UNIVERSIDAD DE MÁLAGA

E.T.S. INGENIERÍA INFORMÁTICA
UNIVERSIDAD DE MÁLAGA

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

Realizado por
Donat Shergalis

Tutorizado por
Gabriel Jesús Luque Polo

Departamento
Departamento de Lenguajes y Ciencias de la Computación
UNIVERSIDAD DE MÁLAGA

MÁLAGA, septiembre de 2025

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

Fecha defensa:

El Secretario del Tribunal

Agradecimientos

Página de agradecimientos.

Resumen

Los benchmarks juegan un papel crucial en el desarrollo, evaluación y selección de modelos para su uso práctico. Los benchmarks y frameworks existentes presentan limitaciones. Diferentes benchmarks se compensan parcialmente entre sí, pero ninguno es universal: los problemas incluyen saturación, fuga de datos, métricas y resultados limitados, y altos costos energéticos y computacionales.

Proponemos un enfoque para extraer la máxima información útil de cada ejecución de benchmark, aumentando la comprensión para investigadores y profesionales, al mismo tiempo que se reduce el impacto ambiental. Al hacer las ejecuciones más informativas, se pueden evitar repeticiones innecesarias, disminuyendo el consumo de energía. El enfoque permite la selección flexible de tareas, configuraciones detalladas e integración de múltiples criterios de evaluación en una sola ejecución. Se implementó un prototipo para demostrar la viabilidad, validar la funcionalidad y examinar las limitaciones.

Palabras Clave: LLM, modelos de lenguaje grande, benchmarks, marco de evaluación, impacto ambiental, consumo de energía, saturación de benchmarks, fuga de datos, LLM-como-juez

---

Abstract:

Benchmarks play a crucial role in developing, researching, and evaluating models for practical use. Existing benchmarks and frameworks have limitations. Different benchmarks partially compensate for each other, but none is universal: issues include saturation, data leakage, limited metrics and outputs, and high energy and computational costs.

We propose an approach to extract the maximum useful information from each benchmark run, enhancing insights for researchers and practitioners while reducing environmental impact. By making benchmark executions more informative, redundant runs can be avoided, lowering energy consumption. The approach enables flexible task selection, fine-grained configuration, and integration of multiple evaluation criteria in a single execution. A prototype was implemented to demonstrate feasibility, validate functionality, and examine limitations.

Keywords: LLM, large language models, benchmarks, evaluation framework, environmental impact, energy consumption, benchmark saturation, data leakage,LLM-as-a-judge

Table of Contents

1 Introduction

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

This work explores existing benchmarks for code-generating LLMs and proposes a novel approach: an easily customizable benchmark with detailed, easily analyzable 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.
- Consideration of environmental and cost factors in the benchmarking process.
- An interactive web interface for configuring benchmarks and analyzing results.

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, this thesis addresses the lack of a flexible, customizable, and sustainable benchmarking framework for evolving LLM capabilities and user needs.

1.2. Objectives

The main objectives of this work are:

1. Conduct a detailed analysis of existing benchmarks and identify their limitations.
2. Design a modular benchmarking system that enables:
    - Custom task selection and filtering.
    - Configuration of evaluation criteria.
    - Support for multiple programming languages and task types.
    - Integration with CI/CD pipelines for automated benchmarking.
3. Implement an interactive web interface for benchmark configuration and result analysis.

4. Develop a cost-efficient and environmentally conscious approach to benchmarking.

The goal of this work is not to compete with large-scale frameworks, but to present a prototype that illustrates a different approach: a benchmarking tool for LLMs with a graphical user interface and configuration-driven design. Instead of hard-coded pipelines, users can flexibly define tasks, parameters, and evaluation criteria through configuration files generated using the UI.

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 reduces time and energy consumption but also enhances the practical relevance of benchmarking results.

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 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 at OpenAI by Chen *et al.* 2021 with 164 hand-crafted problems, aiming to avoid data leakage (ensuring that the problems and golden solutions are not present in the training dataset).
- APPS by Hendrycks *et al.* 2021 features 10,000 Python tasks with 131,777 test cases, borrowed from open-access sites like Codewars and Codeforces.

These benchmarks are still used today for comparing the performance of different models in scientific papers. Notably, APPS benchmark is commonly used for fine-tuning LLMs for programming tasks, as it allows the use of separate sets of problems for training and evalua-

tion (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 by Vendrow *et al.* 2025.

Amazon's Recode benchmark Wang *et al.* 2022 addressed this issue by introducing perturbations on docstrings, function names, and code, while staying semantically close to the original task. However, this is more a way to test the robustness of the model rather than 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, while 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 Python projects. SWE-bench is famous for its leaderboards, where laboratories and companies worldwide compete to achieve the highest percentage of solved tasks. However, the benchmark is limited to tasks from only 12 open-source repositories and supports only the 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 a controlled environment and a solid numeric 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 by Yang *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. However, 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 the number of scenarios that can be tested at once.

Many of the mentioned benchmarks have inspired researchers to implement new benchmarks based on them. These could be adaptations in other programming languages or refined datasets with verified and new hand-crafted tasks, such as in the case of SWE-bench and the 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.

In continuation, we will 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. As Vendrow *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.* 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 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. Comparison of LLM Code Generation Benchmarks

Benchmark	**Size**	**Innovation**	**Limitations**
MBPP	974 tasks	Crowdsourced, test cases	Leakage, basic tasks
APPS	10,000+	Large scale	Leakage, too easy for modern
ReCode	3k	Robustness via perturbation	Synthetic, limited
SWE-bench	2k	Real GitHub issues	Limited repos/langs
HumanEval	164 tasks	Handcrafted, avoids leakage	Small, saturated
HumanEval+	400+	Extension of HumanEval	Still small, leakage
HumanEval Pro	2,000+	Multi-step tasks	Python-only, resource-heavy
BigCodeBench	1M+	Massive scale	Hard to run, saturation risk
InterCode	3k+	Interactive Docker env	Heavy resources, complex
CopilotArena	Unlimited	Real user experience	No numeric metric or controlled env

Tabla 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 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, Limits of LLM response, etc.	Task datasets list (other parameters are is defined on task level), Limits of LLM response, System prompt, etc.
Extended output	LLM response or references as JSON	Prompt, LLM response, and metrics results as JSON
Run interface	CLI-based, no GUI	CLI-based, no GUI, an API for training loops
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
Caching of LLM generations	No	Yes with an argument

Tabla 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? The literature analysis showS 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.

3

Design and Implementation of the Modular Benchmarking Framework

3.1. Prototype Goal and Scope

The goal of the prototype is to provide a controlled environment for running benchmarks and analyzing their outputs.

The key idea is to separate three concerns: (1) definition of *task sources* and *execution configurations*, (2) execution and tracking of benchmarks, (3) presentation and inspection of results. This separation enables the use of different combinations of task sources and execution configurations.

The scope of the prototype is limited to managing benchmark runs, collecting outputs, and exposing them through a simple interface. It does not aim to provide large-scale distributed evaluation, advanced analytics, or integration with external benchmark platforms. These aspects remain outside the present work.

3.2. System Design

This section describes the architecture of the prototype, its main components, and the workflow of interactions between the user, the frontend, and the backend. The focus is on modularity, extensibility, and clarity of responsibilities. The system operates directly on files (task sets, benchmark configurations, and results) instead of a database. This choice makes it easy to share datasets and configurations, edit them outside of the web interface, and execute benchmarks directly from the CLI.

3.2.1. Overview

The system consists of two main entry points:

- **Web interface**, built with Spring MVC, that allows a user to browse files, edit them, launch benchmarks, and inspect results.
- **Console interface**, built with Spring Framework, which allows running benchmarks directly from the terminal or from CI/CD pipelines.

Both entry points communicate with the **benchmark executor**, which is modular and extensible by design. Its components are:

- **Controller class**, responsible for orchestrating benchmark execution.
- **LLM communication layer**, implemented with Spring AI, which abstracts interactions with different LLM providers.
- **Quality evaluation layer**, consisting of modules for code quality checks (e.g., PMD, Checkstyle, SonarQube, and LLM-as-a-judge).
- **Test execution layer**, currently implemented for in-memory Java execution, but extendable to remote execution or containerized execution via Docker.

3.2.2. Technologies

The choice of technologies is motivated by their suitability for modularity and integration:

- **Spring MVC** — provides a clean abstraction for implementing REST endpoints and handling requests from the frontend.
- **Spring Framework (console interface)** — allows reuse of the same components for CLI execution and CI/CD integration.
- **Spring AI** — unifies access to multiple LLM APIs, simplifying the addition of new providers.
- **PMD, Checkstyle, SonarQube** — widely used static analysis tools available for Java, which make it possible to measure code quality with established metrics.
- **Docker** — offers an isolated environment for executing arbitrary languages and tools, ensuring reproducibility and security.

3.2.3. Workflow

At a high level, the workflow is as follows:

1. The user opens the frontend.
2. The frontend requests from the backend the list of files available in three folders: *configs*, *tasks*, and *results*.
3. The user selects a file; the frontend fetches it from the backend.
4. The user edits a task file in the frontend and saves it back to the backend.
5. The user edits a config file in the frontend and saves it back to the backend.
6. The user launches a benchmark by selecting one config and a set of task files; the frontend sends this request to the backend.
7. The backend delegates execution to the *worker*, which starts running tasks and returns a run identifier.
8. The frontend polls the backend with the run identifier to fetch status (e.g., number of tasks completed per file).
9. Once status is no longer in-progress, the frontend informs the user and stops polling.

10. The user requests a result file, the frontend fetches it from the backend, and presents statistics, errors, and detailed outputs.

3.2.4. Sequence Diagram

The interaction described above is shown as a sequence diagram (Appendix 1).

3.2.5. Execution Configuration File

The execution configuration (`exec-config.yaml`) defines the parameters under which a benchmark run is executed. It is a structured YAML file that provides a modular way to select tasks, models, and evaluation criteria without changing the core logic of the system. This design allows the user to combine different *task sources* with different *execution configurations*, enabling flexible experimentation. A full example of the file is provided in the Appendix, while here we describe its structure and purpose.

General structure. The file contains the following top-level sections:

- **Version** — format version of the configuration file, used for compatibility checks.
- **Difficulties** — a list of difficulty levels of tasks to include (e.g., *easy*, *medium*, *hard*).
- **Areas** — a placeholder section to restrict tasks to certain thematic domains.
- **Languages** — programming languages targeted by the benchmark (e.g., Java, Python).
- **Parameters** — a set of boolean switches controlling how the LLM is expected to generate and use code, tests, or libraries.
- **Criteria** — a list of evaluation criteria that will be applied to candidate solutions. Each criterion can be individually enabled or disabled.
- **LLMs** — a list of model identifiers to be benchmarked (including provider and model version).

Here's the brief example of how the file looks like:

```
version: 1.0
difficulties: [easy, medium, ]
areas: [math, ]
languages: [python, ]
parameters:
  - name: use-llm-judge
    # if we want to check the results with an LLM-judge
    enabled: true
  - name: all-tests-public
    # if we want to give the LLM all tests as a reference
    enabled: true
  - name: should-write-comments
    # if we want the solution to contain comments
    enabled: false
criteria: # list of criteria to evaluate the solutions
  - name: unit-test
    # only for languages that we can run in a sandbox
    enabled: true
  - name: ram-usage
    # works only if unit_test criteria is enabled
    enabled: true
  - name: llm-judge-code-quality
    enabled: true
  - name: sonarqube
    enabled: true
llms: [mistralai/devstral-small:free, ]
```

In this case, the benchmark will run tasks of easy and medium difficulty in the math area. It will only select tasks that list Python in the list of supported languages, and provides a prompt for it.

For each task, the benchmark will build a prompt from a FreeMarker template using the task description and parameters. Some parameters will be used in prompt building. For example, the *all-tests-public* means when tests are divided into public and hidden, the LLM will receive all available tests anyway. And the prompt will mention that the solution doesn't have to contain comments.

The prompt will be sent to all the mentioned LLMs.

After receiving a solution from the LLM, the benchmark will evaluate it using unit tests, measure RAM usage during test execution, analyze code quality with SonarQube, and use an

LLM-as-a-judge to assess code quality. The unit-tests themselves can use the list of parameters to adjust their behavior. For example, if the *should-write-comments* parameter is enabled, the tests can check for the presence of comments in the code.

Now we will review each section in more detail. Also, in the following section about the Task Source file, we will see how the metadata of tasks is set, and how parameters are used in building prompts and tests.

Parameters. The `parameters` section contains toggles that modify how code generation and testing should behave. Examples include whether the LLM should generate its own unit tests, whether it should receive reference tests, or whether it is allowed to use external libraries. These options make it possible to simulate different levels of information and tooling that a model may rely on.

Criteria. The `criteria` section specifies which metrics and tools are used to evaluate solutions. The framework supports both static analysis tools (*PMD, Checkstyle, SonarQube*), dynamic execution (*unit tests, coverage via JaCoCo, resource usage*), and LLM-based judgment (*LLM-as-a-judge for code quality and comments*). Each criterion can be individually activated depending on the language and desired evaluation scope. This modular approach makes it easy to extend the benchmark with new metrics.

LLM selection. Finally, the `llms` section lists one or more models to be benchmarked. Each entry corresponds to a model identifier, which may include provider-specific information and quota (e.g., free-tier models). Multiple models can be specified in one configuration file, allowing direct comparison under identical conditions.

Additional advantages of the approach. By keeping all execution settings in a declarative YAML file, users can:

1. Share and reproduce benchmark configurations easily. Shared environment means that the researchers use the resources more optimally.

2. There is no need to set up the benchmarking in each workstation, as they can be used as thin clients.

3. Every user can access the same version of the benchmark, the same versions of datasets and configurations, ensuring consistency in experiments.

4. Run the same benchmark setup both via the web interface and the CLI.

5. Extend the framework by adding new parameters, criteria, or model identifiers without changing the core code.

6. Secrets for APIs are managed in a centralized way, so there is no risk of leaking them by accident.

3.2.6. Task Source File

Task sources define the pool of tasks available for execution. Each source is represented as a YAML file that contains task definitions together with metadata, prompts, tests, and reference solutions. The system can combine multiple task sources with different execution configurations, enabling flexible benchmarking setups.

```
version: 1.0
name: task-source-example-1
tasks:
 -
 name: highest common factor
 difficulty: easy
 area: math
 source: MBPP    # additional metadata
 languages: [python, ]
 available_parameters:
    [use-llm-judge, all-tests-public, all-tests-hidden]
 available_criteria:
    [unit-test, llm-judge-code-quality, sonarqube]
 task:
  common_prompt: |
    Write a function in ${language} to calculate the highest common
    factor of two numbers.
    <#if !parameters['all-tests-hidden'] >
        Here are the tests: ${public_tests}
        <#if parameters['all-tests-public'] > ${hidden_tests} </#if>
    </#if>

  languages_specific:
    python:
      description: ${common_prompt}
      public_tests:
        - code: |
            result = ${solution.function_name}(10, 15)
            assert result == 5

  golden_solution:
    pseudocode: "..." # can be used for LLM-as-a-judge or reference

  llm_judge_prompt: |
    You are an experienced interviewer...
```

The YAML structure starts with global metadata, including version and source name. Individual tasks are described with attributes such as task name, type, difficulty, area, and benchmark of origin. Metadata fields are later used to filter and sort tasks during experiment configuration.

Each task includes available parameters and criteria. Parameters specify options that can be toggled at execution time, such as whether to reveal all tests or to involve an LLM judge. Cri-

teria define the evaluation methods that can be applied, including both programmatic checks (e.g., unit tests, code style) and LLM-based assessments.

Prompts are built using `FreeMarker` templates. Templates take parameters from the execution configuration and integrate them into task instructions. This allows the same task to be instantiated in multiple forms, depending on the chosen setup. Prompts are divided into three categories: a *common prompt*, *language-specific prompts*, and an *LLM judge prompt*.

Tests are defined alongside prompts and may also use parameters. They can validate functional correctness, adherence to constraints (such as allowed libraries), or style requirements. Both public and hidden tests are supported, giving flexibility in how much reference material is provided to the model.

Finally, each task may contain golden solutions, expressed in one or more programming languages or pseudocode. These serve as reference implementations and may be used for validation or as baseline solutions.

3.2.7. Benchmark Criteria and Metrics

The benchmark supports multiple criteria for evaluating generated solutions. Each criterion is implemented as a separate Spring Component, which allows for easy extension by adding new criteria classes.

Each criterion class implements a common interface with methods for filtering applicable tasks and executing the evaluation. Filtering is based on execution configuration (if this criterion is explicitly enabled) and task metadata: for example, by the programming language.

The supported criteria include:

- **Unit tests in Java** — runs the provided unit tests against the generated code using in-memory Java compiler;
- **CPU usage** — measures CPU time during test execution in milliseconds;
- **LLM-as-a-judge for code quality** — uses an LLM to evaluate the quality of the generated code based on a prompt;

- **PMD** — analyzes the generated code using PMD and collects metrics such as code smells, complexity, and potential bugs, calculated from code smells and complexity, with different weights assigned to different types of issues, and normalized to a 0-1 scale;
- **Tokens count** — counts the number of input and output tokens used during LLM interaction to estimate cost-effectiveness;
- **Checkstyle** — analyzes the generated code for JVM languages using Checkstyle and collects metrics such as style violations and formatting issues, with different weights assigned to different types of issues, and normalized to a 0-1 scale.

Criteria to be implemented in the future include:

- **Test Execution in Docker** — runs the generated code in a Docker container to support multiple programming languages and isolate execution environments;
- **Code quality with SonarQube** — analyzes the generated code using SonarQube and collects metrics such as code smells, bugs, vulnerabilities, and technical debt;
- **Memory usage** — measures peak memory consumption during test execution;
- **Code coverage with JaCoCo** — measures the percentage of code covered by unit tests;
- **Language-specific style checks** — analyzes the generated code using Pyright and similar tools and collects metrics such as style violations and formatting issues;
- **LLM-as-a-judge for comments** — uses an LLM to evaluate the quality of comments in the generated code based on a prompt.

Each criterion produces a structured LLM Response Evaluation Result object that includes:

- **Executor class** — the name of the class that performed the evaluation;
- **Execution ID** — a unique identifier for the execution, shared
- **Criteria** — the name of the criterion;
- **Score** — the numeric result of the evaluation (e.g., 0 or 1 for tests, a value for metrics);
- **Unit** — the unit of measurement for the score (e.g., ”success”for tests, ”ms”for time, ”bytes”for memory);

- **Output** — any additional output produced during the evaluation;
- **Error** — any error message if the evaluation failed;
- **Time millis** — the time taken to execute the evaluation in milliseconds.
- **Test case ID** (only for unit tests) — the identifier of the specific test case;
- **Solution and test case code** (only for unit tests) — the prepared source code that was executed;

Each object is associated with a specific task and LLM response. This allows for detailed analysis of results across different dimensions. For example, one can see how a particular model performed on a specific task across multiple criteria, or compare the performance of different models on the same task. For each test failure, the output includes the output and the exact code that was executed, making it easier to debug issues.

3.3. Results and Evaluation

After implementing the prototype, we compare it with existing frameworks and discuss its advantages, limitations, and future work.

3.3.1. Experimental Runs

To demonstrate the functionality of the prototype, we conducted several experimental runs using different configurations and task sources. We selected a subset of tasks from the HumanEval dataset, focusing on easy and medium difficulty levels in Java.

The execution configuration file specified the use of two LLMs: `mistralai/devstral-nemo` and `mistralai/devstral-small`. We enabled the following criteria for evaluation:

- Unit tests measured in percentage of passed tests;
- CPU usage during test execution measured in milliseconds;
- Token count for input and output measured in number of tokens;
- LLM-as-a-judge for code quality;

- PMD codestyle analysis.

The benchmark was executed via the web interface, allowing us to monitor progress and view results in real-time. The results were collected in structured YAML files, which included detailed outputs for each task and criterion.

Table 3 shows a summary of the results:

Criteria	**Unit**	**mistral-nemo**	**devstral-small-2505**
Parameters count		12B	24B
Release date		July 2024	May 2025
llm-judge-code-quality	score	69.5	70.5
llm-judge-solution-correctness	score	84.5	86.67
java-pmd	1/errors	0.84	0.8
token-count	tokens	268.33	454.5
unit-test	success	67 %	100 %
cpu-usage	ms	0.12	0.57

Tabla 3: Summary of benchmark results for two LLMs on selected tasks

TO-DO: more models and tasks, maybe also with other datasets and configurations

So as we can see from this table, the smaller and newer model outperformed the larger and older one across code quality and the amount of tests passed. However, it used significantly more tokens and CPU time, indicating a trade-off between performance and efficiency. In general, the results are consistent with expectations based on the models descriptions in *Models Benchmarks | Mistral AI* s.f..

The UI allows us to inspect detailed results for each task and criterion. For example, by clicking on the unit-test criterion, we can see which specific tests were passed or failed, along with the exact code that was executed. For example, we can take a closer look at the task 1 in HumanEval for Java which caused an error for mistral-nemo model. We see that there is an AssertionError that occurred on line 20 of a unit-test. We can understand that the code was compiled and executed successfully, but the result was incorrect due to an error in logic, and can inspect the code closer if we wish so.

This level of investigation would not be possible without having the rich output, and would be more difficult without having a UI for browsing them.

Thus, this example demonstrates its effectiveness of the benchmarking framework in providing detailed insights into model performance across multiple dimensions.

3.3.2. Comparison with Existing Frameworks

Table 4 compares the proposed framework with bigcode-evaluation-harness and lm-evaluation-harness.

Feature	**bigcode-evaluation-harness**	**lm-evaluation-harness**	**proposed benchmarking framework**
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	Ecologically concious, taking the most information out of a test and a generated solution
Included benchmarks	MBPP, MBPP+, DS-1000, MultiPL-E, Mercury, GSM8K, etc.	MBPP, HumanEval, SpanishBench, basqueGLUE, and many more.	HumanEval for Python and Java, MBPP.
Adding tasks, datasets	Requires source code modification	Requires source code modification	With UI or by editing a YAML file editing
Available configuration	Task dataset name, Number of tasks, Temperature, Saving LLM responses, Limits of LLM response, etc.	Task datasets list (other parameters are is defined on task level), Limits of LLM response, System prompt, etc.	Task dataset and filters (difficulty, domain, language), Criteria to use (unit-tests, token count, code quality, etc.), Caching generations, test code in prompts
Extended output	LLM response or references as JSON	Prompt, LLM response, and metrics results as JSON	Detailed with generations results, metrics, test assertion errors
Run interface	CLI-based, no GUI	CLI-based, no GUI, an API for training loops	Web UI and CLI
Result analysis	Overall numeric metric and LLM responses saved as a file	Overall numeric metric	Detailed for each metric: with evaluation details and generated solutions
Visualization	No visualization tools	No visualization tools	An UI to edit datasets, configure runs, browse its results
Multiple LLMs	One model per run	One model per run	Multiple models per run
Model loading via	Yes	Yes	No

3.4. Results Discussion and Limitations

The current prototype demonstrates that the proposed approach is viable and already supports the core functionalities needed to run benchmarks and collect meaningful results. At the same time, it remains a prototype and lacks several features required for reliable use in real research and production settings.

First, the execution environment is not sufficiently isolated. Changes in the host machine may affect execution time and distort metrics such as CPU usage. A more robust sandboxing mechanism is needed to ensure reproducibility. Moreover, the system currently supports only a subset of programming languages. Extending test execution to other widely used languages would broaden applicability.

Second, while the prototype allows shared use on a common server, it raises concerns about data consistency and security. Proper authorization, as well as locking mechanisms for files being edited, should be implemented to support collaborative scenarios safely.

Third, although the user interface already covers the essential operations, further improvements are necessary to make it more convenient and user-friendly.

Finally, before results produced by this benchmark can be used in scientific studies, datasets need to be expanded and standardized. Feedback from experienced researchers and developers will also be critical for validating the approach and guiding its further development.

4

Conclusion and Future Work

Benchmarks play a central role in the development of large language models and in guiding their use in real-world applications. In this work, we analyzed existing benchmarks and harnesses and found that each approach has its own limitations. Benchmarks become saturated, suffer from data leakage, support only a limited set of metrics or languages, and most consume significant energy and computational resources. Harnesses, in turn, often lack detailed analysis of results and do not provide convenient mechanisms for comparing models across multiple metrics. Different benchmarks partially compensate for each other, but none of them is universal.

To address these issues, we proposed an approach that maximizes the insights gained from each benchmark run. By combining multiple evaluation criteria within a single execution, the framework increases the usefulness of results for researchers and practitioners, while at the same time reducing the environmental impact of benchmarking by limiting redundant runs. To demonstrate the feasibility of this idea, we implemented a prototype system and validated its basic functionality. The prototype confirms that it is possible to integrate traditional program analysis metrics with LLM-based evaluations, while also highlighting several challenges that need to be solved before such a tool can be used in practice.

The system integrates a backend with a REST API, a frontend interface, and a configuration mechanism based on declarative YAML files. The prototype demonstrates that benchmarks can be executed with different combinations of task sources and execution settings, producing results that include both traditional metrics, such as unit tests and static analysis, and LLM-based evaluations.

The experiments with the prototype illustrated the complete pipeline: uploading task sources and configurations, launching benchmarks, monitoring their progress, and inspecting results. Even with a small number of tasks and languages, the system proved that such a workflow is feasible and can support different benchmarking scenarios.

The current implementation remains a prototype. It lacks isolated execution environments, which makes performance-related metrics sensitive to the conditions of the host machine. Test execution is supported only for a subset of programming languages, and collaborative use on a shared server would require authentication and file locking. Datasets also need to be expanded and standardized before results could be reliably applied in research.

Future work should address these limitations by improving execution isolation, extending language support, and refining the user interface. Security and collaboration features will be necessary for real-world deployment. Finally, before the approach can be adopted more widely, it will be important to collect feedback from the research community and adjust the design accordingly.

Referencias

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).

19. *Models Benchmarks | Mistral AI*, https://docs.mistral.ai/getting-started/models/benchmark/, [Accessed 27-08-2025].

Appendix A
Sequence Diagram

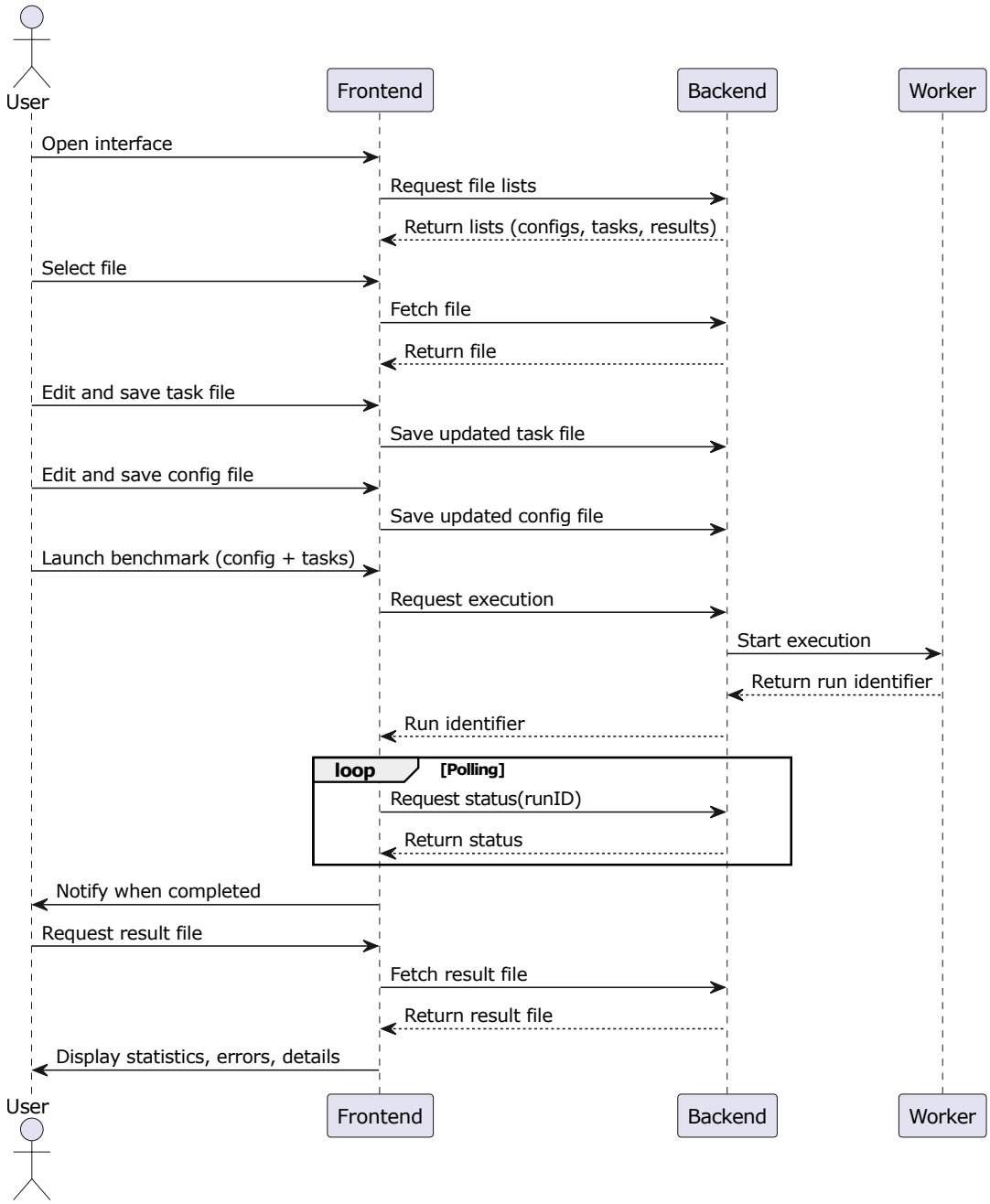

Figura 1: Sequence Diagram.

Appendix B

User Interface Screenshots

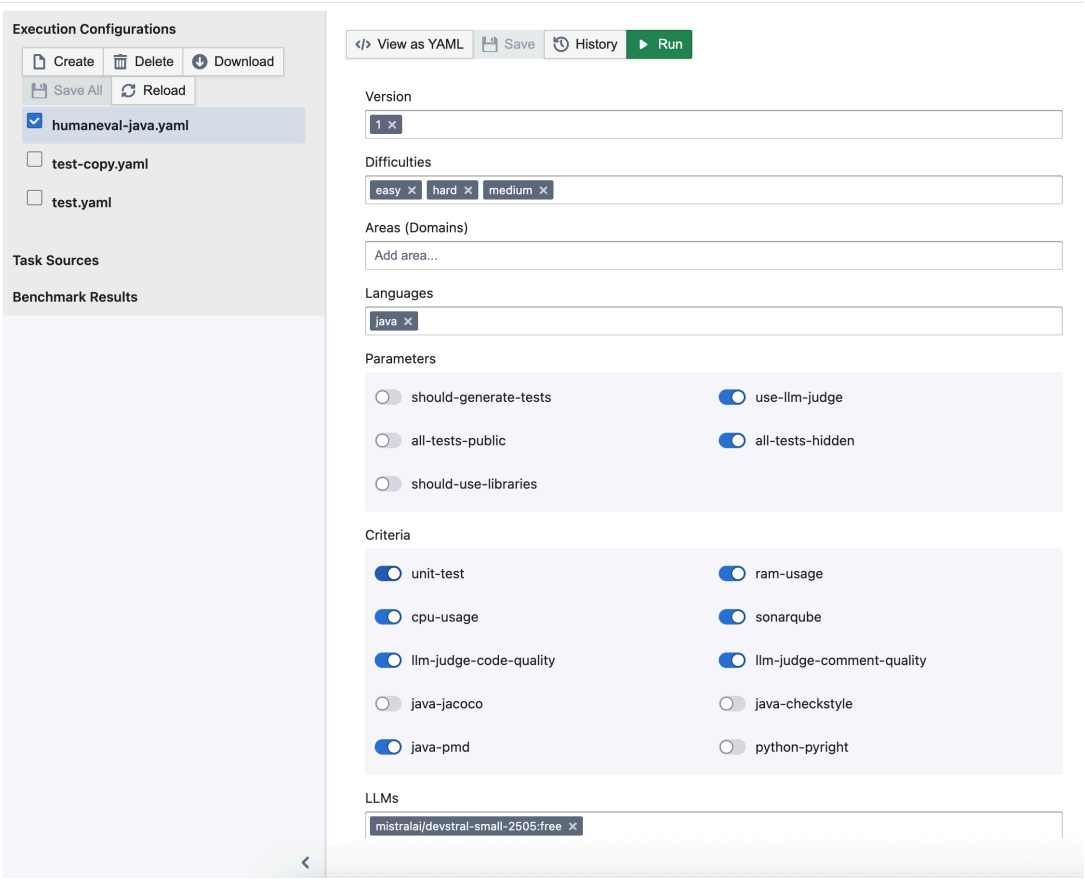

Figura 2: Execution Configuration visual editor.

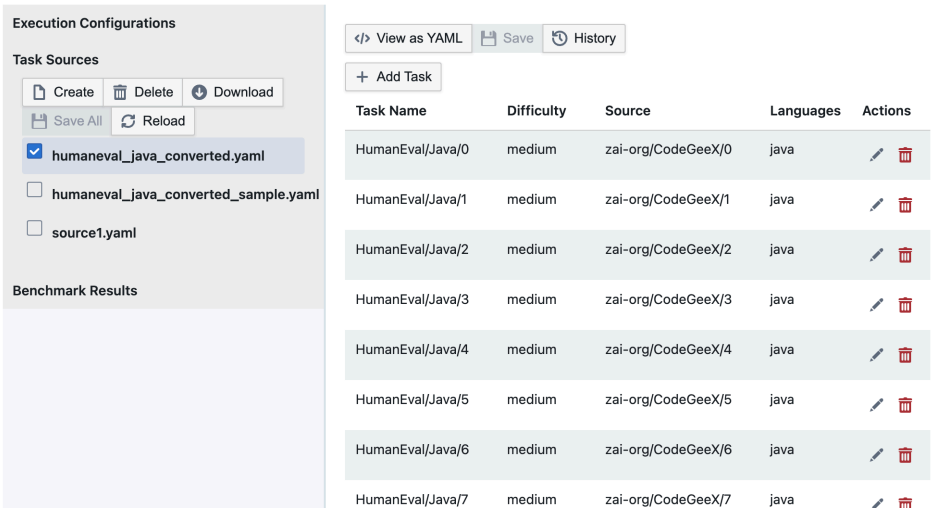

Figura 3: Task Source visual editor. List of tasks in a Task Source.

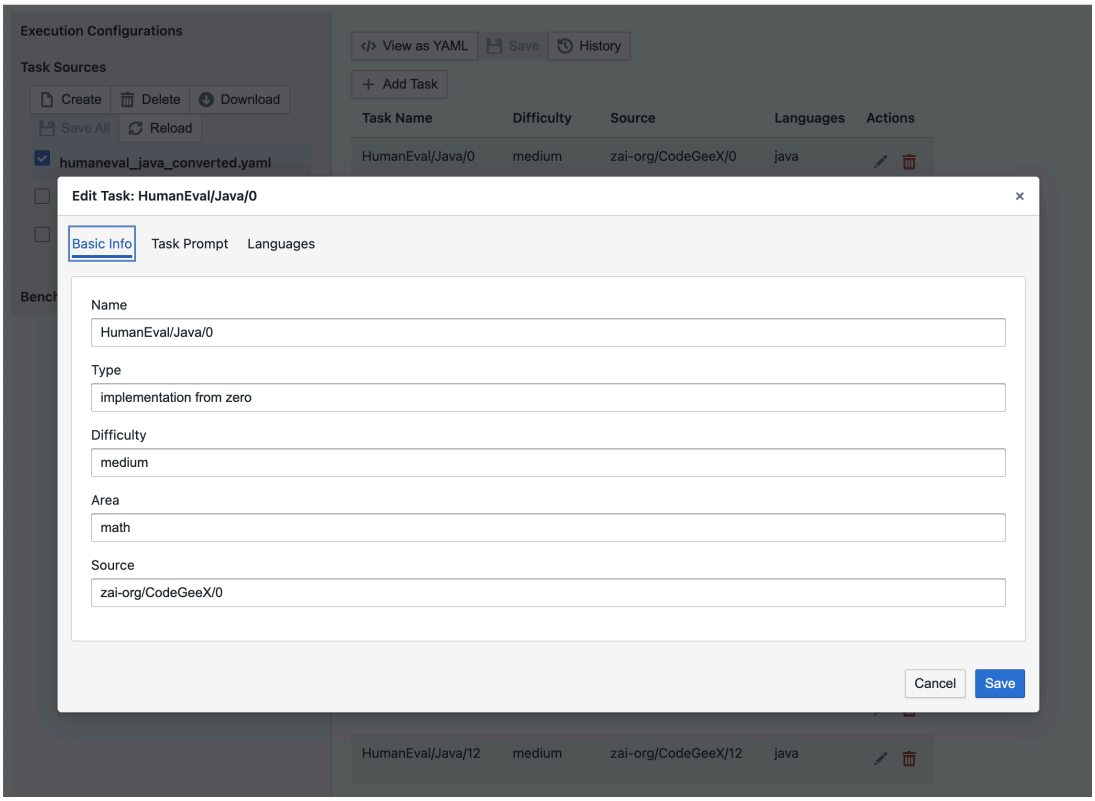

Figura 4: Task Source visual editor. Basic information of a task.

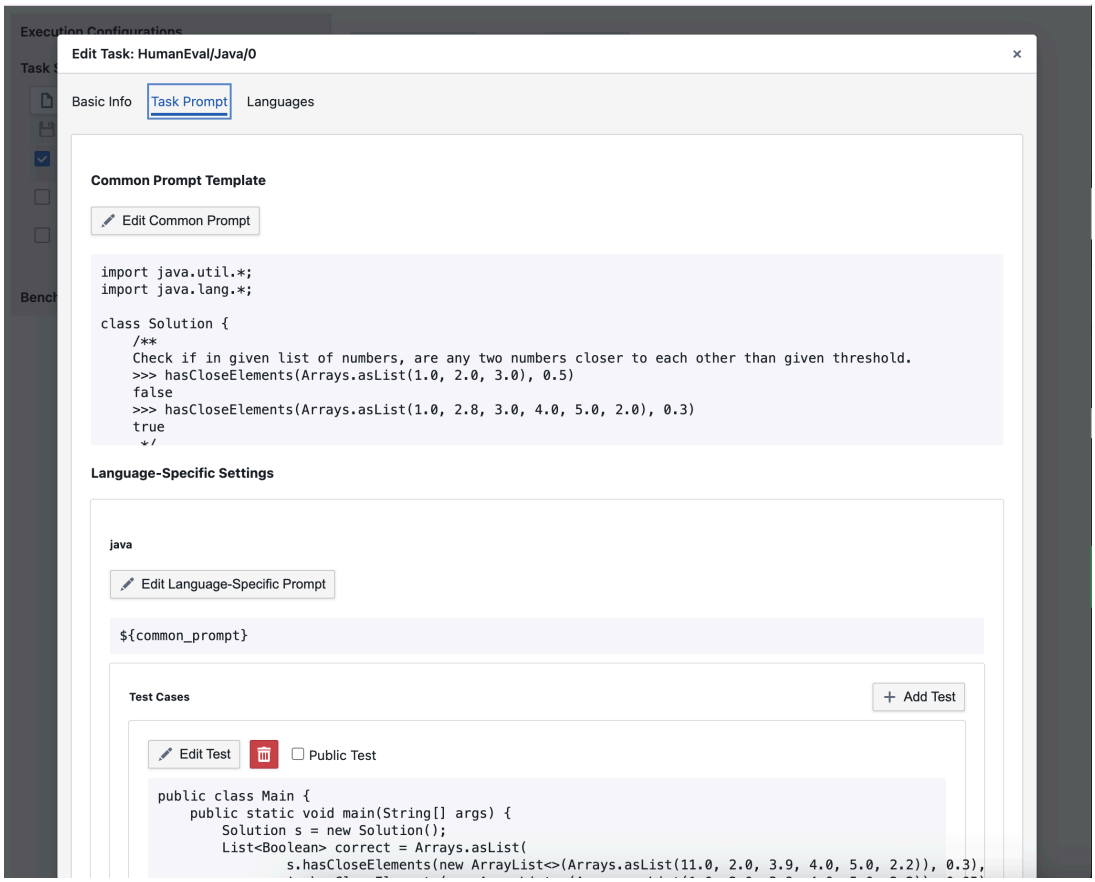

Figura 5: Task Source visual editor. Prompts and tests of a task.

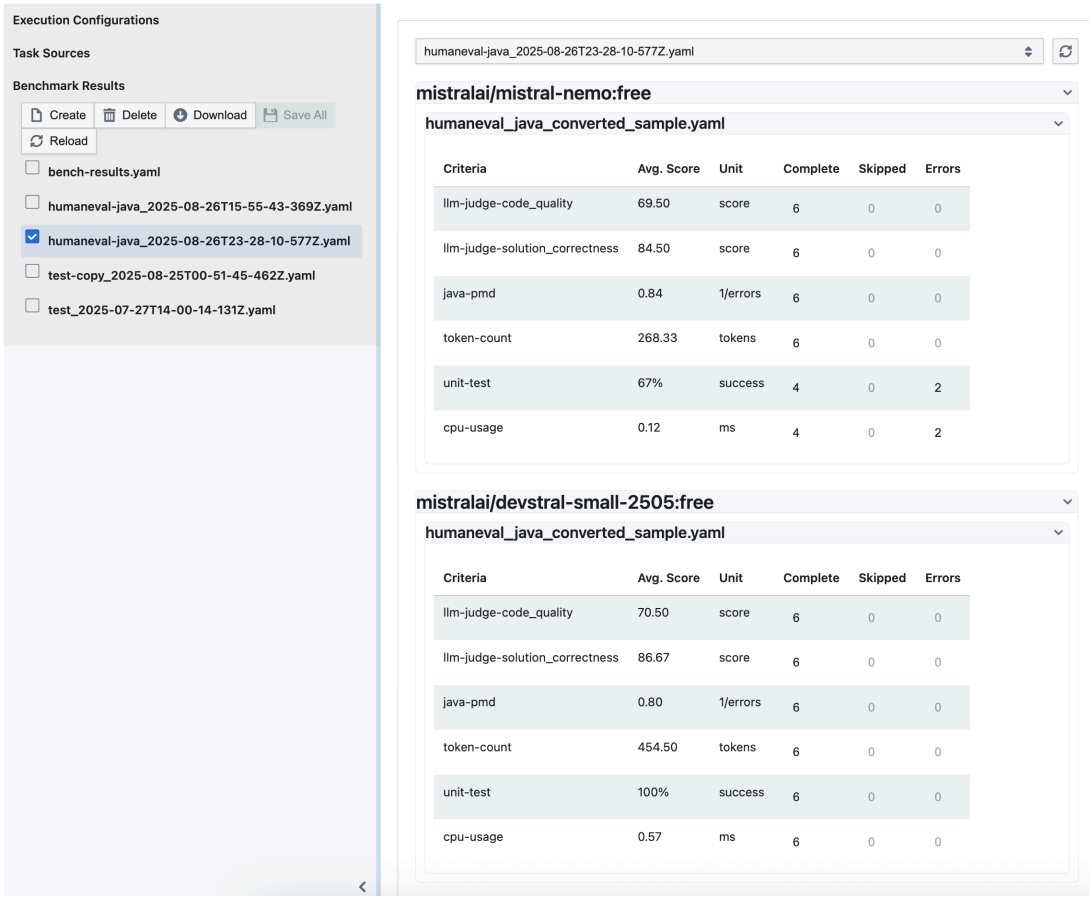

Figura 6: Benchmark Results browser.

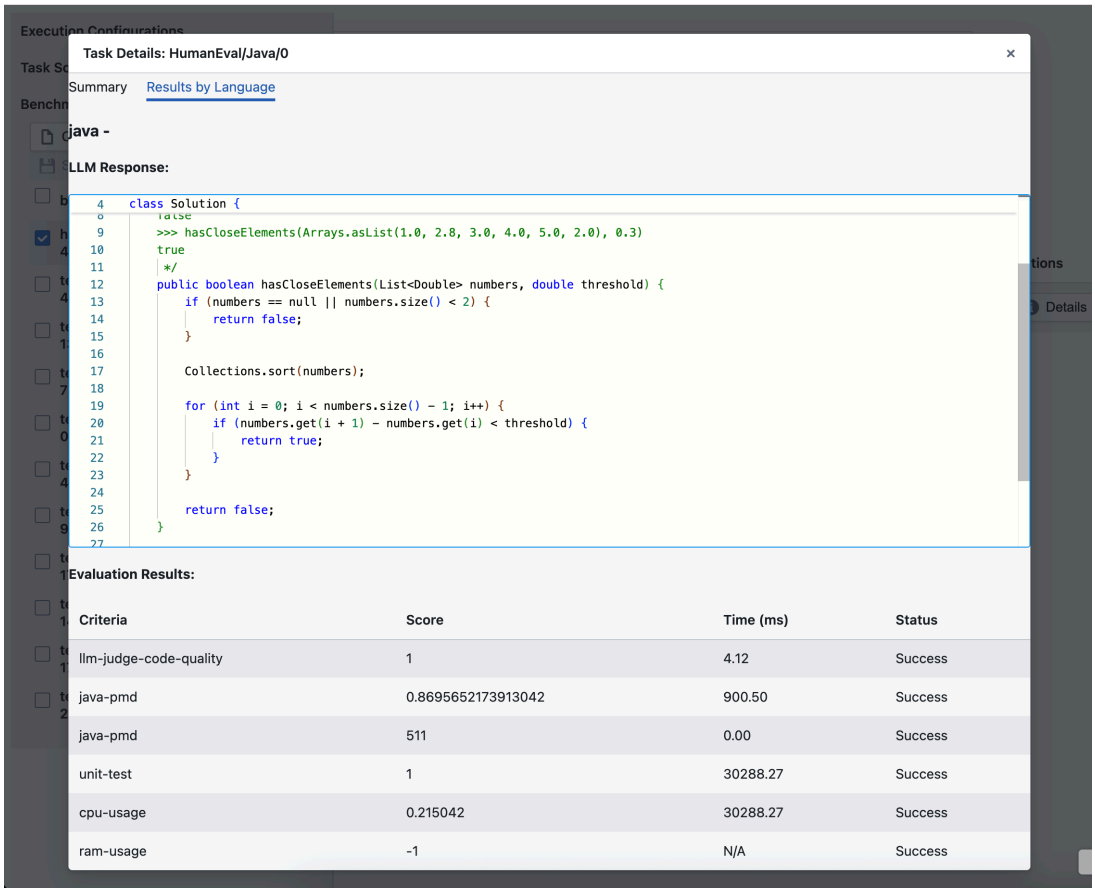

Figura 7: Benchmark Results browser.

Appendix C
Configuration and Result YAML Files

Example of Configuration YAML file.

File: exec-config-example-1.yaml

```yaml
version: 1.0
difficulties:
  - easy
  - medium
  - hard
areas:
languages:
  - java
parameters:
  - name: should-generate-tests # if the LLM should generate tests for the code
    enabled: false
  - name: use-llm-judge         # if we want to check the results with an LLM-judge
    enabled: true
  - name: all-tests-public      # to include tests into a prompt as a reference
    enabled: false
  - name: all-tests-hidden      # if we don't want to give any tests as a reference
    enabled: true
criteria: # list of criteria to evaluate the solutions
  - name: unit-test         # only for languages that we can run in a sandbox
    enabled: true
  - name: ram-usage         # only if unit_test criteria is enabled
```

```yaml
    enabled: true
  - name: cpu-usage          # only if unit_test criteria is enabled
    enabled: true
  - name: sonarqube          # for all languages
    enabled: true
  - name: llm-judge-code-quality    # for all languages
    enabled: true
  - name: llm-judge-comment-quality # for all languages
    enabled: true
  - name: java-jacoco        # only for jvm languages
    enabled: true
  - name: java-checkstyle    # only for java
    enabled: true
  - name: java-pmd           # pmd supports more languages
    enabled: true
  - name: python-pyright     # only for python
    enabled: false
llm-judge: mistralai/devstral-small-2505:free
llms:
  - mistralai/devstral-small-2505:free
  - mistralai/mistral-nemo:free

```

Example of Task Source YAML file.

File: mbbp-sanitized-sample-1.yaml

```yaml
version: 1
name: google-research/mbpp-sanitized
tasks:
- name: mbpp-sanitized/2
  type: implementation from zero
  difficulty: easy
```

```yaml
area: math
source: google-research/mbpp-sanitized/Benchmark Questions Verification V2.ipynb
languages:
- python
available_parameters:
- use-llm-judge
- all-tests-public
- all-tests-hidden
available_criteria:
- unit-test
- ram-usage
- cpu-usage
- sonarqube
- llm-judge-code-quality
- llm-judge-comment-quality
- python-pmd
- python-pyright
task:
  common_prompt: Write a function to find the shared elements from the given
    two lists.
  languages_specific:
    python:
      description: ${common_prompt}
      hidden_tests:
      - code: assert set(similar_elements((3, 4, 5, 6),(5, 7, 4, 10))) ==
          set((4, 5))
      - code: assert set(similar_elements((1, 2, 3, 4),(5, 4, 3, 7))) ==
          set((3, 4))
      - code: assert set(similar_elements((11, 12, 14, 13),(17, 15, 14, 13)))
          == set((13, 14))
golden_solution:
  python: |-
    def similar_elements(test_tup1, test_tup2):
```

```
        res = tuple(set(test_tup1) & set(test_tup2))
        return (res)
  llm_judge_prompt: |-
    You are an experienced interviewer assessing the candidate's solution.
    Here is the task that was given to the candidate:
    ```
    ${prompt}
    ```
    Based on the given task, the candidate wrote the following solution:
    ```
    ${solution_code}
    ```

    Based on the provided task and candidate's solution,
    respond with a YAML that contains numeric evaluations of the
    following concepts on a scale from 0 to 10:
    ```
    solution_correctness: int
    code_quality: int
    style_quality: int
    ```
```

Example of Benchmark Run Status YAML file.

File: mbbp-sanitized-sample-1.yaml

```
---
execConfigFile: "exec_configs/humaneval-java.yaml"
taskSourceStatuses:
  humaneval_java_converted_sample.yaml:
    total: 6
    completed: 4
    filteredOut: 0
```

```
    error: 2
resultFilename: "humaneval-java_2025-08-26T14-37-52-827Z.yaml"

```

TO-DO: Example of Benchmark Result YAML file.

UNIVERSIDAD DE MÁLAGA | uma.es

E.T.S de Ingeniería Informática
Bulevar Louis Pasteur, 35
Campus de Teatinos
29071 Málaga

E.T.S. DE INGENIERÍA INFORMÁTICA