# Evaluating Test-Driven Code Generation: A Replication Study

Giovanni Rosa*, Jesus M. Gonzalez-Barahona*

*SoftDev group, Universidad Rey Juan Carlos, Spain*

**Abstract**

The software engineering community is exploring ways of integrating Large Language Models (LLMs) into software developing processes. One of such explorations is the use of techniques based on Test-Driven Development (TDD), which presents significant challenges due to model variability, evolving APIs, and computational resource demands. In this case, the interest is usually not in showing how well a model behaves, but how a technique may improve the results of the LLM in certain scenarios.

This paper presents a replication study of one of those explorations: *AlphaCodium*, an approach implementing a TDD-based workflow for producing code with LLMs, evaluated by solving competitive programming problems from the *CodeContests* dataset. The primary goal of our study is not the replication itself, but to understand how to empirically evaluate LLM-based code generation approaches, and the challenges involved in replicating such experiments. However, our replication also aims to add more evidence about the effectiveness of the approach.

We extended the original open-source *AlphaCodium* implementation to support some open-weights models, replicating the evaluation on the same dataset. In the process, we collected some supplementary inference metrics such as latency and token usage.

By discussing the outcome and challenges encountered, we offer a set of actionable takeaways that can help future replication studies of multi-step approaches for LLM-based code generation.

**Keywords**

code generation, TDD, LLM, replication study, performance evaluation, software engineering

## 1. Introduction

Large Language Models (LLMs) have shown remarkable capabilities in various natural language processing tasks, and have been widely adopted in the software engineering domain [1]. Starting from code assistants to help developers write, debug, and refactor code [2], the development of code-specific LLMs has been a major focus in recent years [3, 4]. LLMs have been employed to tackle competitive programming problems, where the models are measured by their capability of generating code following specific functional requirements and passing a set of test cases. An important mention is *AlphaCode* [5], developed by Google DeepMind, which demonstrated effective in that purpose.

The software engineering community is exploring ways of integrating LLMs into software development processes, with different purposes: to produce better code, to collaborate in software maintenance tasks such as bug fixing, or to improve how they take into account contextual information. One of these explorations deals with the use of Test Driven Development (TDD) [6, 7] to improve code generation. The general idea in this realm is to make LLMs iteratively generate and refine code based on a provided set of test cases, which are expected to guide the model towards producing correct and robust implementations, given a specification. These approaches are usually evaluated by trying the approach to solve coding problems for which tests can be found.

However, evaluating and replicating any kind of experiment involving LLMs can be challenging due to the rapid evolution of the models, the inherent variability in LLM outputs, which can lead to inconsistent results across different runs, and the fair evaluation of the approaches relying on LLMs, which can be difficult due to the high costs associated with using proprietary models and the complexity

*Benevol'25: 24th Belgium-Netherlands Software Evolution Workshop 17–18 November 2025, Enschede, The Netherlands*

*Corresponding authors.

giovanni.rosa@urjc.es (G. Rosa); jesus.gonzalez.barahona@urjc.es (J. M. Gonzalez-Barahona)
https://giovannirosa.com (G. Rosa); https://gsyc.urjc.es/jgb/ (J. M. Gonzalez-Barahona)
0000-0002-5241-1608 (G. Rosa); 0000-0001-9682-460X (J. M. Gonzalez-Barahona)

of setting up open-source models. This is even more challenging when the aim is not to evaluate the model itself, but how a given technique (*i.e.,* derivative of TDD) may improve the performance of LLMs.

In this paper, we present a replication study designed to understand how to empirically evaluate LLM-based code generation approaches, and the challenges involved in replicating such experiments. To this aim, we replicate the study of Ridnik *et al.* [8] which introduced *AlphaCodium*, a TDD-based approach for enhancing the effectiveness of LLMs in code generation. The approach involves a robust pre-processing phase, including problem reflection and test case augmentation, which extends similar approaches in the literature [6, 7], and most importantly, the source code is publicly available on GitHub [9], facilitating the replication process. We evaluate the approach on the *CodeContests* dataset [5], composed of coding problems, which is the same used by the original study, demonstrating significant improvements over a simple direct code generation method.

Note that in this kind of evaluation, the focus is not on the capabilities of the LLMs themselves, but rather how certain techniques, usually related to software engineering practices, can be used to improve results in certain tasks. Given this situation, our goal is to understand the challenges related to the replication and evaluation of the complex workflow implemented in *AlphaCodium*, rather than the performance of the LLMs themselves.

The authors of the original study provided an open-source implementation of *AlphaCodium*, which we used as a starting point for our study. We leveraged this implementation by extending and modifying it to support additional LLMs, such as open-weights models, and to extend the evaluation parameters. Using our tool, we could replicate the original study with other LLMs, and report on some other performance-related parameters not considered in it, such as latency and token usage. We report the results of our replication study and provide some insights into the challenges and common errors that arose during the replication process, along with a set of takeaways for future replication studies.

The rest of the paper is organized as follows. Section 2 presents the related work. Section 3 describes the experimental protocol of the replication study. We present the results in Section 4, while in Section 5 we discuss what we learned during the replication process. Finally, in Section 6 we report the threats of validity for the study and Section 7 concludes the paper.

## 2. Related Work

Large Language Models (LLMs) have been used extensively in code-related tasks, such as code completion and code generation. A seminal study in this field introduced the *Codex* models and evaluated them on competitive programming problems [10]. They introduced the `pass@k` metric, which is now widely used to evaluate code generation models. Several studies have conducted similar experiments to produce models and approaches specialized in solving coding problems. For example, Google Deepmind introduced *AlphaCode* [5], improved in *AlphaCode 2* [11], which is able to reach a competitive level in programming competitions.

Exploring ways of combining LLMs with software engineering practices aimed at improving code quality, several studies investigated the usage of LLMs to solve coding problems with the aid of Test Driven Development (TDD) practices. Piya *et al.* [6, 12] investigated the impact of TDD practices on the performance of LLMs in solving *LeetCode* problems, showing that TDD can significantly improve the performance of LLMs in solving coding problems. Fakhoury *et al.* [13] introduced TICODER, a workflow for test-driven, interactive code generation. TICODER automatically generates tests and code candidates from user requirements, allowing iterative refinement through user feedback. Their evaluation, using both user studies and benchmark datasets such as *MBPP* [14] and *HumanEval* [10], demonstrates that test cases can effectively enhance code quality. Similarly, Mathews *et al.* [7] proposed TGen, which applies TDD principles to LLM-based code generation. Starting from input test cases, TGen generates code and iteratively remediates it based on failed test outputs until all tests pass. Experiments on MBPP and HumanEval show that TDD-based workflows significantly improve the correctness and robustness of LLM-generated code.

*AlphaCodium* [8] proposes a similar technique, introducing a robust pre-processing phase including a problem reflection step and a test cases augmentation procedure. The authors evaluated their approach on the *CodeContests* dataset, showing that it significantly improves the performance of LLMs in solving competitive programming problems. We selected *AlphaCodium* as the replication target for our study, leveraging the open implementation provided by its authors [9]. We describe it more in detail in Section 3.1.

# 3. Replication Study Design

We report a replication study of the paper of Ridnik *et al.* [8], with the *goal* of understanding how to evaluate approaches built upon LLMs and the challenges faced when dealing with LLM-based experiments designed not to evaluate the models themselves, but relatively complex pipelines using them.

Our replication study is performed to answer the following *research question*:

*RQ: To what extent are we able to replicate the empirical evaluation of the AlphaCodium approach?*

We aim to replicate the results of the original paper as closely as possible, focusing on how to evaluate approaches built upon LLMs and the related challenges when dealing with LLM-based experiments.

## 3.1. The AlphaCodium Approach

The approach presented in the original study we replicate, called *AlphaCodium*, is based on the idea that generating code in multiple steps, with intermediate verification and refinement, can lead to better results than a single-step generation. The overall approach is shown in Fig. 1, and can be summarized as follows: Taking as input a coding problem description (specification) and the test cases to verify the solution, *AlphaCodium* (i) generates a possible resolution procedure (*i.e.,* list of steps to solve the problem), (ii) generates a set of possible solutions and selects the best one, (iii) generates additional test cases, and (iv) executes a code generation loop running the test cases and fixing the function implementation. For additional details on the approach, we refer the reader to the original paper [8].

The original study conducted an empirical evaluation of the approach using different LLMs, including open-weights models (*DeepSeek*) and closed ones, like OpenAI's *GPT-3.5* and *GPT-4*. The baseline is composed of a single-step code generation using the same models, using the Zero-shot prompting technique [15], which is basically the simplest way of prompting an LLM to generate code from a specification. Therefore, this is a good example of an evaluation not aimed at evaluating the models themselves, but how they can be improved by using some specific technique. Moreover, the original study compared the results with *AlphaCode* [5], a state-of-the-art approach for code generation based on

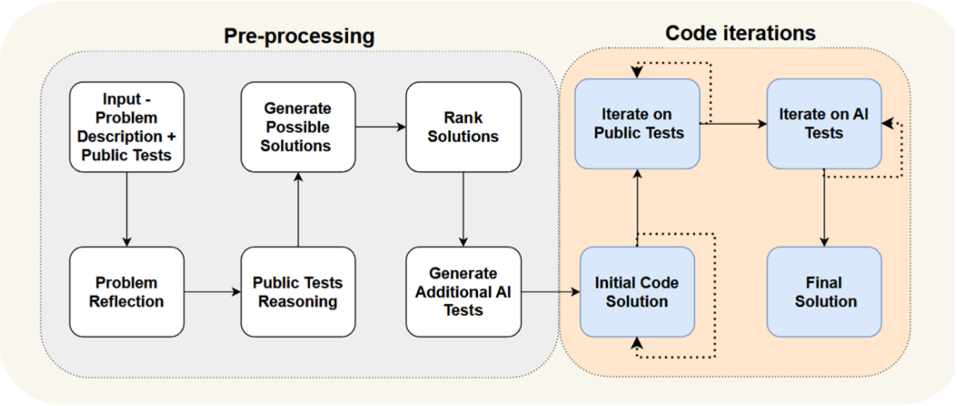

**Figure 1:** The *AlphaCodium* approach proposed by [8].

LLMs. We selected *AlphaCodium* because we were interested in existing approaches for code generation based on an iterative workflow, combined with TDD, and also because the source code of the tool is publicly available [9].

## 3.2. Study Context

The *context* of the original study consists of the *CodeContests* dataset [5], a set of coding problems extracted from several programming platforms, such as *Codeforces*[1] and *Codechef*[2]. Each instance of the dataset contains the data to train and evaluate code generation models, including: (i) the problem description, (ii) the function signature, (iii) the public and private test cases, and the (iv) solution code. The full list of fields is summarized on the dataset card[3].

We started with the exact same dataset used in the original paper, publicly available on HuggingFace [4]. While the dataset is composed of training, validation, and test splits, we only use the *validation* and *test* splits, as did the *AlphaCodium* paper, for a total of 117 and 165 instances, respectively. They discarded a total of 12 instances from the test set, resulting in a total of 156 instances, and 9 instances from the validation set, resulting in a total of 105 instances. We investigated the reason behind this, and we found in the code that if less than 20% of the solutions provided in the dataset are correct, the problem is marked as invalid, and then skipped by the tool. Approximately, the knowledge cut-off date for the problems contained in the dataset is around 2021.

## 3.3. Experimental Procedure

In this section, we describe the experimental procedure we followed to replicate the results of the *AlphaCodium* approach, describing the differences with the original experiment.

### 3.3.1. Tool implementation

The *AlphaCodium* tool is implemented in Python and is publicly available on GitHub [9], which served as our starting point for the code used for replicating the experiment. One of the main differences in our replication is the supported LLMs. While the original paper uses OpenAI and DeepSeek models, we extended the tool to support open-weights models deployed via *vLLM*[5] and models accessible through *OpenRouter APIs*[6]. We chose vLLM because it allows deploying HuggingFace models with production-ready performance and improved reliability, compared to alternatives like *Ollama*[7]. OpenRouter was selected for the wide availability of model providers, including OpenAI and Anthropic, and a unified pay-per-use API.

Moreover, we made several improvements and fixes to enhance the tool's reliability and robustness. These include better output parsing (handling corner cases), improved exception handling, extended logging (*e.g.,* token usage and latency), and the support for additional configuration parameters. An example is the option to run a single problem instance or a subset of instances, which facilitates debugging and targeted re-execution in case of failures or interruptions. Also, the problem iterations are configured to stop once a valid solution was found. We modified this behavior to always perform the maximum number of iterations, to obtain a complete set of results of the approach effectiveness.

We implemented the storage of intermediate results for each iteration. This allows us to resume the execution from the last completed iteration after interruptions. Also, we set an empty result entry for the cases where the model fails to generate a solution or the generated solution does not pass all the test cases within the maximum number of iterations. This because by default the tool will

[1] https://codeforces.com/

[2] https://www.codechef.com/

[3] https://huggingface.co/datasets/deepmind/code_contests

[4] https://huggingface.co/datasets/talrid/CodeContests_valid_and_test_AlphaCodium

[5] https://docs.vllm.ai

[6] https://openrouter.ai/

[7] https://ollama.com

re-execute these iterations. These changes allowed us to parallelize the execution, significantly speeding up the experiments since each problem instance is independent. We can conclude that we made mostly engineering improvements while preserving as much as possible the original behavior of the tool. The modified version of the tool used in this study is publicly released as part of the replication package [16].

### 3.3.2. Experimental parameters

The models used for our replication are the following:

- *openai/gpt-3.5-turbo* via OpenRouter APIs[8]: Since *gpt-3.5* is not available on OpenRouter, we opted for the turbo version. The knowledge cut-off is reported as September 2021.
- *deepseek-ai/deepseek-coder-33b-instruct*[9] [4] via vLLM: The original paper only reports *deepseek-coder-33b* as the model. We opted for the *instruct* version, better suited for instruction-following tasks. The knowledge cut-off is not explicitly reported, but the model was released in November 2023, so we can assume the knowledge cut-off is a few months prior to that date.
- *codellama/CodeLlama-34b-Instruct-hf*[10] [3] via vLLM: Created from Meta's *Llama 2* family of models, it is an open-weights model widely used for code generation tasks [17]. The model has been trained between January and July 2023.

Given the cut-off date of the evaluation dataset, we can assess on the likelihood of contamination: that the models had access to the dataset, including solutions, as a part of their training set. The versions of Deepseek and CodeLlama which we use may have contamination, but GPT-3.5-Turbo should not have it according to the reported knowledge cut-off date. In any case, we do not know how much the contamination may affect the results.

We used the same execution parameters provided in the open source implementation of the tool (*i.e.,* TOML configuration file[11]). We made some modifications, mostly by adding new parameters, while trying to keep them as close as possible to the original ones. Specifically, we kept the same model temperature, which depends on the individual prompt and varies between 0.2 and 0.3. We set the parameter for maximum number of iterations to 5. An iteration corresponds to a single attempt to solve the problem and thus, we need a total of five attempts for each problem to compute the `pass@5` metric. We kept the original parameters for the number of possible solutions generated during the pre-processing step (*i.e.,* 3), the total attempts to generate the initial code solution for the TDD loop (*i.e.,* 8). Also, we kept the total number of feedback iterations in the TDD loop set to 3, meaning that the model has three chances to fix the code based on the failed test cases. Please refer to the original configuration file for the complete list of available parameters.

We changed the execution timeout: While the original tool used a timeout of 90 seconds for inference calls, we increased this to 600 seconds to help with the higher latency of open-weights models. This value is the default for the OpenAI Python SDK. We used the `use_baseline` flag as a parameter to switch between the *AlphaCodium* approach and the single-step code generation baseline, resulting in two different resolution procedures for the experiment.

### 3.3.3. Evaluation metrics

The original paper uses the `pass@k` metric [10], the reference measure for code generation tasks. The metric is computed as follows: Given a set of $n$ generated solutions samples for a problem, for up to $k$ attempts, the `pass@k` metric measures the probability that at least one of the $n$ samples passes all the test cases. As stated in Section 5.3 of the original paper, *"[...] we perform 15-20 LLM calls per solution, so a pass@5 submission involves 100 LLM calls"*. Therefore, we assume that they generated 5 solutions for each problem. Then, we computed the `pass@5` metric[12] using 5 samples per problem

[8] https://openrouter.ai/models/gpt-3.5-turbo

[9] https://huggingface.co/deepseek-ai/deepseek-coder-33b-instruct

[10] https://huggingface.co/codellama/Codellama-34b-instruct

[11] https://github.com/Codium-ai/AlphaCodium/blob/main/alpha_codium/settings/configuration.toml

[12] By definition, computing `pass@k` requires at least $n \geq k$ samples per task.

**Figure 2:** Example of the command used to deploy *DeepSeek-Coder* via vLLM.

```
vllm serve deepseek-ai/deepseek-coder-33b-instruct \
--host 0.0.0.0 --port 8000 \
--gpu-memory-utilization 0.95 \
--max-model-len 53700 \
--trust-remote-code \
--tensor-parallel-size 2
```

($n = 5$). Each instance of the dataset, along with public and private test cases, contains an additional set of *generated* test cases, created based on the existing public and private ones. We computed the metric using both sets, as in the original paper, and also report the results using only the private test cases to provide an alternative evaluation perspective that is closer to a real-world scenario for coding challenges. Additionally, we report the collected statistics for each inference call as latency (*i.e.,* time taken to get the response from the model), token usage (*i.e.,* total and completion tokens). We report the average values computed for a single iteration, along with the total number of calls to the model.

### 3.3.4. Execution environment

We deployed the open-weights models on a multi-GPU shared server with a total of 4 A100 GPUs (80 GB), an AMD EPYC 7313 16-Core and 256 GB of RAM. This allowed us to execute full precision models, without the need for quantization, which could influence the performance of the models. For the experiments using the *OpenRouter* APIs, we used a standard laptop machine with an Intel i7-12700H CPU and 64 GB of RAM. To ensure a clean and isolated execution environment, we ran the tool inside a dedicated Docker container. This enhances reproducibility of the execution environment and minimizes any potential security risks associated with unexpected behavior of the generated code. The tool also incorporates a sandboxing mechanisms limiting memory and potentially dangerous commands (*e.g.,* `rm`). The vLLM instance has been deployed inside a separate Docker container on the same server, using two of the four GPUs with tensor parallelism. We then configured the *AlphaCodium* tool to interact with it via REST APIs. Although the GPUs have sufficient memory to load the models, for *DeepSeek-Coder* we needed to limit the maximum context length of the model. This did not affect our experiments, since the prompts are much smaller than this limit. In Fig. 2 we provide an example of the command used to deploy the model.

## 4. Replication Study Results

Table 1 presents the `pass@k` scores from our replication study, using $k = 5$ and $n = 5$ samples per problem instance. We compare these results with those reported in the original paper [8]. The outcomes are comparable and generally consistent with the original scores.

For the baseline approach, our pass rates are slightly higher, while for *AlphaCodium* the results are similar. This difference may be attributed to the high variance in results due to the limited number of samples per problem instance, particularly for the baseline, which relies on a single call to the LLM.

*CodeLlama* remains the worst performing model, but its results are still consistent with the original study, where the *AlphaCodium* approach outperforms direct solving. We do not have a direct comparison for *GPT-4*, the best-performing model in the original paper. However, given the release dates and knowledge cut-offs, it is reasonable to assume that it is more capable than the other models, likely benefiting from newer data, training procedures, and having portions of its training data overlapping with the *CodeContests* dataset.

Table 2 shows the results obtained by measuring the `pass@5` score only on the private test set. We can conclude that having fewer test cases makes the overall score higher, assuming that it is easier to

**Table 1**
Summary of the results for the pass@5 score evaluated on *validation* and *test* splits. The symbol † indicates those from the original paper [8].

| **Model** | **Validation Set** (**pass@5**) | | **Test Set** (**pass@5**) | |
|---|---|---|---|---|
| | *Zero-Shot* | *AlphaCodium* | *Zero-Shot* | *AlphaCodium* |
| DeepSeek-33B† | 7% | 20% | 12% | 24% |
| GPT-3.5† | 15% | 25% | 8% | 17% |
| GPT-4† | **19%** | **44%** | 12% | **29%** |
| deepseek-ai/deepseek-coder-33b-instruct | 10% | 18% | **14%** | 21% |
| openai/gpt-3.5-turbo | 11% | 22% | 10% | 21% |
| codellama/CodeLlama-34b-Instruct-hf | 2% | 5% | 4% | 8% |

**Table 2**
Summary of the results for the pass@5 score evaluated only on the problems' private test set.

| **Model** | **Validation Set** (**pass@5**) | | **Test Set** (**pass@5**) | |
|---|---|---|---|---|
| | *Zero-Shot* | *AlphaCodium* | *Zero-Shot* | *AlphaCodium* |
| deepseek-ai/deepseek-coder-33b-instruct | **20%** | **27%** | **22%** | **23%** |
| openai/gpt-3.5-turbo | 19% | **27%** | 13% | 19% |
| codellama/CodeLlama-34b-Instruct-hf | 6% | 9% | 10% | 5% |

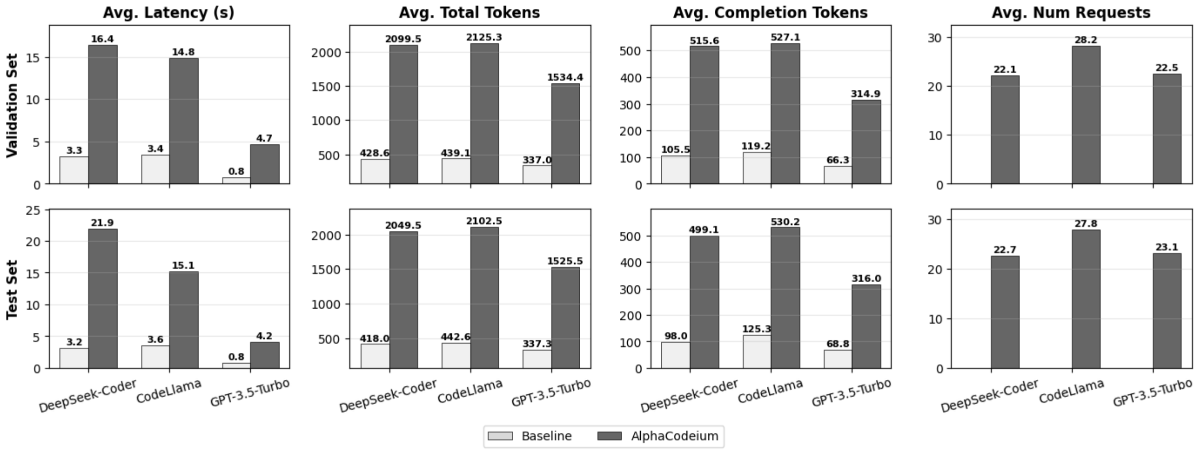

**Figure 3:** Summary of the inference statistics collected for the experiments running *AlphaCodium*.

produce a solution that passes all the tests. However, the relative performance of the models remains the same, with *AlphaCodium* achieving better results than the baseline approach.

Fig. 3 shows the statistics collected during the model inference. The values represent the average value for a complete execution of a single instance (single iteration). The latency of *GPT-3.5-turbo* is lower than the other models, because of *OpenRouter*. A fairer comparison is between *DeepSeek-Coder-33B* and *CodeLlama-34B*, both deployed on the same hardware, where we can see that the former is much slower than the latter, probably due to the complexity of the model. In terms of token usage, *GPT-3.5-turbo* is more efficient despite performing similarly to *DeepSeek-Coder-33B* in terms of number of requests and achieved pass rate. Additionally, the total token usage for *AlphaCodium* is significantly higher than for the baseline approach, as expected, since it makes multiple requests to the LLM for each problem instance. For a fairer comparison, it would be better to consider the overall cost of each approach, or to allocate the same token budget to both methods and compare their results accordingly.

# 5. Challenges and Lessons Learned

Replicating and evaluating the experiments proved to be challenging, specifically when dealing with large language models (LLMs) that evolve rapidly. Below, we outline the issues we encountered, our mitigations, and lessons learned.

**Flaky formatting errors.** The non-determinism of the execution of the models led to flaky errors related to formatting in the generated YAML output. Actually, the prompt requests to generate a YAML-formatted output with the required fields (*e.g.,* test cases, resolution procedure, etc.). However, some models struggled to adhere strictly to this format, leading to parsing errors when the output is processed by a YAML parser. We chose to not force the parsing of that output, considering those cases as invalid responses from the model.

**Token generation loop.** In our replication, we worked with LLMs that are older and less capable than current ones. A recurrent issue, especially with `deepseek-coder-33b-instruct`, is that the model sometimes enters a token generation loop, repeatedly generating the same or random tokens without making progress toward a valid response. In those cases, the inference exceeds the timeout and the generation stops. Identifying these cases is not trivial. One strategy could be to use streaming output and monitor the generated tokens, although that would make the code more complex. Fortunately, we observed that the average latency for normal responses is much lower than for these generation loops (see Fig. 3). Thus, a simple and effective mitigation is to use an anomaly detection heuristic based on a threshold on average latency to identify these cases, and rerun the inference.

**Unable to execute the generated code.** Another issue is that, in several cases, the generated code cannot be executed either due to syntax errors or missing dependencies. Even if the *AlphaCodium* pipeline includes an iteration phase with a code-fixing loop, we found that some models struggle to generate syntactically correct code, leading to execution failures. Examples are invalid code indentation, error while parsing the input of test cases, recalls to missing functions or files, accessing invalid list ranges or invalid type comparison (*e.g.,* string with integer). Other frequent errors come from the presence of explanations inside the code block that the model provided, leading to execution errors. In a few cases, the execution silently fails with no output or errors. We concluded that the code is not executable, and we count these cases as failures.

## 5.1. Takeaways

In the following, we summarize some lessons learned and challenges faced during the replication study.

🔍 **Reliable Evaluation Tools.** We aimed to replicate the evaluation pipeline of the target paper as closely as possible, extending the provided codebase to support different models and infrastructures. Our goal was to focus on having a very reliable tool, allowing to re-run individual problem instances without restarting the entire process. We found it fundamental to have a reliable and well-tested evaluation tool, or at least, to have a starting basis to build upon. We encourage the research community to share their evaluation tools and pipelines to facilitate replication and comparison of results. A strategy could be to reuse and extend existing benchmarks such as LiveCodeBench [18], or consider adopting libraries like HuggingFace's *evaluate*[13].

🔍 **Fast and reliable inference infrastructure.** A significant amount of effort was devoted to setting up an inference infrastructure capable of running large models efficiently. We initially opted for *Ollama*, a popular and easy-to-use model serving tool. However, we found that its model compression format is not well suited for multi-GPU settings and complicates replication. We switched to *vLLM* to avoid provider-specific artifacts and ensure that our results could be replicated by others by using standard tools and the open-weights models offered by Hugging Face.

We deployed a single *vLLM* instance with tensor parallelism across our multi-GPU server, leading to a better load balancing and faster inference times compared to a single instance per GPU. Setting up such an environment is non-trivial and requires significant technical expertise. A useful contribution is

[13] https://github.com/huggingface/evaluate

to have a set of guidelines and scripts to facilitate the deployment and management of model inference infrastructures, such as having reference infrastructure-as-code artifacts.

🔍 **Rapid Model Evolution.** The exact versions of the models used in papers reporting empirical experiments were not always obtainable, some are no longer available or became unreasonably expensive (to force users to move to newer versions), and many have different knowledge cut-off dates due to continuous updates to their training data (*e.g.,* OpenAI models). Therefore, we find it important to prioritize open-weight models in evaluations, and specifically when the evaluation is not about the model, but about some pipeline on top of it. It is also important to document as much as possible the model version, knowledge cut-offs, and prompt settings to facilitate future replications. An effort in providing such a set of recommendations has been proposed by Baltes *et al.* [19].

Additionally, the model capabilities are becoming more and more advanced, and therefore some practices become obsolete. For example, using structured output with modern models leads to more reliable results rather than using YAML formatting. We initially implemented structured output parsing to mitigate formatting errors and parsing issues. This is true for modern models, such as *GPT-4o*, but led to inconsistent results with the models used in the experiment. To ensure a fair comparison with the original results, a good strategy for future work would be to choose a set of models having similar capabilities and cut-off date.

🔍 **Metric Limitations.** The standard metric for code generation tasks, `pass@k`, measures the ability to generate code that passes predefined test cases. However, it overlooks factors like the number and complexity of test cases, code quality, efficiency, and overall pipeline cost (*e.g.,* tokens used). This leads to an unfair comparison between simple baselines and more complex approaches like *AlphaCodium.* The same is true when there are differences in terms of capabilities of the model (*e.g., GPT-4* vs. *Codellama*), token usage (*AlphaCodium* uses four times more tokens), and inference time, which is crucial in practical applications.

We suggest considering additional metrics and qualitative analyses in these kinds of evaluations, such as weighting scores by test case count, measuring average time to solution, or assessing code quality and efficiency with static analysis tools. Considering problem complexity and test case coverage is also important, as simple test cases can inflate scores.

## 6. Threats to validity

In this section, we report the threats to the validity of our study.

**Construct Validity**. Even if we used most of the code and parameters from the original study, our modifications could have altered the final results. The obtained results are quantitatively consistent with those in the original paper. However, we cannot completely rule out since the outputs of LLMs can be highly variable even with the same inputs and parameters.

**Internal Validity**.

Although `pass@k` is widely used as a metric for evaluating code generation models [10], the sample selected in our experiment could have led to high variance in the results. However, a solid replication would have required far more computational resources and time. Since our goal was limited to exploring the replicability of the *AlphaCodium* approach, we acknowledged this threat when discussing the results.

Another threat concerns the choice of models. We selected models as close as possible to the originals (*i.e., Deepseek-Coder* and *GPT-3.5-Turbo*), adding *CodeLlama* for comparison. There is also a risk of knowledge overlap between model training data and the *CodeContests* dataset, potentially favoring some models. However, only *GPT-3.5-Turbo* should be unaffected by this, and its results are comparable to *Deepseek-Coder*, suggesting that the model has not been penalized.

**External validity**.

This threat mostly refers to the applicability of the discussed challenges and takeaways. Despite they are general suggestions applicable to LLM-based experiments, there could be cases in which some encountered problems are less prominent. For example, newer models have fewer issues in generating structured outputs.

## 7. Conclusion and Future Work

Test-Driven Development (TDD) has been shown to be an effective approach to enhance the performance of Large Language Models (LLMs) in solving competitive programming problems. In this paper, we presented a replication study of the work by Ridnik *et al.* [8], which introduced *AlphaCodium*, a TDD-based approach to improve LLMs' capabilities in this domain.

Our replication study extended the original implementation to support additional LLMs, including open-source models, and we evaluated their performance on the *CodeContests* dataset [5]. We obtained results that were generally consistent with those reported in the original paper and discussed the challenges and common errors encountered during the replication process.

However, in the process of building the software for the replication, and while running the replication itself, we also learned several details, which we have presented as observations and takeaways.

We plan to further investigate the applicability of different metrics for a broader and fair evaluation of LLM-based code generation approaches, as well as extend the evaluation to additional models and newer datasets. Last but not least, we also plan to release in the future the evaluation tool we are developing to help the community in replicating and benchmarking LLM-based code generation approaches.

## 8. Acknowledgements

The study presented in this paper was funded in part by the Advise project, funded by the Spanish AEI with reference 2024/00416/002.

## 9. Data Availability

The replication package with the modified version of the tool used in this study is publicly released [16].

## References

[1] A. Fan, B. Gokkaya, M. Harman, M. Lyubarskiy, S. Sengupta, S. Yoo, J. M. Zhang, Large language models for software engineering: Survey and open problems, in: 2023 IEEE/ACM International Conference on Software Engineering: Future of Software Engineering (ICSE-FoSE), IEEE, 2023, pp. 31–53.

[2] N. Nguyen, S. Nadi, An empirical evaluation of GitHub Copilot's code suggestions, in: Proceedings of the 19th International Conference on Mining Software Repositories, 2022, pp. 1–5.

[3] B. Roziere, J. Gehring, F. Gloeckle, S. Sootla, I. Gat, X. E. Tan, Y. Adi, J. Liu, R. Sauvestre, T. Remez, et al., Code Llama: Open foundation models for code, arXiv preprint arXiv:2308.12950 (2023).

[4] D. Guo, Q. Zhu, D. Yang, Z. Xie, K. Dong, W. Zhang, G. Chen, X. Bi, Y. Wu, Y. Li, et al., DeepSeek-Coder: When the large language model meets programming–the rise of code intelligence, arXiv preprint arXiv:2401.14196 (2024).

[5] Y. Li, D. Choi, J. Chung, N. Kushman, J. Schrittwieser, R. Leblond, T. Eccles, J. Keeling, F. Gimeno, A. Dal Lago, et al., Competition-level code generation with AlphaCode, Science 378 (2022) 1092–1097.

[6] S. Piya, A. Sullivan, LLM4TDD: best practices for test driven development using large language models, in: Proceedings of the 1st International Workshop on Large Language Models for Code, 2024, pp. 14–21.

[7] N. S. Mathews, M. Nagappan, Test-driven development and LLM-based code generation, in: Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering, 2024, pp. 1583–1594.

[8] T. Ridnik, D. Kredo, I. Friedman, Code generation with AlphaCodium: From prompt engineering to flow engineering, 2024. arXiv:2401.08500.

[9] CodiumAI, AlphaCodium: Official implementation for the paper "code generation with alphacodium", https://github.com/Codium-ai/AlphaCodium, 2024. Accessed: 2025-09-28.

[10] M. Chen, J. Tworek, H. Jun, Q. Yuan, H. P. D. O. Pinto, J. Kaplan, H. Edwards, Y. Burda, N. Joseph, G. Brockman, et al., Evaluating large language models trained on code, arXiv preprint arXiv:2107.03374 (2021).

[11] AlphaCode Team, Google DeepMind, AlphaCode 2 technical report, https://storage.googleapis.com/deepmind-media/AlphaCode2/AlphaCode2_Tech_Report.pdf, 2023. Accessed: 2025-09-28.

[12] S. Piya, A. Samadi, A. Sullivan, Is more or less automation better? an investigation into the LLM4TDD process, in: 2025 IEEE/ACM International Workshop on Large Language Models for Code (LLM4Code), IEEE, 2025, pp. 161–168.

[13] S. Fakhoury, A. Naik, G. Sakkas, S. Chakraborty, S. K. Lahiri, LLM-based test-driven interactive code generation: User study and empirical evaluation, IEEE Transactions on Software Engineering (2024).

[14] J. Austin, A. Odena, M. Nye, M. Bosma, H. Michalewski, D. Dohan, E. Jiang, C. Cai, M. Terry, Q. Le, et al., Program synthesis with large language models, arXiv preprint arXiv:2108.07732 (2021).

[15] T. Kojima, S. S. Gu, M. Reid, Y. Matsuo, Y. Iwasawa, Large language models are zero-shot reasoners, Advances in neural information processing systems 35 (2022) 22199–22213.

[16] G. Rosa, J. M. Gonzalez-Barahona, Replication Package for "Evaluating Test-Driven Code Generation: A Replication Study", 2025. https://doi.org/10.6084/m9.figshare.30934718.

[17] X. Gu, M. Chen, Y. Lin, Y. Hu, H. Zhang, C. Wan, Z. Wei, Y. Xu, J. Wang, On the effectiveness of large language models in domain-specific code generation, ACM Transactions on Software Engineering and Methodology 34 (2025) 1–22.

[18] N. Jain, K. Han, A. Gu, W.-D. Li, F. Yan, T. Zhang, S. Wang, A. Solar-Lezama, K. Sen, I. Stoica, LiveCodeBench: Holistic and contamination free evaluation of large language models for code, arXiv preprint arXiv:2403.07974 (2024).

[19] S. Baltes, F. Angermeir, C. Arora, M. Muñoz Barón, C. Chen, L. Böhme, F. Calefato, N. Ernst, D. Falessi, B. Fitzgerald, et al., Guidelines for empirical studies in software engineering involving large language models, arXiv e-prints (2025) arXiv–2508.