adjusted based on their importance to core docstring functionality. Readability, for instance, received a lower weight compared to metrics like accuracy or completeness, as it's less important for functionality and is partially assessed by the ruff formatting check.

- **Prioritizing LLM Evaluation Metrics:** A weighted mean was then calculated, combining the ROUGE/BLEU and GPT-4 scores. Here, GPT-4 metrics received a higher weight due to its advanced capabilities and the specific definitions it employs for each metric, well-suited to our evaluation goals. However, including both methods mitigates potential biases in LLM-generated metrics.

- **Penalizing Errors:** The score from the previous step was further adjusted by applying penalties based on the unit test error rate (higher penalty) and the number of ruff formatting errors (lower penalty) detected. This reflects the varying impact these errors have on docstring functionality.

The composite score was created by applying a linear combination of all of the scores gathered, as described in Algorithm 1 in the appendix. This approach, while providing a preliminary measure of overall docstring quality, is based on heuristic weightings and may require further refinement. More sophisticated weighting schemes or alternative aggregation methods could potentially enhance the score's reliability.

4 Statistical Analysis

The statistical_analysis.ipynb notebook was utilized to comprehensively analyze the data collected using the methods outlined in the Evaluation section. This analysis included basic statistical measures and various visualizations, such as box plots and bar charts, to effectively illustrate the results and provide insights into the performance of the models.

4.1 Model Comparison per Metric

Figure 2 reveals performance variations across our tested models on the comparative evaluation metrics set. While T5 consistently underperforms, Claude's performance trails behind both GPT-3.5 Turbo and Gemini-1.0-pro by a smaller, yet noticeable margin. Notably, Gemini-1.0-pro and GPT-3.5 Turbo exhibit similar medians for most metrics, with Gemini-1.0-pro potentially having a slight BLEU score advantage. Interestingly, Gemini-1.0-pro shows a wider range of scores across all metrics (except ROUGE-1 f-score) compared to GPT-3.5 Turbo and Claude, suggesting possible performance variability.

Figure 3 shows the mean scores for the 5 LLM based evaluation metrics. T5 does poorly on all measures, while GPT does best, with Claude and Gemini closely behind, respectively. Interestingly, for all models except T5, relevance scored highest and completeness scored lowest, meaning there's room to capture more important information. Overall, this data provides valuable insights for fine-tuning these models for docstring generation tasks.

By focusing on areas where specific models

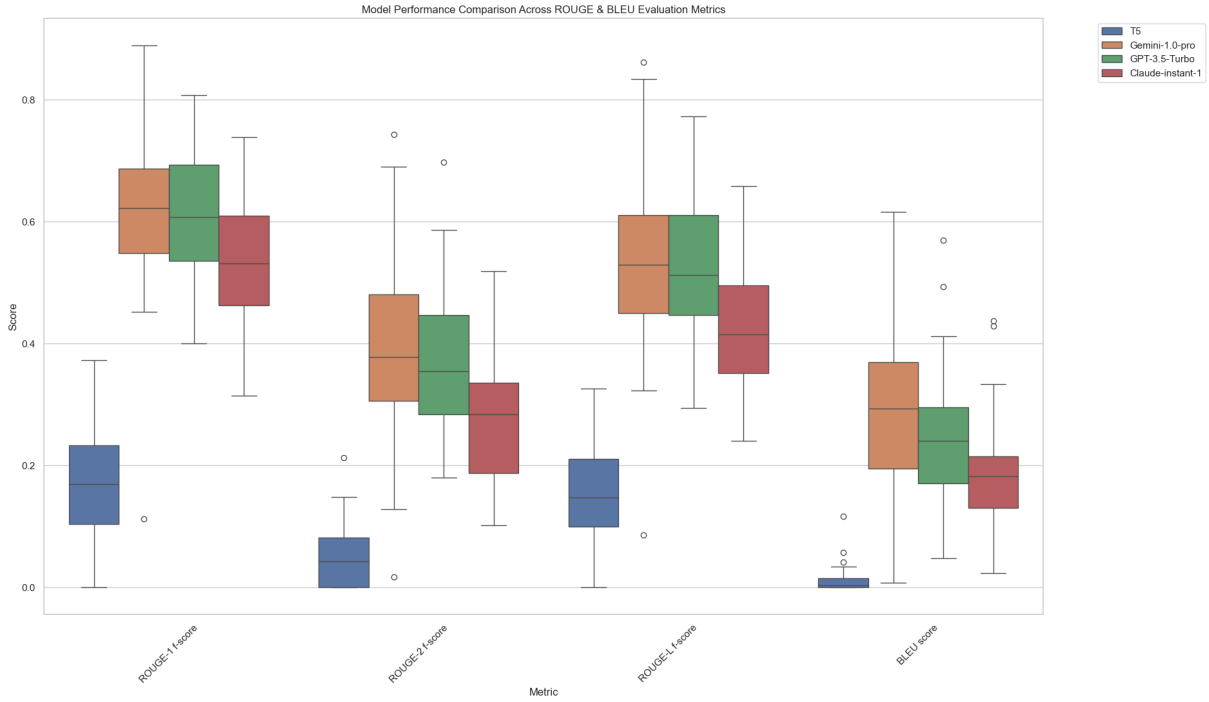

Figure 2: Comparative metrics model comparison boxplot

underperform, we can potentially enhance their ability to generate better and more reliable docstrings.

When examining the boxplot for the same metrics without T5 (in the appendix, 9), we can see that interestingly in all metrics but readability, all three models have the same median value. We should note that GPT exhibits the least variability, with fewer outliers and smaller interquartile ranges. In contrast, Claude generally shows the largest range among the models in most cases, indicating more variability in its performance.

In Figure 4 we see the total number of formatting errors caught by the *ruff* formatter per model. We should note that many of the docstrings generated by T5 were single-line,making it difficult for ruff to catch errors due to the less strict formatting requirements for single-line docstrings. GPT and Gemini hardly made any formatting mistakes caught by the formatter, while Claude had a lot more. When examining the output of the formatter we've found that many errors were minor and easily correctable (e.g., a space after a colon), meaning this information can improve the model quickly and easily as a part of a fine-tuning effort.

Table 1 presents the key statistics on unit test failure rates for code generated based on the docstrings produced by each model. As expected, T5 exhibits the poorest performance, with exceptionally high mean and median values. Conversely, the other models show significantly better performance, with means and medians around 50%. Notably, GPT emerges as the most robust model, featuring the lowest mean, median, and standard deviation. Gemini and Claude display comparable results, with Gemini having a slightly lower standard deviation but a higher median, while Claude has a lower median and higher mean. It is important to note that, due to the limited size of our database, the observed differences between the three models might not be statistically significant.

Model	Mean	Median	S.D
T5	0.93	1.00	0.21
Gemini	0.47	0.50	0.38
GPT	0.46	0.42	0.37
Claude	0.51	0.47	0.38

Table 1: Unit test error rate statistics per model

Table 2 presents the final scores for each model on the docstring benchmark. As expected, T5 scored the lowest, while Claude, Gemini, and GPT secured the top three spots, respectively. Despite Gemini exhibiting a larger standard deviation, it outperformed Claude with significantly higher

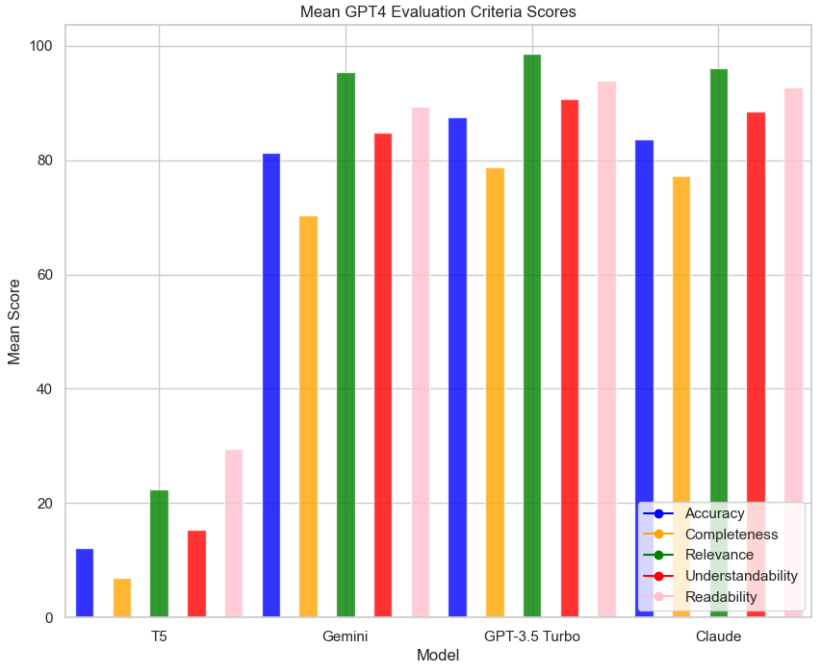

Figure 3: Barplot for mean values for quality metrics evaluated by GPT-4

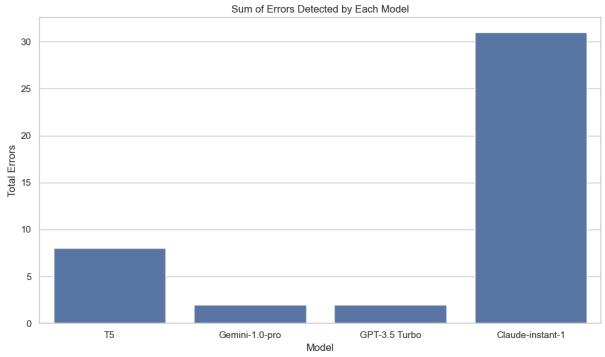

Figure 4: ruff error counts per model

Model	Mean	Median	S.D
T5	0.07	10.04	0.08
Gemini	0.55	0.54	0.18
GPT	0.57	0.58	0.15
Claude	0.47	0.49	0.16

Table 2: Composite metric statistics per model

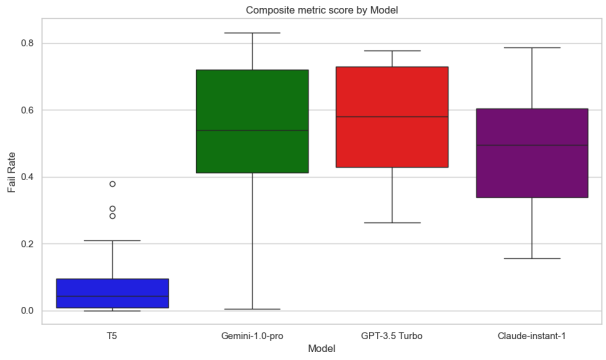

Figure 5: Composite scores model comparison boxplot

median and mean scores, making it the second-best performer. Figure 5 illustrates this by showing Gemini's wider range of minimum and maximum values compared to Claude, yet its box is positioned above Claude's, reinforcing our earlier claims. Overall, it is evident that the final scores on the benchmark are still far from perfect, indicating that there is still substantial room for improvement and potential advancements in docstring generation by these models.

4.2 Code Type comparison per metric

Examining performance across code groups on comparative evaluation metrics reveals class docstrings achieving the highest scores, while those for functions using Numpy and Pandas ranked lowest. This trend is further illustrated in Figure 6, which shows the boxplots for ROUGE-L f-score across code types. We believe class docstrings performed better because they have a clear structure with designated, well defined sections for methods, attributes, etc., while functions using Pandas and NumPy faced greater freedom, and potentially more ambiguity, in describing their functionality and outputs.

When observing the unit tests error rates at each

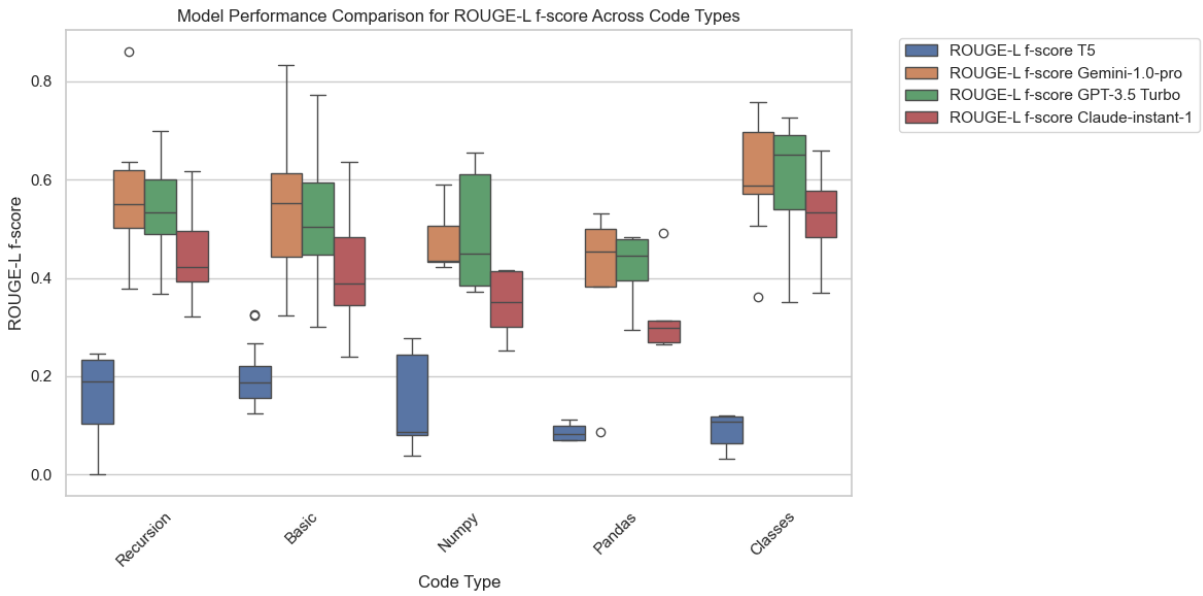

Figure 6: Code type ROUGE-L performance comparison

code type granularity (Fig 7), we can see that each model had some strengths over its counterparts in some code types. For instance, while GPT surprisingly underperformed the other three main models on basic functions, it surpassed them in handling Numpy and Pandas functions. We can use this information to finetune each model on the code types it struggles the most with.

Figure 8 provides a code type-specific breakdown of the composite metric, revealing GPT's consistent performance against Gemini's strengths in certain areas like recursion. This visualization aids in assessing model robustness (box size, min/max values), identifying top and bottom performers for each code type, and informing model selection for specific docstring generation tasks.

5 Summary & Conclusion

This paper proves the potential in further exploring our proposed benchmark, which offers a robust framework for evaluating docstring generation capabilities of LLMs. By combining various evaluation metrics and incorporating LLM-based assessments, we provide a comprehensive understanding of model performance. While our results demonstrate the limitations of current models, they also highlight the potential for significant improvements, as the results were both informative and not trivial for the models.

Future research ideas to explore include further investigating the evaluation metrics generated by GPT-4 and fine tuning the model for more precise evaluation, exploring different metric combinations for the final, composite metric to score the models, investigate code generation for functions and classes with non-descriptive names and variables to assess model performance in less contextually rich environments and more.

6 Code

Our code for docstring generation, metric calculations and all evaluations is located in our GitHub repo.

References

Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, Alex Ray, Raul Puri, Gretchen Krueger, Michael Petrov, Heidy Khlaaf, Girish Sastry, Pamela Mishkin, Brooke Chan, Scott Gray, Nick Ryder, Mikhail Pavlov, Alethea Power, Lukasz Kaiser, Mohammad Bavarian, Clemens Winter, Philippe Tillet, Felipe Petroski Such, Dave Cummings, Matthias Plappert, Fotios Chantzis, Elizabeth Barnes, Ariel Herbert-Voss, William Hebgen Guss, Alex Nichol, Alex Paino, Nikolas Tezak, Jie Tang, Igor Babuschkin, Suchir Balaji, Shantanu Jain, William Saunders, Christopher Hesse, Andrew N. Carr, Jan Leike, Josh Achiam, Vedant Misra, Evan Morikawa, Alec Radford, Matthew Knight, Miles Brundage, Mira Murati, Katie Mayer, Peter Welinder, Bob McGrew, Dario Amodei, Sam McCandlish, Ilya

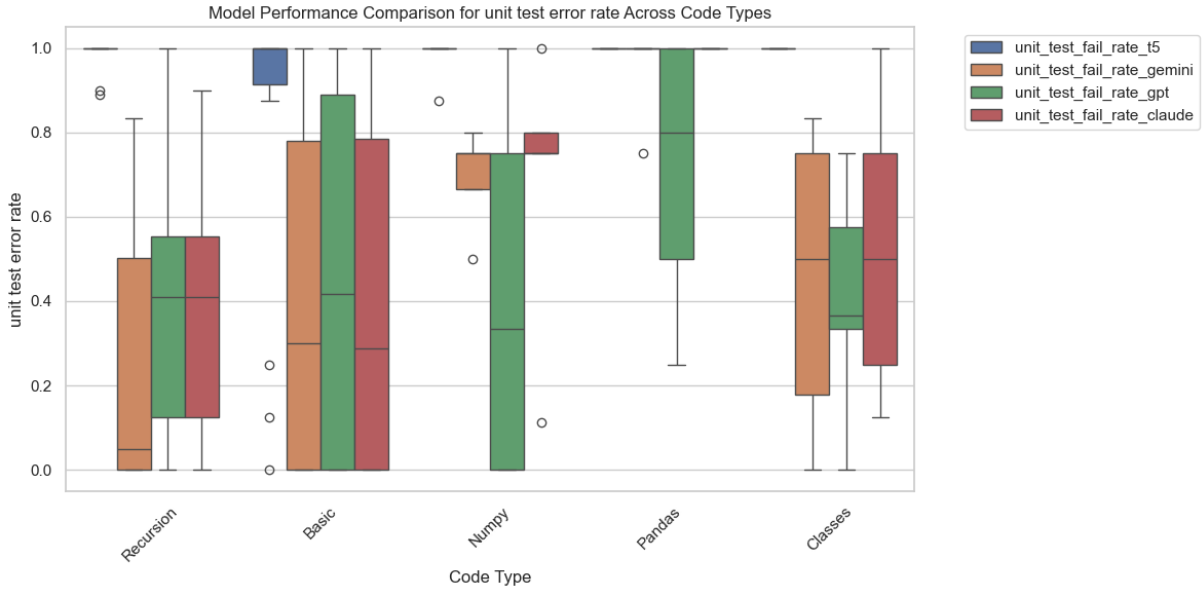

Figure 7: Code type unit test error rate comparison

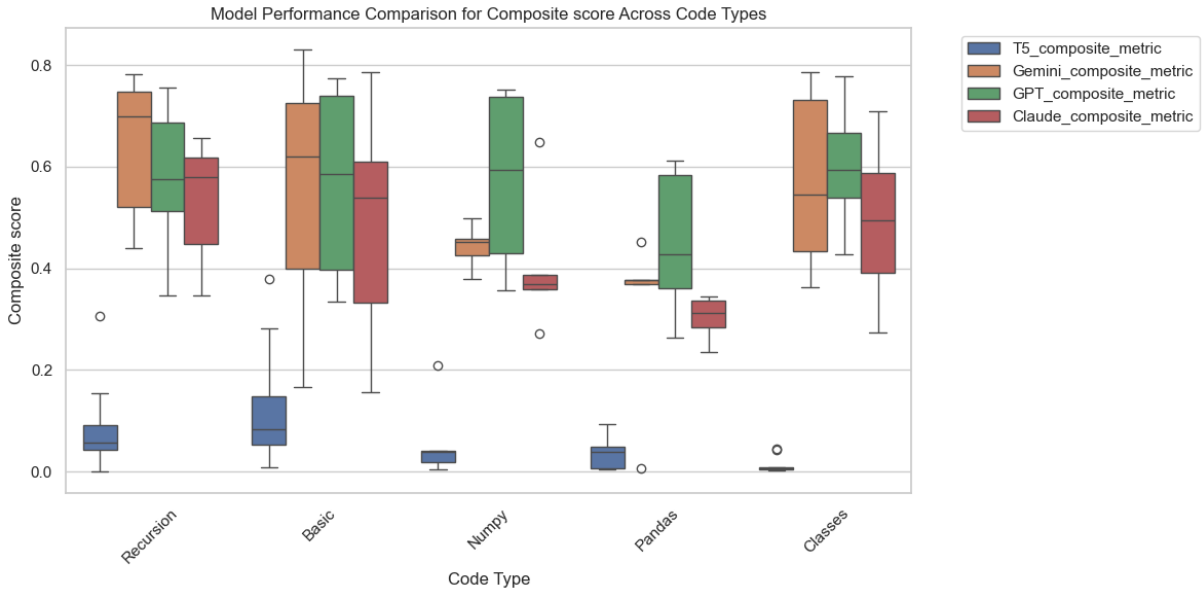

Figure 8: Composite score comparison per code type

Sutskever, and Wojciech Zaremba. 2021. Evaluating large language models trained on code. *ArXiv*.

Shubhang Shekhar Dvivedi, Vyshnav Vijay, Sai Leela Rahul Pujari, Shoumik Lodh, and Dhruv Kumar. 2023. A comparative analysis of large language models for code documentation generation. *ArXiv*.

Carlos E. Jimenez, John Yang, Alexander Wettig, Shunyu Yao, Kexin Pei, Ofir Press, and Karthik Narasimhan. 2024. Swe-bench: Can language models resolve real-world github issues? *ArXiv*.

Chin-Yew Lin. 2004. ROUGE: A package for automatic evaluation of summaries. In *Text Summarization Branches Out*, pages 74–81, Barcelona, Spain. Association for Computational Linguistics.

Fernando Martínez-Plumed, Pablo Barredo, Seán Ó hÉigeartaigh, and José Hernández-Orallo. 2021. Research community dynamics behind popular ai benchmarks. *Nature Machine Intelligence*, 3:581–589.

Kishore Papineni, Salim Roukos, Todd Ward, and Wei-Jing Zhu. 2002. Bleu: a method for automatic evaluation of machine translation. In *Proceedings of the 40th Annual Meeting of the Association for Computational Linguistics*, pages 311–318, Philadelphia, Pennsylvania, USA. Association for Computational Linguistics.

A Appendix

A.1 ruff formatter rules

The formatter was configured to check for Google-style formatting. This means it holds to all of the pydocstyle rules, besides the following list: D203, D204, D213, D215, D400, D401, D404, D406, D407, D408, D409, and D413. Other than that, we've also excluded rule D212.

Algorithm 1 Composite Metric Calculation

Require: Data row for each model with metrics
Ensure: Composite metric for each model

Metrics and Weights:

Comparative Metrics (0-1 scale): $(M_1, M_2, M_3, M_4) = (\text{ROUGE-1}, \text{ROUGE-2}, \text{ROUGE-L}, \text{BLEU})$

Comparative Weights: $(W_1, W_2, W_3, W_4) = (0.1, 0.1, 0.3, 0.5)$

Quality Metrics (0-100 scale): $(Q_1, Q_2, Q_3, Q_4, Q_5) = (\text{Accuracy}, \text{Completeness}, \text{Relevance}, \text{Understandability}, \text{Readability})$

Normalize Quality Metrics: $Q_i \leftarrow Q_i/100$ for $i = 1, 2, 3, 4, 5$

Quality Weights: $(W_{Q_1}, W_{Q_2}, W_{Q_3}, W_{Q_4}, W_{Q_5}) = (0.225, 0.225, 0.225, 0.225, 0.1)$

Ruff Formatting Metric: Count of Errors (non-negative integer)

Unit Test Fail Rate: Fail Rate (0-1 scale)

Calculation:

Calculate Weighted Mean of Comparative Metrics:

$$M_W = \sum_{i=1}^{4} (W_i \cdot M_i)$$

Calculate Weighted Mean of Quality Metrics:

$$Q_W = \sum_{i=1}^{5} (W_{Q_i} \cdot Q_i)$$

Calculate Combined Weighted Mean:

$$\text{Weighted\_Combined\_Mean} = 0.3 \cdot M_W + 0.7 \cdot Q_W$$

Adjust for Unit Test Fail Rate:

$$\text{Adjusted\_Metric} = \text{Weighted\_Combined\_Mean} \cdot \frac{(2 - U)}{2}$$

Scale by Ruff Formatting:

$$\text{Composite\_Metric} = \text{Adjusted\_Metric} \cdot \frac{5}{5 + R}$$

Return Composite Metric

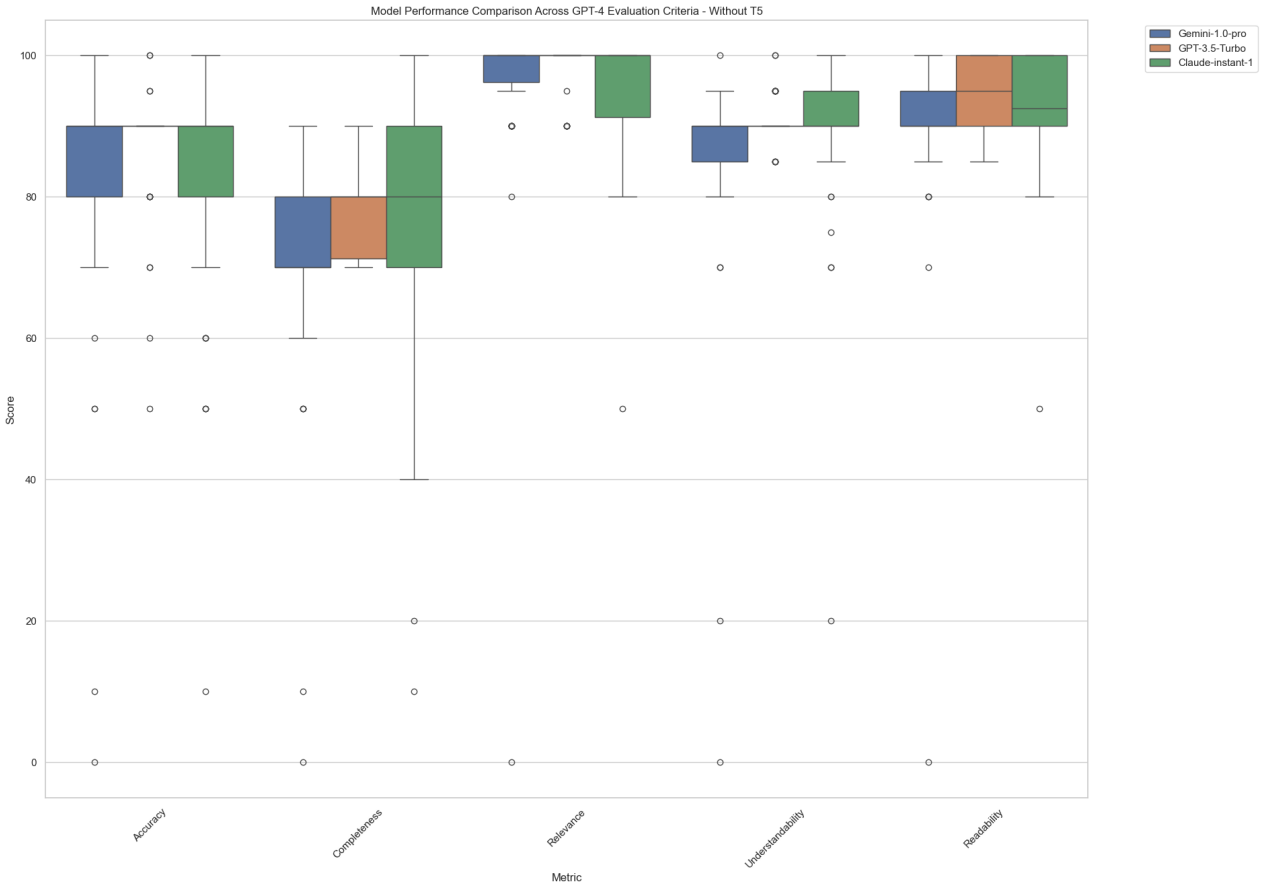

Figure 9: Quality metrics evaluated by GPT-4 boxplot (without T5)