ESCUELA TÉCNICA SUPERIOR DE
INGENIERÍA INFORMÁTICA

MÁSTER UNIVERSITARIO EN INGENIERÍA
DEL SOFTWARE E INTELIGENCIA ARTIFICIAL

Diseño de un método comparativo para copilotos de código con IA
Design of an Evaluation Method for AI Programming Assistants

Realizado por
Donat Shergalis

Tutorizado por
Gabriel Jesús Luque Polo

Departamento
ESCUELA TÉCNICA SUPERIOR DE
INGENIERÍA INFORMÁTICA,
UNIVERSIDAD DE MÁLAGA

MÁLAGA, Septiembre de 2025

ESCUELA TÉCNICA SUPERIOR DE INGENIERÍA INFORMÁTICA
GRADO EN MÁSTER UNIVERSITARIO EN INGENIERÍA DEL SOFTWARE E INTELIGENCIA ARTIFICIAL

Diseño de un método comparativo para copilotos de código con IA
Design of an Evaluation Method for AI Programming Assistants

Realizado por
Donat Shergalis
Tutorizado por
Gabriel Jesús Luque Polo
Departamento
ESCUELA TÉCNICA SUPERIOR DE INGENIERÍA INFORMÁTICA

UNIVERSIDAD DE MÁLAGA
MÁLAGA, Septiembre de 2025

Fecha defensa:
El Secretario del Tribunal

Agradecimientos

Página de agradecimientos.

Resumen:

Resumen ES.

Palabras claves: clave1, clave2, ..., clave5

Abstract:

Large Language Models (LLMs) are increasingly used for code generation, but their evaluation remains challenging. Existing benchmarks are often rigid, resource-intensive, and prone to issues such as task saturation and data leakage. This thesis analyzes the limitations of current benchmarking practices and introduces a modular, customizable framework for evaluating LLMs in programming tasks. The proposed system enables flexible task selection, fine-grained configuration, and integration of environmental and cost considerations. An interactive interface supports task management and detailed result analysis, making benchmarking more practical and sustainable. The main contributions include a review of existing benchmarks, the design and implementation of a modular framework, and recommendations for improving benchmarking practices. The results suggest that customizable and eco-aware benchmarks can provide more relevant insights while reducing computational overhead.

Keywords: keyword1, keyword2, ..., keyword5

Índice de contenidos

CAPÍTULO 1

Introduction

Large Language Models (LLMs) have revolutionized software development through AI-assisted programming tools like GitHub Copilot. However, evaluating and fine-tuning these models requires extensive benchmarking, which comes at significant computational, financial, and environmental costs. Current benchmarks often contain irrelevant tasks and provide limited customization options, making them inefficient for specific use cases.

This work explores existing benchmarks for code-generating LLMs and proposes a novel approach: an easily customizable benchmarks with detailed and conveniently explorable outputs that can be tailored to specific needs while remaining cost-effective and environmentally conscious.

This thesis began with a comprehensive literature review, using recent critical reviews as a foundation for analyzing the state of LLM benchmarking. The review was extended by following citations to relevant articles published in late 2024 and 2025, focusing on keywords related to benchmarking and LLMs. The findings from this literature analysis informed the design and implementation of a modular benchmarking system, which was then evaluated for efficiency, flexibility, and environmental impact.

The literature review and further research were guided by the following questions:

- RQ1: What are the main limitations of current LLM benchmarks for code generation?
- RQ2: What metrics best reflect real-world usability and code quality?
- RQ3: How can benchmarks be made customizable for different user needs?
- RQ4: What is the environmental impact of repeated benchmarking, and how can it be reduced?

Main contributions of this work:

- A critical analysis of existing LLM code generation benchmarks and their shortcomings.
- The design and implementation of a modular, customizable benchmarking framework.
- Integration of environmental and cost considerations into the benchmarking process.

- An interactive web interface for configuring benchmarks and analyzing results.
- Recommendations for future benchmarking practices based on empirical findings.

1.1. Problem Statement

Benchmarking LLMs for code generation is essential for both research and practice, but current approaches face critical limitations. Most benchmarks are fixed datasets, which leads to task saturation and data leakage as models are trained on their contents. The lack of customization prevents researchers and practitioners from focusing on tasks relevant to their use cases. At the same time, running large benchmarks consumes significant computational resources, resulting in high costs and environmental impact. Furthermore, benchmark outputs are often limited to single numeric metrics, which fail to capture nuanced aspects of model performance such as efficiency, style, or error patterns.

Therefore, the problem addressed in this thesis is the absence of a flexible, customizable, and sustainable benchmarking framework that can adapt to evolving LLM capabilities and user needs.

1.2. Objectives

The main objectives of this work are:

1. Analyze existing benchmarks and their limitations
2. Design a modular benchmark system that allows:
    - Custom task selection and filtering
    - Configuration of testing criteria
    - Support for multiple programming languages and task types
    - Integration with CI/CD pipelines
3. Implement an interactive web interface for benchmark configuration and result analysis
4. Develop a cost-efficient and environmentally conscious approach to benchmarking

1.3. Relevance of the Work

The relevance of this work lies in addressing the growing inefficiency and environmental cost of LLM benchmarks. By introducing a customizable framework, this thesis provides a practical solution for researchers and practitioners who need targeted, resource-efficient evaluations. The proposed system not only saves time and energy but also improves the usability of benchmarking results, making them more relevant for real-world applications.

CAPÍTULO 2

State of the Art

2.1. Evolution of Code Generation Benchmarks

The evolution of benchmarks for code generation has been driven by the need to evaluate the capabilities of Large Language Models (LLMs) in programming tasks. Early benchmarks focused on isolated tasks, but as LLMs became more sophisticated, the need for more comprehensive and realistic evaluation methods emerged.

The first benchmarks were aimed at text comprehension and contained questions and expected answers, such as GLUE, SQuAD and GSM8K with grade-school math problems (Vendrow *et al.* 2025).

As LLM capabilities expanded, benchmarks shifted towards programming tasks, with a focus on code generation and understanding. Some pioneers in LLM code benchmarking that are MBPP, HumanEval and APPS.

- MBPP (Mostly Basic Python Problems) published by Austin *et al.* 2021 contains 974 crowdsourced Python programming problems with tests.
- HumanEval was developed in OpenAI byChen *et al.* 2021 with 164 hand-crafted problems, with a goal of avoiding data leakage (to ensure that the problems and golden solutions are not present in the training dataset).
- APPS by Hendrycks *et al.* 2021 features 10000 Python tasks with 131777 test cases, borrowed from open-access sites like Codewars and Codeforces.

These benchmarks are still used to this day for comparing performance of different models in scientific papers. Notably, APPS is a benchmark commonly used for fine-tuning LLMs for programming tasks, as it allows to use separate sets of problems for training and evaluation (Ben Allal *et al.* 2022).

The mentioned benchmarks became less effective, as newer models were trained on the same tasks they are being evaluated on. This phenomenon is known as **data leakage** as described byVendrow *et al.* 2025.

Amazon's Recode benchmark Wang *et al.* 2022 addressed this issue by introducing perturbations on docstrings, function names, and codes, while staying semantically close

to the original task. However, that is more of a way to test the robustness of the model, rather than a way to test its ability to solve brand-new problems.

More recent developments like HumanEval Pro and MBPP Pro Yu *et al.* 2024 introduced more sophisticated testing approaches. Their multistep evaluation process tests an LLM's ability to work with its own generated code. First, an LLM generates a solution to a known problem from HumanEval or MBPP datasets. Then, it is given a new task that requires calling a function generated in the first step. This approach revealed limitations in some models that perform well on simpler tasks.

The mentioned academic benchmarks focus on controlled and isolated tasks, SWE-bench (Jimenez *et al.* 2024) moved toward real-world scenarios. Software engineering tasks were taken from resolved Issues from GitHub repositories of open-source projects Python. SWE-bench is famous for its leaderboards, where laboratories and companies all over the world compete to achieve the higher percentage of solved tasks. But the benchmark is limited to tasks from only 12 open-source repositories, and only supports Python programming language.

Researchers Chi *et al.* 2025 have found a different way to evaluate LLMs in real-world scenarios. Instead of using a fixed set of tasks, they developed a plugin called CopilotArena for an IDE. The plugin provides two code completion options from different LLMs, and allows a user to choose the one they prefer. This approach allows for a more realistic evaluation of LLMs in coding tasks, but it lacks the controlled environment and a solid numerical result for each model. Such an approach could be useful for A/B testing of LLMs in production, but it is not suitable for scientific research and repeated evaluations during fine-tuning.

The benchmarks mentioned above perform in a static environment, where the model is given a task and expected to generate a solution. The InterCode framework byYang *et al.* 2023 introduces interactive environments using Docker. This enables evaluation of LLMs in realistic and interactive development scenarios with compilation and runtime feedback. The environments and scenarios were prepared for Python, SQL, and BASH, but the framework allows introducing new environments and scenarios. This approach more closely mirrors actual developer workflows and allows for testing LLMs in the role of a partially independent agent. But this approach comes with increased computational overhead of running a Docker environment, a virtual operating system, and an instance of a database, which limits the overall speed and a number of scenarios that can be tested at once.

Many of the mentioned benchmarks have inspired researchers to implement new benchmarks based on them. That could be their adaptations in other programming languages or refined datasets with verified and new hand-crafted tasks. Such as in the case of SWE-bench and following *SWE-bench Verified* and *Multi-swe-bench*.

2.2. Limitations of LLM Code Generation Benchmarks

Based on the analysis of existing benchmarks, we can identify several key limitations that our work aims to address:

- Benchmark saturation,

- Data leakage,
- Limited feedback,
- High resources' consumption,
- Environmental impact,
- Error-proneness of tasks,
- Limited feedback and output.

Let's address each of these limitations one by one.

2.2.1. Benchmark Saturation

When a benchmark becomes saturated, it means that the tasks in the benchmark are too easy for the current state-of-the-art LLMs, leading to high pass rates and diminishing returns on further improvements. It can be caused by either advances in LLMs or by data leakage, where the tasks and their solutions are present in the training datasets of the models being evaluated.

At some moment, testing on the simplest tasks becomes irrelevant, as all models pass them with high scores. Some datasets contain **metadata** that allows filtering out tasks based on their difficulty, thus saving resources and time on each evaluation.

2.2.2. Data Leakage

Benchmark saturation mentioned in the above sections is partially explained by advances in models, but it can also be attributed to information **leaking**: the popular and publicly available benchmarks appear in the training datasets accompanied by the golden solutions. This leads to a situation where the models are trained on the same tasks they are being evaluated on.

There are several ways to avoid the consequences of data leakage:

- hand-crafting brand-new tasks without publishing them or using for in-house training;
- generating new tasks based on the existing ones as it was done with HumanEval Pro and MBPP Pro;
- or perturbing existing tasks as it was done in ReCode.

2.2.3. High Cost and Environmental Impact

Repeated training and benchmarking of LLMs require significant computational resources, leading to significant electricity consumption and carbon emissions. This environmental impact is increasingly important in the context of global efforts to reduce carbon footprints. We will want for benchmarks to account for these factors and encourage more sustainable evaluation practices.

There are leaderboards that account for CO_2 emissions, such as Hugging Face*LLM CO2 emissions calculation - a Hugging Face Space — huggingface.co* s.f., which tracks

the carbon footprint of using models. However, these metrics are often not integrated into traditional benchmarks, leading to a lack of awareness about the environmental impact of LLM evaluation practices.

The most common metric in benchmarks is pass@k that measures the percentage of correct solutions among the k solutions generated by the model. This implies that for each task in the benchmark dataset, a model repeatedly generates a number of solutions, just to receive a single numeric result to use for a metric. This metric is used in ClassEval, MBPP, MathQA-Python, CoderEval, and HumanEval+. Notably, HumanEval and HumanEval+ use $k = 100$ (`pass@100`). However, as Miah and Zhu Miah y Zhu 2024 pointed out, users do not normally run the LLM several times, so pass@k does not reflect its usability.

2.2.4. Limited Feedback and Output

This limitation is intertwined with the high cost. The output of most benchmarks is a single numeric metric, such as pass@k, which indicates the percentage of tasks solved correctly by the model. Compared to the amount of work and energy that was consumed to produce this result, and the amount of information that could be extracted from the model's responses and test runs, this approach is very limited.

For example, SWE leaderboard *SWE-bench Leaderboards* s.f. is created based on a single number that does not reflect the types of tasks that the model is better or worse at solving Miah y Zhu 2024. Thus, a researcher or a user might choose a suboptimal model for their specific needs, resulting in lower performance or higher cost. This is partially countered by websites that aggregate results on several benchmarks *LLM Leaderboard 2025 — vellum.ai* s.f. which can give a very high-level picture.

Some of the ways to gather more information from the model's responses are:

- Gather performance metrics for each task, such as execution time, memory usage, and CPU load;
- Count the number of input and output tokens used for each generation to compare cost-effectiveness of models;
- Analyze the generated code for style and quality, such as cyclomatic complexity, number of lines, and code duplication;
- Provide a way to analyze individual task failures, such as incorrect solutions, timeouts, and exceptions;
- Using LLM-as-a-judge approach to evaluate the quality of the generated solution.

2.2.5. Error-Proneness of Tasks

When creating and managing big datasets, errors are inevitable. AsVendrow *et al.* 2025 found out, popular benchmarks contain up to 5 percent of mislabeled or erroneous tasks. This can lead to incorrect evaluation results and misinterpretation of model capabilities.

To mitigate this issue, a researcher should be able to examine the failures and more easily spot the errors in the tasks. This will also allow the researcher to spot patterns in model's errors, and possibly mitigate them by improving training datasets, updating a system prompt, and adjusting temperature and other parameters.

2.3. Problems in LLM Benchmarking

Benchmark saturation mentioned in the above sections is partially explained by advances in models, but it can also be attributed to information **leaking**: the popular and publicly available benchmarks appear in the training datasets accompanied by the golden solutions. Even then, it doesn't mean that an LLM won't struggle when presented with the same task. When changing the task phrasing while keeping the semantic consistent, there is a 4.5-percent drop in solvability, showing that the models remember the phrasing of the descriptions in the original dataset. Uniyal *et al.* 2024

One of the important aspects of LLM evaluation is the choice of the metrics. For code quality, there are BLEU, CodeBLEU, RUBY, ROUGE-L, METEOR, ChrF. They assess the similarity of the generated code to the golden solution, taking into account the properties of source code. Evtikhiev et al. Evtikhiev *et al.* 2023 takes 6 metrics, commonly used in papers. The authors conduct a study, comparing the results of metrics with human evaluation of the solutions. The results suggest that none of the analyzed metrics can correctly emulate human judgment, but ChrF metric is considered better than the others commonly used in papers.

A paper byCrupi *et al.* 2025looks into an approach of using LLM to evaluate the quality of the solution generated by another model (LLM-as-a-judge approach). As a result, they come to a conclusion that LLM-as-a-judge is a substantial improvement over mentioned metrics, and GPT-4-turbo can mimic closely a human evaluation.

2.4. Comparison of LLM Code Generation Benchmarks

Benchmark	**Size**	**Domain**	**Innovation**	**Limitations**
MBPP	974 tasks	Python	Crowdsourced, test cases	Leakage, basic
APPS	10,000+	Python	Large scale	Leakage, too ea
ReCode	3k	Python	Robustness via perturbation	Synthetic, limit
SWE-bench	2k	Python (12 repos)	Real GitHub issues	Limited repos/
HumanEval	164 tasks	Python	Handcrafted, leakage-avoidance	Small, saturate
HumanEval+	400+	Python	Extension of HumanEval	Still small, leak
HumanEval Pro	2,000+	Python	Multi-step tasks	Python-only, re
BigCodeBench	1M+	Multi	Massive scale	Hard to run, sa
InterCode	3k+	Python, SQL, Bash	Interactive Docker env	Heavy resource
CopilotArena	Unlimited	Unlimited	Real user experience	No numeric me

Tabla 2.1: Comparison of major LLM code generation benchmarks

2.5. Existing Benchmarking Frameworks and Their Limitations

Apart from the benchmarks themselves, there are several frameworks that facilitate LLM evaluation. These frameworks provide tools for running benchmarks and collecting results.

Two widely used benchmarking frameworks are **bigcode-evaluation-harness** Ben Allal *et al.* 2022 and **lm-evaluation-harness** Gao *et al.* 2024. Both provide tools for running standardized benchmarks on LLMs, but they have notable limitations. The lm-evaluation-harness is more general-purpose and supports a broader range of language tasks, yet it also relies on fixed task sets and lacks modularity for user-defined benchmarks. Neither framework provides built-in support for environmental metrics or fine-grained task selection, highlighting the need for more flexible and sustainable benchmarking solutions.

In the Table2.2 we compare the two frameworks based on their features and limitations.

Feature	**bigcode-evaluation-harness**	**lm-evaluation-harness**
Specialization	Majorly, code writing tasks, but also allows for documentation generation tasks and natural language reasoning tasks	A universal harness supporting a wide range of tasks
Included benchmarks	MBPP, MBPP+, DS-1000, MultiPL-E, Mercury, GSM8K, etc.	MBPP, MBPP+, HumanEval, SpanishBench, basqueGLUE, and many more.
Defining new tasks	Requires source code modification	Requires source code modification
Available configuration	Task dataset name, Number of tasks, Temperature, Saving LLM responses, Limit of LLM response	Task datasets list,
Run interface	CLI-based, no GUI	CLI-based, no GUI
Result analysis	Overall numeric metric and LLM responses saved as a file	
Visualization	No visualization tools	No visualization tools
Evaluation of multiple LLMs	One model per run	One model per run
Supports model loading via transformers	Yes	Yes

Tabla 2.2: Comparison of bigcode-evaluation-harness and lm-evaluation-harness

2.6. Research Questions Analysis

RQ1: What are the main limitations of current LLM benchmarks for code generation? We dug through the literature and found that the main limitations of current LLM benchmarks for code generation are: Benchmark saturation, Data leakage, Limited feedback, High resources consumption, Environmental impact, Error-proneness of tasks, Limited feedback and output.

RQ2: What metrics best reflect real-world usability and code quality? Out of the commonly used code quality metrics (BLEU, CodeBLEU, RUBY, ROUGE-L, METEOR, ChrF), the ChrF turns out to be the best-suited for code generation tasks, as it takes into account the properties of source code.

But an LLM-as-a-judge approach is considered a significant improvement over the mentioned metrics. When used with some of the modern models, an LLM can mimic closely a human evaluation.

RQ3: How can benchmarks be made customizable for different user needs? Existing frameworks like bigcode-evaluation-harness and lm-evaluation-harness provide a way to introduce new task datasets and metrics for specific user needs. However, they require source code modification and do not provide a user interface for configuring benchmarks. The reviewed frameworks also lack flexibility in task selection and filtering.

A solution with a friendly user interface can allow users to easily configure benchmarks, select relevant task types, and visualize results.

RQ4: What is the environmental impact of repeated benchmarking, and how can it be reduced? Benchmarks consume significant resources, but environmental impact is rarely measured. Including runtime, energy, and CO_2 reporting makes evaluation more responsible.

The analysis of existing benchmarks highlights both their contributions and their shortcomings, especially in terms of saturation, flexibility, and sustainability. These insights directly motivate the design of a new benchmarking framework, which addresses these limitations by focusing on modularity, customization, and eco-aware evaluation. In the following sections, we describe the design and implementation of this framework, as well as its evaluation through selected experiments.

CAPÍTULO 3

Problem Description

In the previous chapter, we looked at the existing popular benchmarking frameworks. Based on the list of their features, we can create a list of limitations that we can address in our work:

- A more broad and detailed output of the results, instead of simple numeric outputs.
- A flexible and interactive web interface for configuring benchmarks, selecting tasks, and visualizing results, or diving deeper into the failures.
- An easier approach for defining new tasks or modifying the existing ones, without the need to modify the source code of the framework.
- TO-DO

3.1. Analysis of Current Limitations

The limitations of existing benchmarks extend beyond just inefficiency. Studies like have identified several critical issues:

3.2. Requirements for a Solution

Based on the analyzed limitations and user needs, we identified key requirements for a more effective benchmarking approach:

Modularity:

- Support for multiple programming languages
- Ability to add/modify tasks easily
- Configurable testing criteria

Efficiency:

- Task filtering capabilities

- Optimized resource usage
- Quick feedback loops

User Experience:

- Interactive configuration interface
- Detailed result analysis
- Integration with development workflows

The benchmark should:

- Provide a way to analyse individual task failures
- Allow users to select tasks based on their specific needs
- Get the most information from each generated solution
- ...

CAPÍTULO 4

Proposed Solution

4.1. Architecture Overview

Our solution combines three main components:

- A Spring Boot backend with MVC architecture
- A React-based frontend for configuration and visualization
- Docker environments for isolated task execution

Figura 4.1: System Architecture Overview

4.2. Task Dataset Format

Tasks are defined in YAML format, allowing for easy modification and extension:

```
tasks:
  - name: "Example Task"
    type: "implementation"
    difficulty: "medium"
    languages: ["python", "java"]
    parameters:
      use_libraries: false
      generate_tests: true
```

4.3. Interactive Configuration

The React frontend provides: - Task filtering by type, difficulty, and language - Testing criteria selection - Resource usage configuration - Result visualization

CAPÍTULO 5

Implementation Details

5.1. Project Structure

The project follows a modern microservices architecture with separate backend and frontend applications. Currently, the configuration, status and result files are stored in a file-based storage system, but they can be easily adapted to use a database if needed.

```
Project Root
|-- src/             # Spring Boot application source code
|-- frontend/        # React application source code
|-- exec_configs/    # Execution configuration YAML files
|-- task_sources/    # Task source YAML files
|-- bench_status/    # Benchmark status files
`-- bench_results/   # Benchmark results
```

5.2. Backend Development

The Spring Boot application serves as the core of the system, providing:

5.2.1. REST API Endpoints

The backend exposes a comprehensive RESTful API:

- `/api/configs`: Configuration file management
- `/api/tasks`: Task source file operations
- `/api/benchmarks`: Benchmark execution control
- `/api/status`: Status monitoring for current benchmarks run
- `/api/results`: Result retrieval for visualization and analysis

5.2.2. Core Functionality

- Task processing and execution in isolated environments
- Docker container management for language-specific runtimes
- File-based storage system for configurations and results
- Integration with LLM judges through Spring AI

Figura 5.1: Backend Component Architecture

5.3. Frontend Development

The React frontend, built with TypeScript, provides an intuitive interface for:

5.3.1. User Interface Components

- Configuration file upload and management
- Task source file handling
- Real-time benchmark monitoring
- Result visualization and analysis

5.3.2. Interactive Features

The application enables users to:

- Upload and manage YAML configurations
- Configure benchmark parameters interactively
- Monitor benchmark progress in real-time
- Analyze and export results

Figura 5.2: Frontend Interface Overview

5.4. Deployment

Both components support flexible deployment options:

5.4.1. Local Development

```
# Backend
mvn spring-boot:run

# Frontend
cd frontend
npm install
npm run dev
```

5.4.2. Docker Deployment

The system includes Docker configurations for containerized deployment:

```
# Backend container
docker build -t ai-benchmark-backend -f Dockerfile.backend .
docker run -p 8080:8080 ai-benchmark-backend

# Frontend container
docker build -t ai-benchmark-frontend -f Dockerfile.frontend .
docker run -p 5173:5173 ai-benchmark-frontend
```

CAPÍTULO 6

Results and Evaluation

6.1. Performance Analysis

We evaluated our solution against traditional benchmarks:

Tabla 6.1: Comparison with Traditional Benchmarks

Key metrics:

- Execution time per task
- Resource usage (CPU, memory)
- Cost per benchmark run
- Environmental impact (estimated CO_2 emissions)

6.2. Case Studies

We present three scenarios demonstrating the system's flexibility:

1. Fine-tuning an LLM for specific programming tasks
2. Evaluating code style consistency
3. Testing real-world project integration

CAPÍTULO 7

Conclusions

7.1. Contributions

Our modular benchmark approach offers several advantages:

- Customizable evaluation scenarios
- Reduced resource consumption
- Improved feedback quality
- Easy integration with development workflows

7.2. Future Work

Potential improvements include:

- Additional programming language support
- Enhanced visualization tools
- CI/CD pipeline integration
- Advanced metrics collection

7.3. Impact

This work contributes to more sustainable and efficient LLM evaluation practices, potentially reducing both costs and environmental impact while providing more meaningful results.

Bibliografía

1. J. Vendrow, E. Vendrow, S. Beery y A. Madry, *Do Large Language Model Benchmarks Test Reliability?*, 2025, arXiv: 2502.03461 [cs.LG], (https://arxiv.org/abs/2502.03461).
2. J. Austin *et al.*, Program Synthesis with Large Language Models. *arXiv preprint arXiv:2108.07732* (2021).
3. M. Chen *et al.*, *Evaluating Large Language Models Trained on Code*, 2021, arXiv: 2107.03374 [cs.LG], (https://arxiv.org/abs/2107.03374).
4. D. Hendrycks *et al.*, Measuring Coding Challenge Competence With APPS. *NeurIPS* (2021).
5. L. Ben Allal *et al.*, *A framework for the evaluation of code generation models*, https://github.com/bigcode-project/bigcode-evaluation-harness, 2022.
6. S. Wang *et al.*, ReCode: Robustness Evaluation of Code Generation Models. (https://arxiv.org/abs/2212.10264) (2022).
7. Z. Yu, Y. Zhao, A. Cohan y X.-P. Zhang, *HumanEval Pro and MBPP Pro: Evaluating Large Language Models on Self-invoking Code Generation*, 2024, arXiv: 2412.21199 [cs.SE], (https://arxiv.org/abs/2412.21199).
8. C. E. Jimenez *et al.*, *SWE-bench: Can Language Models Resolve Real-World GitHub Issues?*, 2024, arXiv: 2310.06770 [cs.CL], (https://arxiv.org/abs/2310.06770).
9. W. Chi *et al.*, *Copilot Arena: A Platform for Code LLM Evaluation in the Wild*, 2025, arXiv: 2502.09328 [cs.SE], (https://arxiv.org/abs/2502.09328).
10. J. Yang, A. Prabhakar, K. Narasimhan y S. Yao, *InterCode: Standardizing and Benchmarking Interactive Coding with Execution Feedback*, 2023, arXiv: 2306.14898 [cs.CL], (https://arxiv.org/abs/2306.14898).
11. *LLM CO2 emissions calculation - a Hugging Face Space — huggingface.co*, https://huggingface.co/docs/leaderboards/open_llm_leaderboard/emissions, [Accessed 17-08-2025].
12. T. Miah y H. Zhu, *User Centric Evaluation of Code Generation Tools*, 2024, arXiv: 2402.03130 [cs.SE], (https://arxiv.org/abs/2402.03130).
13. *SWE-bench Leaderboards*, https://www.swebench.com/, [Accessed 17-08-2025].
14. *LLM Leaderboard 2025 — vellum.ai*, https://www.vellum.ai/llm-leaderboard, [Accessed 17-08-2025].
15. M. Uniyal *et al.*, presentado en Findings of the Association for Computational Linguistics: EMNLP 2024, págs. 15397-15402.

16. M. Evtikhiev, E. Bogomolov, Y. Sokolov y T. Bryksin, Out of the bleu: how should we assess quality of the code generation models? *Journal of Systems and Software* **203**, 111741 (2023).

17. G. Crupi *et al.*, On the Effectiveness of LLM-as-a-judge for Code Generation and Summarization. *IEEE Transactions on Software Engineering* (2025).

18. L. Gao *et al.*, *The Language Model Evaluation Harness*, ver. v0.4.3, jul. de 2024, (`https://zenodo.org/records/12608602`).

Apéndices

APÉNDICE A

Ejemplo de anexo

Anexo primero.