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:

Benchmarks are widely used for measuring and comparing performance of LLMs in different areas, including logical puzzles, factual accuracy, and coding tasks among others. The use of benchmarks allows developers to fine-tune a model, and a user to choose a model that better fits their needs or adjust its responses by changing a system prompt and parameters like temperature. This requires a lot of benchmark runs. Running a benchmark is costly, time- and energy-consuming. Also, it is often inefficient, as they may contain problems that are not relevant for the use case of a model being adjusted.

So the goal of this work is to explore existing benchmarks, and design and develop an approach to benchmarking that is highly customizable and extendible, time-efficient, eco-friendly. As a result, a modular benchmark was developed, that allows to add and modify test cases using a user interface, and configure a benchmark run using task filters and toggles for enabling or disabling specific checks. Also, the output of the benchmark allows to examine the results in detail.

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: modular and customizable benchmarks 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 subsequent research were guided by the following questions:

- What are the main limitations of current LLM benchmarks for code generation?
- What metrics best reflect real-world usability and code quality?
- How can benchmarks be made customizable for different user needs?
- 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

Running benchmarks is essential for LLM developers, researchers and users. Developers and researchers need benchmarks to fine-tune their models or compare approaches, while users rely on them to select appropriate models and adjust parameters like temperature and system prompts. However, this process faces several challenges: Cost and Resource Consumption: Each benchmark run consumes significant computational resources, time, and energy. As noted in recent studies, the environmental impact of repeated benchmark runs is becoming a growing concern.

Lack of Customization: Most benchmarks are developed as fixed sets of tasks, making them difficult to adapt for specific use cases. This limitation was highlighted in the SWE-bench study Jimenez *et al.* 2024, which emphasized the need for more flexible evaluation methods.

Inefficient Task Selection: Many benchmarks include tasks that may be irrelevant for specific applications, leading to wasted resources. Recent research Vendrow *et al.* 2025 has shown that benchmark saturation often results in running unnecessary tests that all modern LLMs pass easily.

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

CAPÍTULO 2

State of the Art

2.1. Evolution of Code Generation Benchmarks

The landscape of LLM benchmarks has evolved significantly since the introduction of HumanEval Chen *et al.* 2021. This pioneering benchmark set a standard for evaluating code generation capabilities, but its limitations became apparent as LLMs advanced.

Recent developments like HumanEval Pro Yu *et al.* 2024 have introduced more sophisticated testing approaches. For instance, their multi-step evaluation process tests an LLM's ability to work with its own generated code, revealing limitations in some models that perform well on simpler tasks.

2.2. Types of LLM Benchmarks

While academic benchmarks like HumanEval focus on controlled, isolated tasks, initiatives like SWE-bench Jimenez *et al.* 2024 have moved toward real-world scenarios. This shift reflects a growing recognition that LLM evaluation should encompass the complexity of actual software development.

The InterCode framework Yang *et al.* 2023 introduced interactive environments using Docker, enabling evaluation of LLMs in realistic development scenarios with compilation and runtime feedback. This approach more closely mirrors actual developer workflows but comes with increased computational overhead.

In Chi *et al.* 2025 the authors developed a plugin for IDE that offers the user two code completion options from two different LLMs and records which option the user chose. This allowed them to compare the performance of different LLMs in real-world coding tasks, providing valuable insights into user preferences and model effectiveness. Another article Mozannar *et al.* 2024 finds out that a user can often accept a first proposed solution in order to see it with proper syntax highlighting and be able to understand it better. But later he could even remove the suggestion and waits for a repeated completion or simply writes the piece of the code themself. Although that should not be considered a significant flaw of the benchmark, it could partially skew the results.

Most benchmarks are very limited in the output they provide. 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 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.

2.3. Problems in LLM Benchmarking

Benchmark saturation mentioned above 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 with 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 percet 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. However, the authours 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 by Crupi *et al.* 2025 looks 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. Environmental impact of Benchmarking

Repeated training and benchmarking of LLMs require substantial 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 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.

2.5. 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.

Feature	**bigcode-evaluation-harness**	**lm-evaluation-harness**
Evaluation of multiple LLMs	One model per run	One model per run
Available configuration	Task dataset name, Number of tasks, Temperature, Saving LLM responses, Limit of LLM response	TO-DO
Included benchmarks	Common benchmark tasks	Limited, fixed task sets
Defining new tasks	Requires source code modification	Requires source code modification
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

Tabla 2.1: Comparison of bigcode-evaluation-harness and lm-evaluation-harness

CAPÍTULO 3

Problem Description

3.1. Analysis of Current Limitations

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

- Benchmark saturation: Many tasks become irrelevant as models improve Vendrow *et al.* 2025
- Task leaking: LLMs are trained on benchmark tasks solutions, making them obsolete Vendrow *et al.* 2025
- Limited feedback: Most benchmarks provide only pass/fail results
- High costs: Running comprehensive benchmarks is expensive and time-consuming
- Environmental impact: Repeated runs contribute to unnecessary carbon emissions
- Inflexibility: Fixed task sets don't adapt to specific needs
- Error-prone: benchmarks contain up to 5 percent of mislabelled or erroneous tasks Vendrow *et al.* 2025

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:

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

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
- `/api/results`: Result retrieval 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. 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`).
2. 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`).
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. 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`).
5. 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`).
6. 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`).
7. H. Mozannar, G. Bansal, A. Fourney y E. Horvitz, *Reading Between the Lines: Modeling User Behavior and Costs in AI-Assisted Programming*, 2024, arXiv: `2210.14306 [cs.SE]`, (`https://arxiv.org/abs/2210.14306`).
8. *SWE-bench Leaderboards*, `https://www.swebench.com/`, [Accessed 17-08-2025].
9. T. Miah y H. Zhu, *User Centric Evaluation of Code Generation Tools*, 2024, arXiv: `2402.03130 [cs.SE]`, (`https://arxiv.org/abs/2402.03130`).
10. *LLM Leaderboard 2025 — vellum.ai*, `https://www.vellum.ai/llm-leaderboard`, [Accessed 17-08-2025].
11. M. Uniyal *et al.*, presentado en Findings of the Association for Computational Linguistics: EMNLP 2024, págs. 15397-15402.
12. 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).
13. G. Crupi *et al.*, On the Effectiveness of LLM-as-a-judge for Code Generation and Summarization. *IEEE Transactions on Software Engineering* (2025).
14. *LLM CO2 emissions calculation - a Hugging Face Space — huggingface.co*, `https://huggingface.co/docs/leaderboards/open_llm_leaderboard/emissions`, [Accessed 17-08-2025].

15. L. Ben Allal *et al.*, *A framework for the evaluation of code generation models*, `https://github.com/bigcode-project/bigcode-evaluation-harness`, 2022.

16. 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.