Optimizing AI Developer Efficiency

Jason Birdsong, Luke Cerminaro, Ryan Eagan, and Bryan Kohler
Department of Computer Science, University of Dayton
Riverside Research
CPS 491: Capstone II
Prof. Lacie Stiffler, Tristan Karnes, and Josie Juhring
March 10, 2025

Overview

Large Language Models (LLMs) have rapidly become a valuable asset in many developer's toolkits. By utilizing LLMs, developers can accomplish tasks in a much more efficient manner, and developers, guided by an LLM, can even work proficiently with new tools and technologies that they might not be familiar with. However, querying external proprietary LLMs can quickly rack up costs and can pose potential latency, platform dependence, and data security issues to the companies that utilize them. This leads us to look for an alternative solution in open source LLMs that are small enough to run efficiently on most devices. By utilizing smaller, locally run open source models, developers can query LLMs with minimal latency and security risks and without paying any fees. In order to maximize developer efficiency using these open source LLMs, we want to make sure that any query that developers have is sent to the most proficient model for that type of problem. To this end, we need to first define a list of potential prompt categories that developers are likely to ask an AI model, curate a list of the best open source LLMs for coding related tasks, and then compare models to find the best model for each prompt category. In this report, we first discuss our methodology in finding and ranking different open source LLMs. Then we will discuss our selected development categories and the rationale behind each one. We will then move on to the benchmarks we have found to compare LLMs by for each category, with some overlap, and we will end the report with our findings and LLM recommendations for each development category.

Methodology

Preliminary Research

Before delving into the concrete research we first individually did broad and general research into LLMs and benchmarks, i.e. how benchmarks typically work, how they can compare benchmarks, which LLMs were open source, and other overview knowledge that would assist us in deeper research. At this stage we discovered some of the more popular benchmark names (HumanEval, MBPP), as well as the more well known open source LLM families (Llama, Mistral). Notably the trend of what categories of coding researchers typically looked at started to take shape, and a general sense of what the categories could look like came to fruition. At this stage, however, it was already evident that thorough research on LLMs in respect to coding abilities were more sparse than we had anticipated.

Searching for Benchmarks

All team members searched through accredited sources to find mentions, if not direct listings of some percentage values, of various coding-based benchmarks. A conscious attempt was made to get the widest breadth and diversity of benchmarks as possible. Each benchmark, its corresponding source, and the general attribute it measured, was kept track of in a Google Sheet. By the end of this a list of ~20 accredited benchmarks were compiled, each having some potential to be used for a final comparison in a major coding category. At the end, 12 benchmarks were chosen primarily due to their prevalence and also their rigorous/pertinent standards of measuring.

Categorizing Benchmarks

These 12 benchmarks (listed below) were divided up among the team members in order to look into each one more in depth, this included how the benchmark measured a successful output, what type of input each fed to the LLM, how the questions that make up the benchmark's dataset were gathered, and other similar attributes. During these searches, any measurements for open source LLMs on a given benchmark were noted and sourced in the central Google Sheet. After looking into these benchmarks and what type of coding aspect they measure, as well as based on preliminary research, 5 major categories were identified: Code Translation, Code Completion, Code Generation, Code Repair, and Code Interpretation. All 12 benchmarks were then sorted into which category they must directly lend to. Some benchmarks appear in 2 separate categories due to subtle underlying differences in measurement within specific benchmarks, for example DevQualityEval has both a translation and debugging portion of their benchmark, and so it was most fitting to ascribe the benchmark to both categories. Each category gives decimal weights to each benchmark inside its scope, with all benchmark weights of a category adding to 1.00. These weights are based off the in-depth benchmark research and are meant to quantify how significant, rigorous, relevant, and overarching each benchmark is in its specific category.

Selecting and Comparing LLMs

The open source LLMs that were seen as most commonly discussed in our preliminary research, and then most commonly reported on from benchmarks, were chosen to be compared against each other on the various benchmarks and categories. After additional research to fill in as many benchmark scores for as many of the LLMs as possible, tables of the measurements sorted by category were formed. If a LLM had no measurement that could be found, it was marked as a 0, as it is assumed not being rated on a benchmark correlates to a lack of popularity which is attributed to a lack of efficacy at said benchmarks, i.e. a highly scoring LLM is assumed to rise to popularity through heavy promotion by its owner company. Additionally, assigning higher than a 0 would be assuming some degree of efficacy at a task without evidence, which is treated as worse than underrepresenting efficacy. The weights per benchmark were used to calculate a weighted average for each LLM in each category, as well as a raw average to compare to in order to remove any potential bias from how the weights were assigned. Since the raw averages only occasionally had re-orderings of closely scored LLMs that weren't ranked first anyway, we assume the weights are unbiased and use the weighted scores for final comparisons.

Additional Changes and Remarks

Since Llama3 was found to exceed its competition in all 5 of the major broad categories, it was decided to add additional more niche categories retroactively to allow for the LLM router to route to more than one LLM. Looking back on previous benchmarks it was found that Granite 3.0 performed better than Llama on the DS-1000, a data science exclusive benchmark, so a data science category was created in order to route to Granite. The DevQualityEval benchmark rated the Mistral family's Codestral Mamba significantly higher than Llama3 at correcting syntactical errors, so the Code Repair category was broken down into general Code Repair and Syntax Repair. Lastly, it is important to note that Llama3 was a noticeably newer generation of LLMs, which when combined with how many sources research it due to being backed by the largest company (for open source models), explains why it seems so overwhelmingly dominant at the benchmarks. It is highly likely that if other recent generation LLM models had as much research into them as Llama3, that there would more readily be a diversity of LLMs to route to by category. This seems especially true considering Codestral Mamba, a similar age as Llama3, outperforms it in one of the only benchmarks both have a measurement for.

Selected Coding Categories

Code Translation

Code Translation refers to the creation of code in one programming language that is equivalent to code in a different language. This category is useful because developers often have specific requirements regarding what programming languages are used in the process of development. Some similar functionality for a project may have already been developed, and quick translation would lead to a more efficient workflow. Accurate and efficient code translation allows for developers to be more efficient and adaptable when addressing programming problems. Developers will be able to save time if they do not have the expertise in a particular language to translate it easily. An AI model that would be sufficient for code translation should be able to be given a section of code and be able to accurately generate a section of code in the specified language that has the same functionality.

Code Completion

Code Completion involves giving a model incomplete code of some kind, whether it be a method or function header or an otherwise unfinished piece of code, and a description of what the code should do in order for the model to fill in the missing parts of the code. This category is viewed as important as often in the development process a developer will have a framework for code but need assistance in completing the functionality intended. This also becomes useful in continuing development in an application that has already been built, allowing for pre existing functionality to be taken into account when creating new code.

Code Generation

Code Generation is very similar to Code Completion in that, in both categories, the LLM is being tested on its ability to end with working and functioning code. However, the Code Generation category focuses more so on measuring an LLM's ability to create working code based off of natural language text input and with no direction in the form of a function/method header. By natural language text, it is meant as a regular string of words to describe the problem and outputs in a way a normal human can read it and understand, or conversely write it. This makes it the easiest and most direct way to prompt an LLM for creating code, making it potentially the most important category due to its expected repetitive use by software developers.

Code Repair

Code Repair is a broad category that refers to the general process of debugging, or identifying and fixing errors in code. This category is incredibly useful to developers, as developers usually spend a significant amount of their time hunting for and fixing bugs in their code, and they may turn to AI to help save them the headache of debugging. LLMs can potentially diagnose and fix code in a fraction of the time that it would take human developers, which can free up developers for other work and make their lives significantly easier.

Code Interpretation

Code Interpretation is giving a model a portion of code and asking it to explain what the code's purpose and functionality is. This category is important as it is key in creating documentation for any program. If a developer is tasked with writing up documentation about a particular piece of code but did not write and does not understand what it does exactly, they might turn to an AI model in order to help them figure it out. This category might also extend to a model being able to insert comments into code directly without first explaining the code to the user.

Data Science

Data Science specifically refers to the generation and completion of code that is focused on the manipulation and use of some form of external data. This category is useful as code is often an integral part of the day to day for a data scientist, however they may not be formally trained in coding in general, or just the language that they are using. By using an AI model to assist the coding parts of data science, it allows those in the field to focus on more of the experimental work rather than debugging code or trying to find the right function.

Syntax Errors

This category is a more specific subset of Code Repair. Syntax errors are one of the most common error types that developers can encounter. They are usually arbitrarily easy to find and fix, but they require an understanding of the programming language that the code is written in. If a developer is unfamiliar with the language that they are working in, they may give the buggy code to an AI to fix.

Benchmark Analysis

Code Translation

DevQualityEval (DQE)

DQE is developed by Symflower, with the goal of testing for proficiencies in software engineering, programming challenges, and solving problems across multiple languages. It currently tests 3 languages, Java, Go, and Ruby. DQE uses a proprietary scoring system, displayed in percentages, that tracks many different elements to evaluate the performance of LLMs. These elements include code compilation rates, test coverage percentages, code efficiency, response completeness and conciseness, task-specific performance, token usage and response relevance, cost-effectiveness, and reliability. DQE specifically has a transpile score which can be beneficial for code translation scoring. Symflower provides a leaderboard showing the performance of many LLMs on the benchmark. However, it is currently locked behind a paywall of $10.93 USD. Due to that, we do not have access to the exact scores of LLMs on the latest version of DQE. There is, however, a separate report of DQE 1.0 with transpilation scores. Of these, the top three models are Codestral Mamba 7B(74.77), LLama 3 8B (66.02), and Mistral 7B(43.65).

ClassEval-T (CE-T)

ClassEval-T is a class-level code translation benchmark that is an extension of the python benchmark ClassEval. CE-T covers multiple languages, those being Python, C++, and Java. It measures the ability of LLMs to translate code from the three aforementioned languages into each other. It works at the class level, so it can better account for more complex coding problems than other benchmarks that score on the method level. CE-T is scored using Compilation success rate (CSR) and computational accuracy on the method level (CAm) and class level (CAc). These are all measured in percentages, and each provide separate insight into the ability of LLM's to accurately translate code. Taking the average of translation success rate metrics to and from each of the three languages for each model, the study shows the highest success rate overall for Llama3 8B and CodeLlama.

Code Completion / Data Science

HumanEval

HumanEval is a benchmark developed by OpenAI which tests whether code generated by a model works as it is supposed to. This is done by giving 164 manually written programming challenges to a model. Each challenge consists of a model being handed a Python method signature and standard English sentences explaining what the method should do and the model must then fill in the method. This benchmark uses a pass@k metric to measure performance, where k is the number of tries that a model gets to successfully complete the challenge. Most times this benchmark is tested using pass@1, meaning that each model only gets one chance to complete the challenge. It is scored between 0 and 100, representing the percentage of tests passed in k tries. As seen in **Figure 1** the top 3 models are Llama3 8B with a score of 62.2 on pass@1, Granite 3.0 8B with a score of 52.44 on pass@1, and codellama-python with a score of 38.4 on pass@1.

MBPP

MBPP or Mostly Basic Python Problems Dataset, is a benchmark developed by Google which tests a model's ability to solve 1000 crowd sourced Python programming problems simple enough to be completed by beginner Python programmers. Each problem contains a description, a code solution, and 3 test cases that are run. Similar to HumanEval, MBPP uses a pass@k metric to determine performance and the score is a percentage of problems correctly solved in k attempts. As seen in **Figure 1**, the top 3 scoring open source models are Llama3 8B with a score of 64.6 on pass@1, StarCoder 2 7B with a score of 54.4 on pass@1, and CodeGen6B with a score of 50.8 on pass@1.

DS-1000

DS-1000 is a benchmark created by collaborators from several well-renowned universities as well as MetaAI which consists of testing models on 451 real StackOverflow Questions relating to Python data science which have been turned into 1000 problems spanning 7 Python libraries. As with the previous 2 benchmarks, DS-1000 also uses a pass@k metric to determine performance and score is a percentage of problems correctly solved in k attempts. As seen in **Figure 1** and **Figure 6**, the top 3 scoring open-source models are Granite 3.0 8B with a score of 33.8 on pass@1, Llama3 8B with a score of 31.5 on pass@1, and StarCoder 2 7B with a score of 31.4 on pass@1. As there is a discrepancy in which model is the best for this benchmark (Granite 3.0 rather than Llama3), this benchmark will also be used to determine the best model for the Data Science Category.

BigCodeBench

BigCodeBench is a benchmark created by collaborators from numerous high-profile universities which tasks models with completing 1140 function-level Python tasks that require 139 libraries to complete, created as a “successor” to HumanEval.As with the previous benchmarks, BigCodeBench uses a pass@k metric to determine performance and score is a percentage of problems correctly solved in k attempts. As seen in **Figure 1**, the top 3 scoring models are Llama3 8B with a score of 36.9, Granite 3.0 8B with a score of 35.4, and StarCoder 2 7B with a score of 27.7 all on pass@1.

Code Generation

APPS (Automated Programming Progress Standard)

A python-based coding benchmark with 10,000 coding problems in total, acquired through various open source coding websites such as “Codeforces”. The questions are asked in the format of natural human language, i.e. a description of the problem and what needs to be done, as well as sometimes an example input and output. There are 3 different classes of questions: introductory, interview, and competition, with each being noticeably more difficult than the last category. Similar to other types of benchmarks, APPS also typically uses a pass@k strategy, and due to its difficulty it appears much more common for pass@5 to be the standard instead of pass@1. Unfortunately, likely due to its multi-class nature and large dataset, APPS is a more uncommon benchmark. As can be seen in **Figure 2**, only 3 open source LLMs had readily available APP scores, which we averaged across the three classes, leaving Llama3 considerably in the lead with a 15.7%, about 9% better than the runner up of Codellama Python with a 6.06.

BigCodeBench (BCB)

As noted under Code Completion, BCB was a python benchmark that came as a collaboration of universities, designed so that the LLM must understand function calls from various libraries to complete function level tasks. BCB has two minor variations, one where the LLM is given a docstring describing what to do, and another where the LLM was given a more real world text query. Because of this subtle nuance, and the fact that many sources reporting BCB scores made no distinction between the two, we thought it fitting to include it in both generation (measuring natural text understanding) and completion (understanding and filling in code). As is the case in **Figure 1**, **Figure 2** shows the same conclusion on BCB, that Llama 3 was the best followed closely by Granite and then less closely by StarCoder2.

NaturalCodeBench (NCB)

The NaturalCodeBench was designed as an upgrade to the HumanEval benchmark, with a focus on real coding applications in both Java and Python. To this end they collected real application user queries from CodeGeeX services, and categorized them into 6 different categories: software, data science, algorithm, systems administration, AI systems, front-end. They end up with 402 problems across these 6 categories, with test inputs including diverse data structures and file types. A major difference between NCB and HumanEval is that NCB uses more natural text language, being more akin to a detailed instruction a real user may give to a LLM. As seen in **Figure 2**, Llama 3 outperforms all other LLMs with a 20.70, followed closely by CodeLlama with a 19.50, then less closely by Mistral with a 15.30.

MBPP (Mostly Basic Python Programming)

A dataset containing 974 beginner level function level python programming tasks. The problems were gathered internally at Google from various human coders. In this manner they managed to crowdsource all problem descriptions, function solutions, and 3 test cases per each of the 974 problems. The tasks range anywhere from regular mathematical manipulations to calling on library functions. Since the format of this benchmark's input includes both a descriptive header and a description of the task at hand in regular text, MBPP could be used for both code completion and code generation. The scores are the same between the two categories, so as in **Figure 1**, **Figure 2** shows MBPP where Llama3 leads with a 64.60, followed somewhat distantly by StarCoder2 with a 54.40, and then CodeGen with a 50.80.

MultiPL-E (Multi Programming Language Eval)

MultiPL-E itself is a way to translate benchmarks into other languages, primarily designed for HumanEval and MBPP. This is possible since no translation of the functions themselves is required and because existing benchmarks already use unit tests, i.e. use cases to see if the desired behavior is obtained. MultiPL-E itself uses 18 mini compilers for Python, each corresponding to a different coding language available in the original benchmark model, to translate a function signature, unit tests, a comment description, and any type annotations that may be necessary for a given language. Our sources used HumanEval translated on MultiPL-E for the ratings, which by how they are translated makes the actual input given to the LLM more akin to MBPP than HumanEval, but in non-python languages. We averaged the scores for Java, Javascript, and C++ to get the numbers found in **Figure 2**: StarCoder 2 in first with 32.80, followed by CodeLlama Python and CodeLlama with a 31.94 and 29.41 respectively. Notably this is the only code generation category Llama3 did not win in, because Llama3 did not have any discoverable MultiPL-E ratings.

Code Repair / Syntax Errors

DebugBench

DebugBench is a code repair tailored benchmark created by researchers from Tsinghua University that tests LLM's ability to correct 4,253 instances of buggy code across Java, Python3, and C++. The code samples were initially collected from LeetCode, and then the researchers used ChatGPT-4 to plant synthetic bugs in the code. The synthetic bugs fall into one of four categories: syntax, reference, logic, and multiple, with each of these categories being further subdivided into more nuanced types, such as undefined objects, illegal indentation, and many more. This benchmark uses a pass rate metric to determine model performance in a zero-shot scenario, where the model is not given examples of corrected code for the bug type. The pass rate, calculated by specific bug subtype, is simply just the ratio of test cases that the model successfully repairs the code to the total number of bug instances converted to a percentage. For the purpose of evaluating the models in this report, we will average the pass rate across categories to arrive at a final score for each model. As seen in **Figure 4**, the top scoring open source model by a wide margin is Llama3 8B-Instruct with a score of 42.4%, and CodeLlama 7b-Instruct comes in second with a score of 17.65%. A Mistral family model, Mixtral 8x7b-Instruct was also tested on this benchmark and has the third highest score with 12.4%. However, it is important to note that the authors claim that Mixtral performed so poorly because it was unable to comply with the format of the benchmark. Additionally, since Mixtral8x7b-Instruct has a higher active parameter count than we allow for this study, we have excluded it from the rest of the report.

DebugEval

DebugEval is a benchmark developed by researchers from Northeastern University in Shenyang, China that tests LLM's on their ability to debug code on

5712 instances in Python, C++, and Java across four different categories: Bug Localization, Bug Identification, Code Review, and Code Repair. Code samples for this benchmark were taken from the aforementioned DebugBench, the LiveCodeBench, and from the Atcoder programming competition. This benchmark uses a mix of accuracy and pass@k metrics in order to evaluate models. The benchmark presents bug localization, bug identification, and code review as multiple choice questions to the LLM. The LLM is then graded on its accuracy for these category types. For code repair, the LLM is measured on a pass@1 metric.These four scores are then averaged together to give the model's final score. As seen in **Figure 4**, of the available open source models tested on this benchmark, Llama3-8B-Instruct scored the highest at 49.7, followed by CodeLlama-7B-Instruct at 31.9, and Llama2-7B-Instruct at 14.9.

DevQualityEval

As mentioned under Code Translation, DevQualityEval is developed by Symflower, with the goal of testing for proficiencies in software engineering, programming challenges, and solving problems across multiple languages (Java, Go, and Ruby).In addition to testing these categories, DevQualityEval also has a specific code repair score for each model, which tests models on their ability to fix specifically syntax errors. As seen in **Figure 4** and **Figure 7**, the top scoring viable open source model on the DevQualityEval-Code Repair benchmark was Codestral Mamba, with a score of 93.15%, followed by Mistral 7B at 91.48%, and Llama 3 with a surprisingly low 49.81%.

Code Interpretation

CIBench

CIBench was developed by the Shanghai Artificial Intelligence Laboratory and ShanghaiTech University in order to test an LLM's ability to use code interpreters to perform data science tasks such as making changes to plots and data using data sets. The score is a percentage representing the average accuracy of the process which the LLM used to perform the task, for example what tools it used when, how accurate were the end results, and how good were the visuals such as graphs. As seen in **Figure 5**, only 2 had been tested in the paper Llama3 8B, which had a score of 64.4 and Mistral 7B, which had a score of 48.3.

Final LLM Choices

LLama3 8B

LLama3 was developed by Meta AI, and is the third version of Meta's LLM series. It is open source and released in 2024, building off of previous models in the LLama system. It was released with models hodling 8 billion and 70 billion parameters. The model with 8 billion parameters was tested in this report. LLama3 was found to be dominant in most of the testing categories, including Code Translation, Code Completion, Code Generation, Code Repair, and Code Interpretation. It scored the highest on most indicators, and performed relatively well on most others (see **Figure 1** through **Figure 5**). Moving forward with the development of the application, questions relating to these categories will likely receive the best answers from LLama3 of the tested LLMs.

Granite 3.0 8B

Granite was developed by IBM, and was released in October of 2024. It is the third open source version of the Granite series. The model tested in this report has 8 billion parameters. Granite was found to receive the highest scores from benchmarks relating to the Data Science category (See **Figure 5**). Therefore, questions regarding the Data Science category will likely be best handled by Granite3.0.

Codestral Mamba 7B

Codestral Mamba was developed by Mistral AI, and was released in July of 2024. It is based off of the Mamba architecture, and is designed with code generation in mind. It uses 7 billion parameters. Codestral Mamba was found to have the highest scores in the Syntax Error benchmarks (see **Figure 6**). Therefore, questions focusing on handling syntax errors will likely be best addressed through Codestral Mamba.

Figures

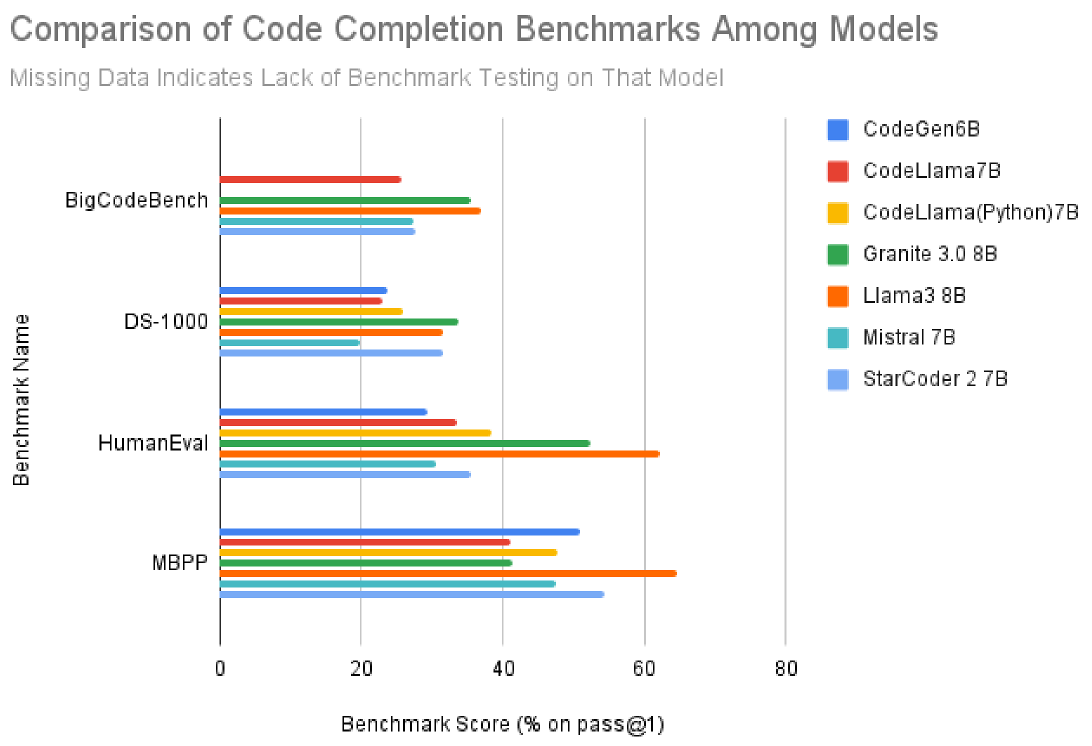

Figure 1: Bar Chart Comparing Code Completion Benchmark Scores on Evaluated Models

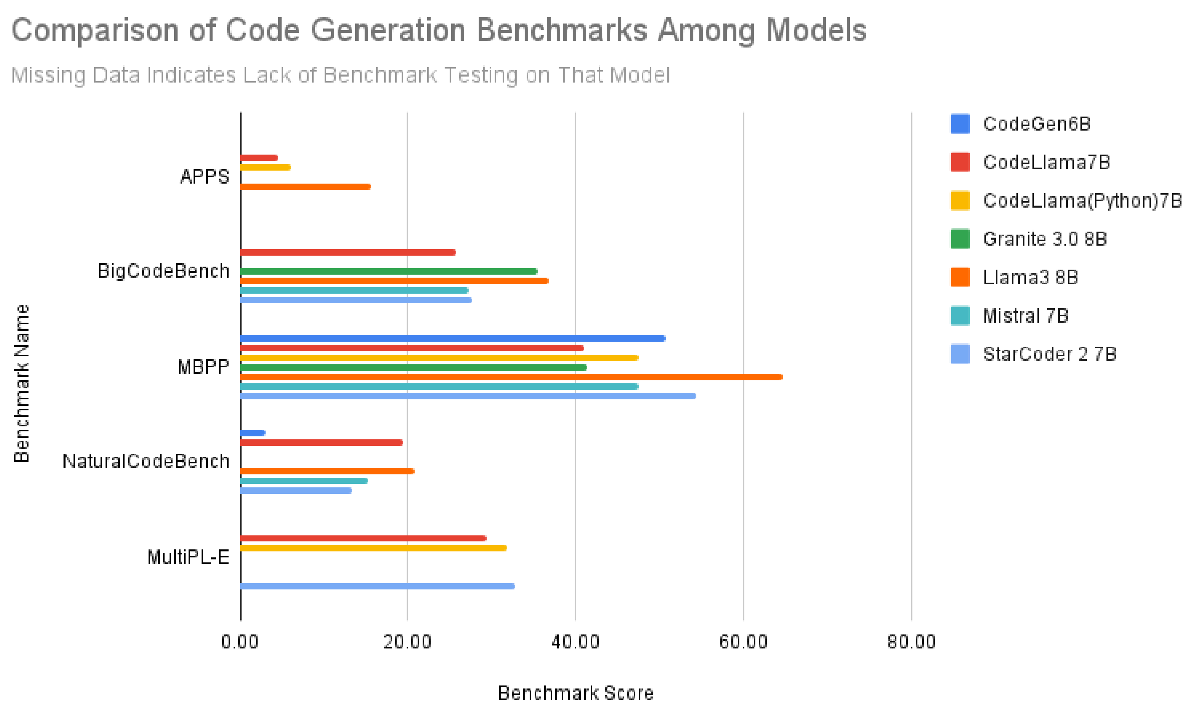

Figure 2: Bar Chart Comparing Code Generation Benchmarks Scores on Evaluated Models

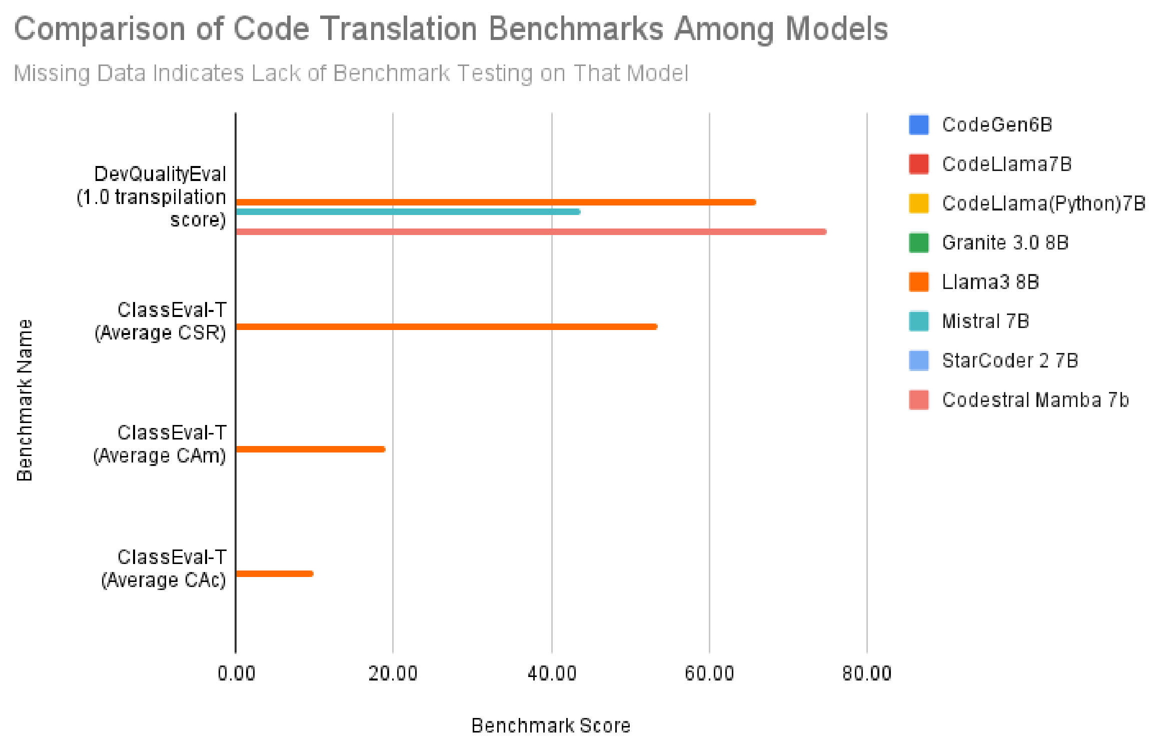

Figure 3: Bar Chart Comparing Code Translation Benchmarks Scores on Evaluated Models

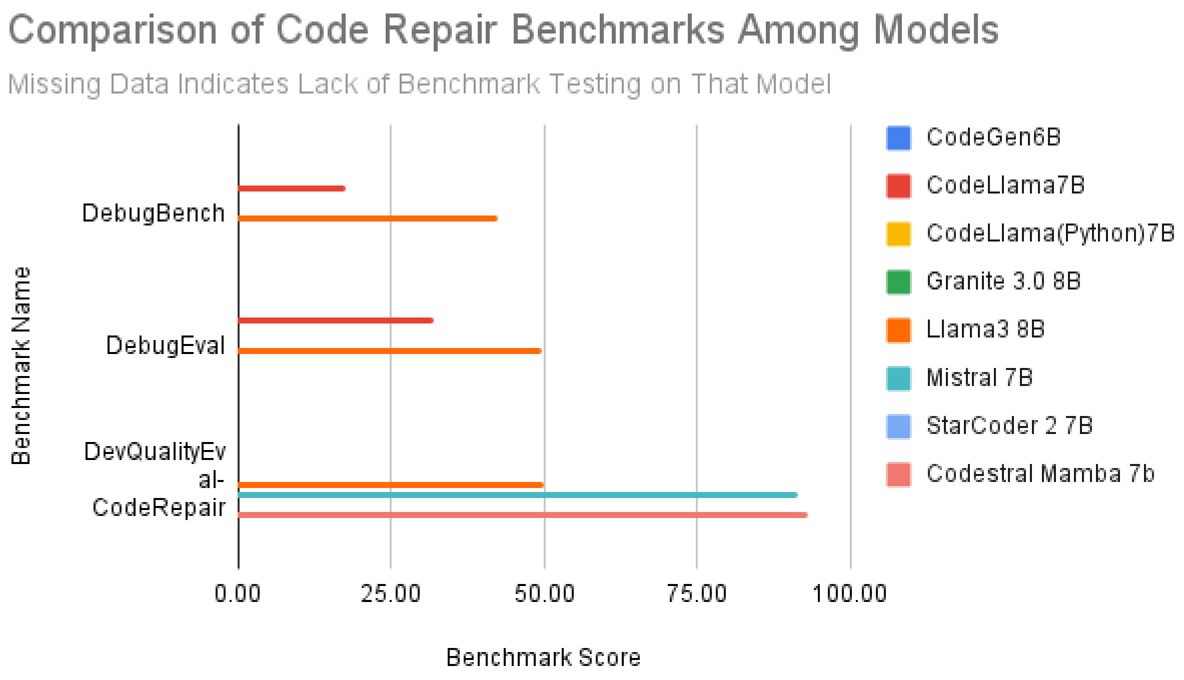

Figure 4: Bar Chart Comparing Code Repair Benchmarks Scores on Evaluated Models

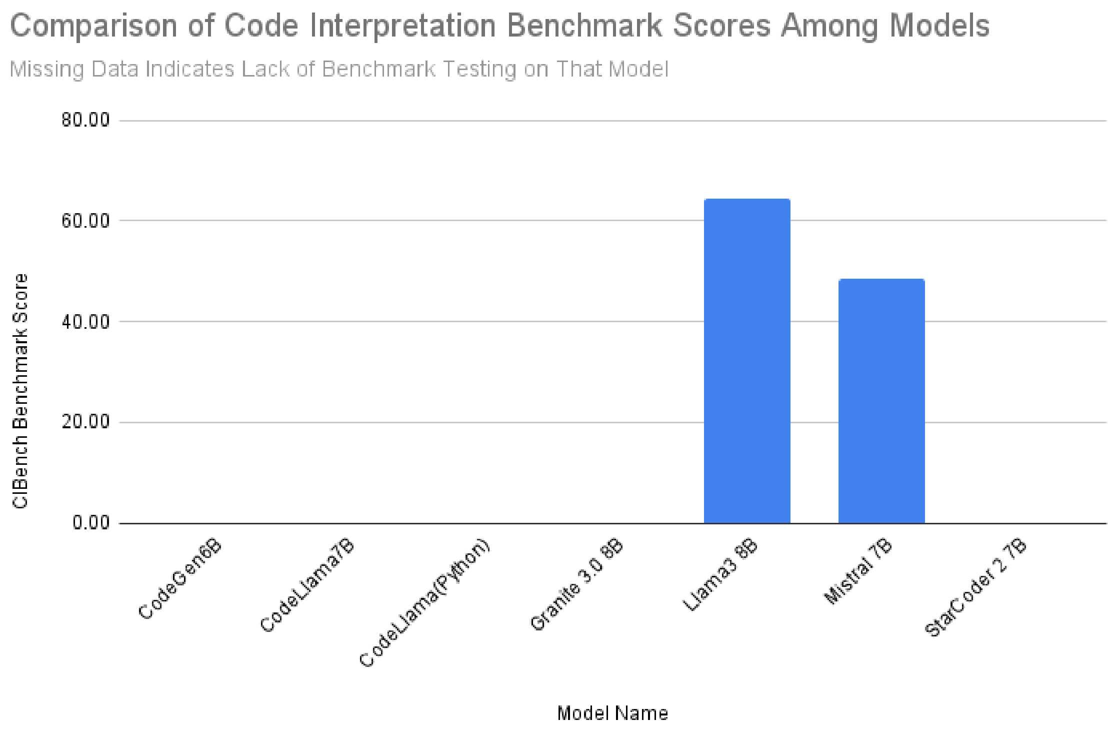

Figure 5: Column Chart Comparing CIBench Scores Among Models

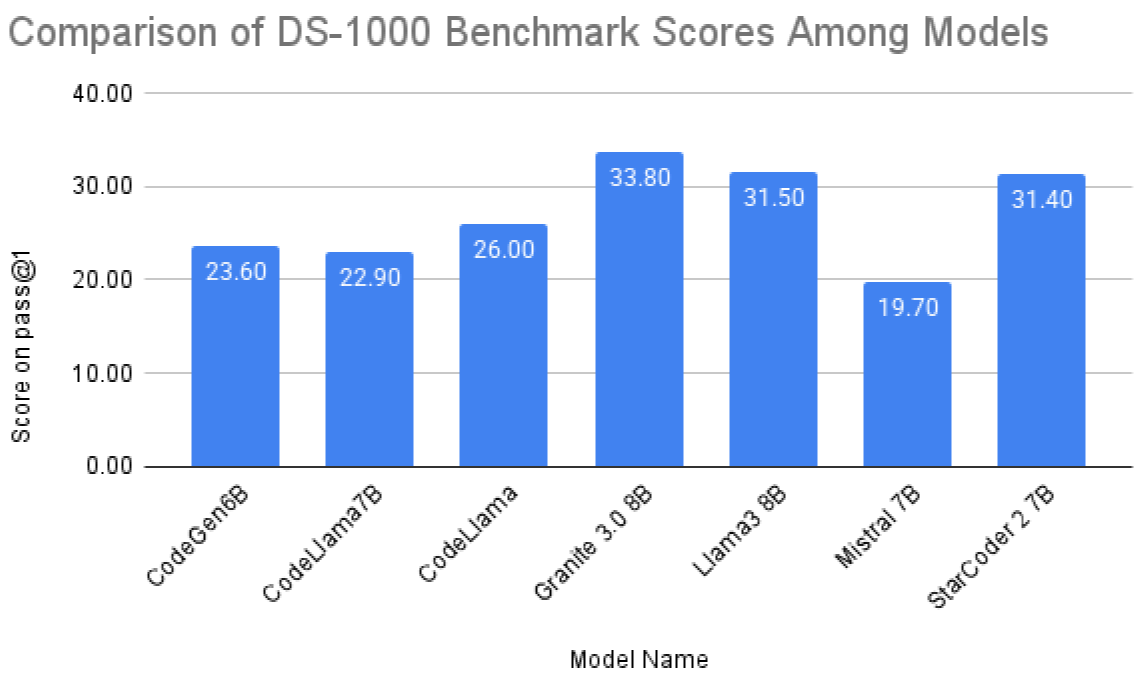

Figure 6: Column Chart Comparing DS-1000 Scores Among Models

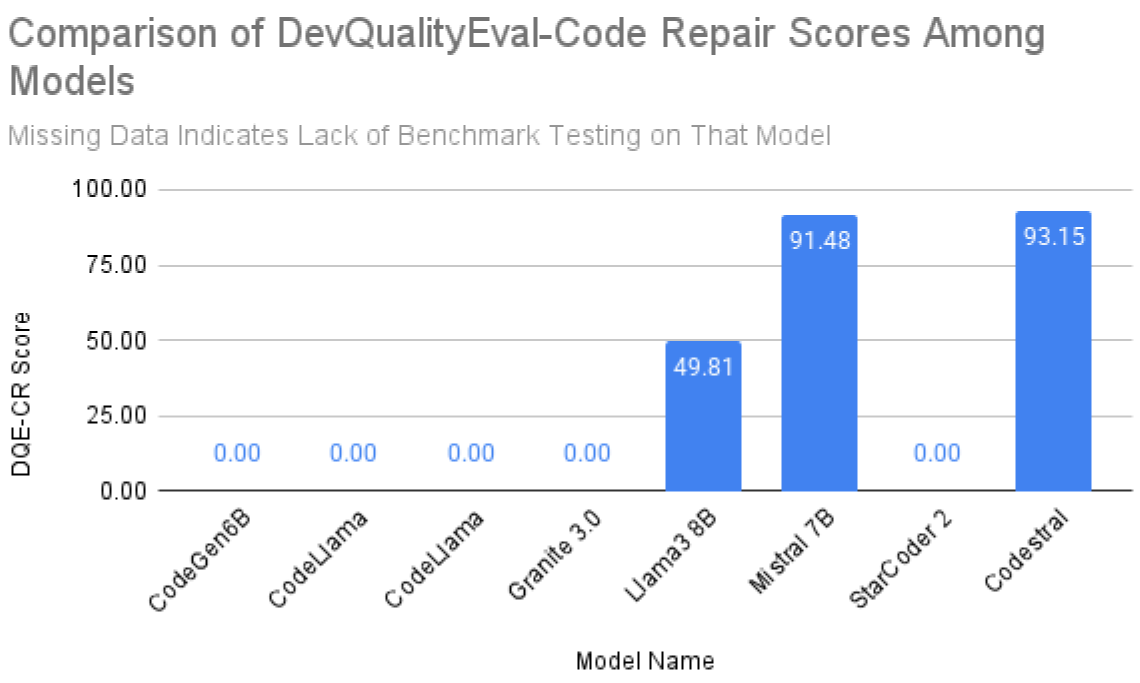

Figure 7: Column Chart Comparing DevQualityEval-Code Repair Scores Among Models

Sources

Association for Computational Linguistics. (2024). Findings of ACL 2024. ACL Anthology. https://aclanthology.org/2024.findings-acl.471.pdf

Baptiste Rozière, Jonas Gehring†, Fabian Gloeckle, Sten Sootla, Itai Gat, Xiaoqing, Ellen Tan, Yossi Adi, Jingyu Liu, Romain Sauvestre, Tal Remez, Jérémy Rapin, Artyom, Kozhevnikov, Ivan Evtimov, Joanna Bitton, Manish Bhatt, Cristian Canton Ferrer, Aaron, Grattafiori, Wenhan Xiong, Alexandre Défossez, Jade Copet, Faisal Azhar, Hugo Touvron, Louis Martin, Nicolas Usunier, Thomas Scialom, Gabriel Synnaeve. (2023). Code Llama: Open Foundation Models for Code. arXiv. https://arxiv.org/pdf/2308.12950v3

BigCode. (n.d.). BigCodeBench Leaderboard. Hugging Face. https://huggingface.co/spaces/bigcode/bigcodebench-leaderboard

Chen, M., Tworek, J., Jun, H., Yuan, Q., Pinto, H. P. D. O., Kaplan, J., Edwards, H., Burda, Y., Joseph, N., Brockman, G., Ray, A., Puri, R., Krueger, G., Petrov, M., Khlaaf, H., Sastry, G., Mishkin, P., Chan, B., Gray, S., ... Zaremba, W. (2021). Evaluating Large Language Models Trained on Code. arXiv. https://arxiv.org/pdf/2105.09938v3

DS1000. (n.d.). Model Performance on DS1000. http://ds1000-code-gen.github.io/model_DS1000.html

Dewu Zheng, Yanlin Wang, Ensheng ShiI, Hongyu Zhang, Zibin Zheng. (2024). How Well Do LLMs Generate Code for Different Application Domains? Benchmark and Evaluation. arXiv. https://arxiv.org/pdf/2412.18573v1

EvalPlus. (n.d.). Leaderboard. https://evalplus.github.io/leaderboard.html

IBM Granite. (n.d.). Granite-3.0-language-models. GitHub. https://github.com/ibm-granite/granite-3.0-language-models/blob/main/paper.pdf

Nuprl. (n.d.). MultiPL-E. Hugging Face. https://huggingface.co/datasets/nuprl/MultiPL-E/blob/main/README.md

Pengyu Xue, Linhao Wu, Chengyi Wang, Xiang Li, Zhen Yang, Ruikai Jin, Yuxiang Zhang, Jia Li, Yifei Pei, Zhaoyan Shen, Xiran Lyu. (2024). Escalating LLM-based Code Translation Benchmarking into the Class-level Era. arXiv. https://arxiv.org/html/2411.06145v1

Runchu Tian, Yining Ye, Yujia Qin, Xin Cong, Yankai Lin, Yinxu Pan, Yesai Wu, Haotian Hui, Weichuan Liu, Zhiyuan Liu, Maosong Sun . (2024). DebugBench: Evaluating Debugging Capability of Large Language Models. arXiv.
https://arxiv.org/abs/2401.04621

Songyang Zhang, Chuyu Zhang, Yingfan Hu, Haowen Shen, Kuikun Liu, Zerun Ma, Fengzhe Zhou, Wenwei Zhang, Xuming He, Dahua Lin, Kai Chen. (2024). CIBench: Evaluating Your LLMs with a Code Interpreter Plugin. arXiv.
https://arxiv.org/html/2407.10499v1

Symflower. (2024). Comparing LLM benchmarks for software development.
https://symflower.com/en/company/blog/2024/comparing-llm-benchmarks/

Weiqing Yang, Hanbin Wang, Zhenghao Liu, Xinze Li, Yukun Yan, Shuo Wang, Yu Gu, Minghe Yu, Zhiyuan Liu, Ge Yu. (2024). Enhancing the Code Debugging Ability of LLMs via Communicative Agent Based Data Refinement. arXiv.
https://arxiv.org/html/2408.05006v1