

Article

GPT-4.1 Sets the Standard in Automated Experiment Design Using Novel Python Libraries

Nuno Fachada ^{1,2,*} , Daniel Fernandes ¹ , Carlos M. Fernandes ^{1,2} , Bruno D. Ferreira-Saraiva ^{1,3} 
and João P. Matos-Carvalho ^{2,4} 

- ¹ Copelabs, Lusófona University, Campo Grande, 376, 1749-024 Lisboa, Portugal; daniel.fernandes@ulusofona.pt (D.F.); p7582@ulusofona.pt (C.M.F.); bruno.saraiva@ulusofona.pt (B.D.F.-S.)
- ² Center of Technology and Systems (UNINOVA-CTS) and Associated Lab of Intelligent Systems (LASI), 2829-516 Caparica, Portugal
- ³ CISCANT, Lusófona University, Campo Grande, 376, 1749-024 Lisboa, Portugal
- ⁴ LASIGE and Departamento de Informática, Faculdade de Ciências, University of Lisbon, Campo Grande, 1749-016 Lisboa, Portugal; jpecarvalho@ciencias.ulisboa.pt
- * Correspondence: nuno.fachada@ulusofona.pt

Abstract

Large language models (LLMs) have advanced rapidly as tools for automating code generation in scientific research, yet their ability to interpret and use unfamiliar Python APIs for complex computational experiments remains poorly characterized. This study systematically benchmarks a selection of state-of-the-art LLMs in generating functional Python code for two increasingly challenging scenarios: conversational data analysis with the *ParShift* library, and synthetic data generation and clustering using *pyclugen* and *scikit-learn*. Both experiments use structured, zero-shot prompts specifying detailed requirements but omitting in-context examples. Model outputs are evaluated quantitatively for functional correctness and prompt compliance over multiple runs, and qualitatively by analyzing the errors produced when code execution fails. Results show that only a small subset of models consistently generate correct, executable code. GPT-4.1 achieved a 100% success rate across all runs in both experimental tasks, whereas most other models succeeded in fewer than half of the runs, with only Grok-3 and Mistral-Large approaching comparable performance. In addition to benchmarking LLM performance, this approach helps identify shortcomings in third-party libraries, such as unclear documentation or obscure implementation bugs. Overall, these findings highlight current limitations of LLMs for end-to-end scientific automation and emphasize the need for careful prompt design, comprehensive library documentation, and continued advances in language model capabilities.

Keywords: large language models; code generation; Python libraries



Academic Editors: Sardar Jaf, Basel
Barakat and Yongqiang Cheng

Received: 30 July 2025

Revised: 28 August 2025

Accepted: 5 September 2025

Published: 8 September 2025

Citation: Fachada, N.; Fernandes, D.; Fernandes, C.M.; Ferreira-Saraiva, B.D.; Matos-Carvalho, J.P. GPT-4.1 Sets the Standard in Automated Experiment Design Using Novel Python Libraries. *Future Internet* **2025**, *17*, 412. <https://doi.org/10.3390/fi17090412>

Copyright: © 2025 by the authors. Licensee MDPI, Basel, Switzerland. This article is an open access article distributed under the terms and conditions of the Creative Commons Attribution (CC BY) license (<https://creativecommons.org/licenses/by/4.0/>).

1. Introduction

The automation of computational experiments is a critical component of modern scientific research, enabling efficient exploration of hypotheses, reproducible analysis pipelines, and scalable data-driven discovery. In recent years, large language models (LLMs) have emerged as powerful tools capable of generating executable code from natural language descriptions [1–4], raising the possibility of automating significant portions of the experimental workflow with minimal human intervention [5–7]. These advancements hold particular promise for researchers and practitioners who wish to rapidly prototype,

modify, or extend computational analyses without requiring deep expertise in software engineering or specific programming libraries.

Despite rapid progress in LLM capabilities, a central challenge remains: the reliable generation of correct and functional code in larger computational experiments, especially when tasks involve lesser-known third-party libraries with limited online presence beyond their official documentation. The ability of LLMs to parse, interpret, and accurately use unfamiliar application programming interfaces (APIs) is crucial for their broader adoption as assistants or agents in real-world scientific settings. Current evidence suggests that LLM performance varies widely with prompt design, library familiarity, and task complexity, often requiring expert oversight or extensive validation to ensure robustness and correctness [1,3,8,9].

This study systematically evaluates the ability of state-of-the-art LLMs to generate functional Python code for complex computational experiments involving two distinct and progressively challenging tasks. Both tasks require the correct integration of specialized Python libraries—*ParShift* [10] for conversational data analysis and *pyclugen* [11] for synthetic data generation—posing realistic scenarios where knowledge of obscure APIs is essential. The evaluation considers a diverse set of leading LLMs, each prompted in a zero-shot setting to maximize reproducibility and emulate the experience of a domain expert querying the model with a technically precise request [8,12].

To achieve a comprehensive assessment, the study employs a multi-level analytical framework. Model outputs are scored for correctness and functional compliance; performance is quantified in terms of overall success rates, and the success rates of different models are statistically compared to determine whether observed differences are significant. Additionally, the variability and consistency of generated outputs are systematically analyzed, providing insight into the stochastic nature of LLM-generated code and the factors influencing its reliability. When outputs fail to execute correctly, error messages and failure modes are qualitatively examined to identify recurring issues and diagnose challenges related to API misuse, type mismatches, and prompt misinterpretation. Finally, for the subset of fully correct outputs, code quality is examined through several software engineering metrics, summarized using descriptive statistics and complemented by an analysis of distributional characteristics to provide a quantitative assessment.

By rigorously benchmarking LLMs on realistic, domain-specific tasks, this work helps characterize the current capabilities of automated code generation for computational experiments. The results show that only a small subset of models—GPT-4.1, Grok 3, and Mistral Large—consistently produced correct, executable code throughout the evaluated tasks. Beyond model performance, the study reinforces the relevance of structured zero-shot prompts as a practical and reproducible baseline for evaluating LLMs in isolation, prior to the confounding effects of multi-shot examples, self-refinement, or external tool use. Moreover, the study demonstrates the potential that these types of experiments have in identifying documentation gaps and edge cases in third-party libraries, offering valuable feedback for library authors and developers of scientific software.

The subsequent sections of this paper are organized as follows. Section 2 provides background information on the use of LLMs for domain-specific code generation and the automation of computational experiments. Section 3 describes the materials and methods employed, including the problem context, prompt design, model selection, experimental setup, and evaluation procedures. Results are presented in Section 4, followed by a detailed discussion in Section 5. Section 6 outlines the study's limitations and opportunities for future research. Finally, Section 7 concludes the paper with final remarks and recommendations.

2. Background

This section reviews relevant literature and provides essential context on how LLMs are being used to support scientific research. It begins by describing a range of scientific tasks where LLMs have already proven useful, such as forming hypotheses, conducting peer-review and performing literature review. Then, it examines frameworks that convert natural-language protocols into executable workflows, manage experimental variables and constraints, and generate ready-to-run code. The final subsection highlights advances in crafting effective prompts to steer model behavior and ensure consistent and reproducible outputs in scientific experimentation.

2.1. LLM in Research and Development

The integration of LLMs into scientific research introduces new workflows in research and development, wherein natural language processing systems assist or autonomously execute a range of scientific tasks, including protocol optimization, data analysis, and experimental design.

A key application of LLMs is automating hypothesis generation. In [13], for instance, Xiong et al. introduce KG-CoI, a system that integrates external structured and unstructured domain-specific knowledge from scientific resources into LLMs, including hallucination detection, to support more reliable scientific exploration. KG-CoI significantly reduces factual errors and increases the relevance of generated hypotheses. In [14], the authors propose PiFlow, a multi-agent LLM framework that integrates domain-specific scientific principles into hypothesis generation to overcome LLMs' difficulty in consistently linking hypotheses with evidence. PiFlow achieved a 94% improvement in solution quality compared to a single LLM conducting the discovery process independently, across applications in nanomaterials and biomolecular research.

Jin et al. [15] present AgentReview, a simulation framework where LLMs emulate the roles of reviewers, authors, and area chairs to analyze biases and decision variability in scientific peer review. By modeling latent sociological factors like authority bias and altruism fatigue, the study reveals that LLM-based agents can offer insights into improving the fairness and robustness of scientific evaluation systems.

In [16], Agarwal et al. introduce LitLLM, a system for generating literature reviews from an abstract using off-the-shelf LLMs. The authors show that LitLLM substantially reduces time and effort for literature review compared to traditional methods.

As the studies reviewed above attest, LLMs are already reshaping key phases of the scientific workflow. However, translating ideas into experiments still depends on detailed protocol planning and execution. In the following section, we discuss how LLM-based agents can draft detailed experimental protocols, and adapt workflows in real time, effectively turning high-level research designs into implementable laboratory action.

2.2. LLMs for Experiment Planning and Implementation

LLMs are increasingly being integrated into computational experiment design workflows, particularly through Python and its rich set of libraries. These pipelines automate repetitive tasks, shorten iteration times, and improve reproducibility [17]. Recently, Charness et al. [7] argued that LLMs can or at least could soon be integrated into the design, implementation, and analysis of social and economic experiments. The remainder of this subsection examines representative approaches to harnessing these capabilities for robust and scalable experiment planning and implementation.

One of the already established abilities of LLMs in assisting scientific research is code generation, either by producing ready-to-run code or by supporting researchers in the iterative development of scripts, pipelines, and analytical workflows. Over time, LLMs

have moved beyond simple code snippets to domain-specific pipelines, enabling more sophisticated prototyping and documentation.

In [18], the authors adapt LLMs trained on code completion for writing robot policy code according to natural language prompts. The generated robot policies exhibit spatial-geometric reasoning and are able to prescribe precise values to ambiguous descriptions. By relying on a hierarchical prompting strategy, their approach is able to write more complex code and solve 39.8% of the problems on the HumanEval [1] benchmark.

Luo et al. [19] use LLMs to generate robot control programs, testing and optimizing the output in a simulation environment. After a number of optimization rounds, the robot control codes are deployed on a real robot for construction assembly tasks. The experiments show that their approach can improve the quality of the generated code, thus simplifying the robot control process and facilitating the automation of construction tasks.

In [4], the authors present a comparative evaluation of 16 LLMs in generating Python code for UAV placement and signal power calculations in LoRaWAN environments, demonstrating that state-of-the-art models like DeepSeek-V3 and GPT-4 consistently produce accurate solutions, while smaller, locally executable models such as Phi-4 and LLaMA-3.3 also show strong performance, highlighting the viability of lightweight alternatives. The study demonstrates the importance of domain-specific fine-tuning, offering valuable insights into the practical deployment of LLMs for specialized engineering tasks.

LLMs can write individual scripts, but full experimental design demands additional abstraction, planning, and domain expertise. It requires integrating hypotheses, protocol structure, data collection strategies, ethical considerations, and analysis plans, a complex orchestration far beyond simple script generation.

A method for evaluating LLMs' ability to generate and interpret experimental protocols is proposed in [5]. The authors describe BioPlanner, a framework that converts natural language protocols into pseudocode and then assesses a model's ability to reconstruct those protocols using a predefined set of admissible pseudofunctions. This approach enables robust, automated evaluation of long-horizon planning tasks in biology, reducing reliance on manual review and paving the way for scalable scientific automation.

Hong et al. [6] introduce Data Interpreter, an LLM-based agent designed to automate end-to-end data science workflows, addressing the limitations of existing LLMs in handling complex and dynamic data science tasks. The system decomposes complex data science tasks into graph-based subproblems and dynamically adapts workflows via programmable node generation. It embeds a confidence-driven verifier to flag and correct logical inconsistencies in generated code. In benchmarks like InfiAgent-DABench and MATH, it achieves roughly 25–26% gains over open source baselines.

This is where our proposal takes shape: it evaluates how well an LLM can parse APIs and assemble complete computational experiments from a single prompt, without further human guidance. While this setup is not agentic in nature, it does bear a superficial resemblance to recent work on LLM-based agents, such as Hong et al.'s Data Interpreter [6], ReAct [20], or Auto-GPT [21]. These frameworks emphasize multi-step reasoning, interactive planning, and iterative tool use, often allowing models to execute, verify, and refine their own outputs. In contrast, the methodology used in this paper deliberately isolates and benchmarks a model's intrinsic ability to interpret unfamiliar APIs and return fully functional code in a single pass, guided only by a zero-shot structured prompt. In this sense, our study is complementary to agent research: by establishing a baseline of core code generation competence, it provides a foundation for understanding the prerequisites upon which more complex agentic systems ultimately depend. Prompt design thus becomes pivotal to the scope of this investigation, and accordingly, the next section focuses on prompt engineering and its foundational principles.

2.3. Prompt Design

The piece of text or set of instructions that a user provides to an LLM to elicit a specific response is referred to as a prompt. Designing effective prompts has therefore become essential to harness the full capabilities of LLMs, and in recent years this craft has evolved into a distinct field of research and development [8]. Advanced prompting techniques—such as in-context learning [22], chain-of-thought [12,19], and ReAct [20]—are frequently employed in studies to improve the reliability and precision of experimental planning within LLM-assisted workflows.

Prompting strategies are often described along multiple dimensions. One common distinction is between structured and unstructured prompts. Structured prompts use precise formulations, with clearly defined inputs, outputs, and constraints, and tend to yield more consistent results—particularly in tasks like code generation. However, structured prompts often demand detailed knowledge of both the problem space and the model’s behavior, which can hinder accessibility for non-experts. Unstructured prompting, in contrast, adopts a more conversational style, making it easier to use in real-world scenarios where users might lack technical expertise. However, this flexibility comes at a cost, as it can lead to output inconsistencies due to the inherent ambiguity of informal language. Table 1 illustrates a minimal example of this structured vs. unstructured prompting dimension.

Table 1. Minimal example of unstructured and structured prompts for the same programming task. The unstructured prompt is informal and open-ended, while the structured prompt specifies function details and expected input/output, illustrating the difference in detail and reproducibility.

Prompt Type	Prompt Example
Unstructured	How do I sum a list of whole numbers in Python?
Structured	Write a Python function <code>sum_numbers(numbers: list[int]) -> int</code> that returns the sum of a list of integers.

Prompts can also be distinguished by the number of examples they provide to guide the model: zero-shot prompts offer no examples, one-shot prompts include a single instance, and few-shot prompts present several illustrations. Each style carries its own trade-offs, as supported by empirical findings—for instance, Liang et al. [18] showed that structured prompts containing code consistently outperform those written in plain language for tasks involving robotic reasoning. That said, ongoing advancements in LLM capabilities continue to narrow this gap, with recent studies demonstrating promising results for natural-language-based prompting in specialized domains like robotics [23].

Complementing existing techniques, our method employs a fully structured, zero-shot prompt that embeds the relevant API docstrings and specifies, without examples, every step needed to assemble a complete computational experiment. This design tests the model’s ability to interpret unfamiliar Python APIs, assemble complete computational workflows—from data extraction and analysis to multi-step data generation, clustering, and evaluation—and return fully functional solutions in a single pass.

3. Materials and Methods

This section details the methodology adopted to assess the ability of state-of-the-art LLMs to interpret and use Python APIs for automated computational experiment design. The following subsections describe the experimental scenarios and prompt construction (Section 3.1), evaluation pipeline (Section 3.2), selection of LLMs (Section 3.3), experimental protocol (Section 3.4), and data analysis approach (Section 3.5).

3.1. Scenarios and Prompts

To evaluate the performance of the selected LLMs (discussed in Section 3.3), two experimental scenarios were prepared, each associated with a carefully designed, structured zero-shot prompt. The zero-shot approach, which does not provide examples within the prompt, was chosen to maximize reproducibility, minimize potential bias from hand-crafted demonstrations, and more closely reflects typical initial queries issued by domain experts to LLMs for technical tasks [8,12]. Zero-shot prompting also facilitates direct benchmarking of models' intrinsic capabilities in code generation and comprehension.

Both prompts employ a highly structured format, specifying requirements such as the function name, input and output types, the set of allowable libraries, coding standards (e.g., indentation and style), and the necessity for the function to be self-contained. This structure serves to minimize ambiguity, streamline post-processing, and ensure fair and consistent evaluation between models and runs [8,12]. Moreover, by explicitly detailing the APIs to use, including relevant docstrings, the prompts challenge models to demonstrate genuine understanding and integration of unfamiliar libraries, rather than relying on memorized patterns from training data. However, since the documentation of the tested libraries is publicly available since 2023, a degree of prior exposure cannot be fully ruled out.

The two experimental scenarios were designed to reflect progressively increasing complexity and to probe distinct aspects of LLM-driven code synthesis using recent, peer-reviewed research software. Both the *ParShift* [10] and *pyclugen* [11] libraries, although not widely known, are open source, fully unit tested, extensively documented, and readily available via the Python Package Index, thus providing a reliable and transparent foundation for the study.

The first experiment, using *ParShift*, requires LLMs to generate a Python function, `do_parshift_exp()`, that processes a list of CSV files containing conversational data, extracting the proportion of utterances pairs labeled as “Turn Usurping” from each file, and returning the results as a list of floats. This scenario is hypothesized to be relatively straightforward, involving a linear workflow that primarily tests basic API usage, data parsing, and compliance with explicit prompt constraints. The prompt, with API docstrings omitted for space considerations, is detailed in Table 2.

Table 2. Prompt used in Experiment 1 (*ParShift*). The table shows the full prompt excluding API docstrings; the complete prompt, including all referenced docstrings, is available in the supplementary material [24].

Prompt 1

Create a Python function named `do_parshift_exp()` that analyses CSV files containing conversations as described below. This function accepts a list of file paths (`List[str]`), each pointing to a CSV file, and returns a corresponding list of floats.

Specifically, the analysis of these conversations should be done with the `parshift` library, whose API documentation is provided below in the form the respective source docstrings:

```
```python
{{DOCSTRINGS HERE}}
```
```

The `do_parshift_exp()` function must analyse the conversations in the CSV files as follows:

1. For each CSV file specified in the list given as the function's argument, extract the proportion of the "Turn Usurping" class (as a float). Here, proportion is the sum of the "Frequency" divided by all frequencies.
2. Collect all the proportions of 'Turn Usurping' into a list of floats.
3. Return the list of floats, with one entry per file.

The `do_parshift_exp()` function should strictly follows these requirements:

- The function must be self-contained. In other words, all variables, constants, and/or helper functions must be defined within the `do_parshift_exp()` function.
- Beyond the Python standard library, only the `parshift`, `pandas`, and `numpy` libraries are allowed.
- The function should be compatible with Python 3.9 and above and follow the PEP 8 style guidelines, with 4-space indentation.
- Each input file uses a semicolon (;) as a delimiter and contains conversational data as expected by the `parshift` library.
- Do not include print statements, plots, or additional output.

Please respond with a single code block only. Inline comments within the code are fine, but do not include any explanation or text outside the code block.

The second experiment employs *pyclugen* to significantly increase task complexity. Here, LLMs must generate a function, `do_clustering_exp()`, that conducts a full clustering analysis by generating multiple synthetic datasets in three dimensions (with varying cluster dimensions along a general direction), applying several clustering algorithms available in *scikit-learn* [25], evaluating clustering quality using the V-measure [26], and collating detailed results (including algorithm, parameter values, and seed) in a *Pandas* data frame. This multi-step pipeline tests the LLMs' ability to integrate unfamiliar APIs, manage parameterization, and synthesize robust, structured code. Potential error points include misuse of APIs (both from *pyclugen* and *scikit-learn*), mishandling of *pyclugen* output, and inconsistent output formatting. The prompt is provided in Table 3—the API docstrings are again omitted for brevity.

The two scenarios were first implemented and validated by the authors to establish the baseline results used for comparison with the LLM-generated functions. Both the baseline implementations and their results are provided in the supplementary material [24].

3.2. Evaluation Pipeline

The process of evaluating LLM-generated Python code is depicted in Figure 1. For each experiment, all combinations of predefined LLMs, random seeds, and prompts are systematically iterated. Each prompt is submitted to the respective LLM, and the resulting output is stored.

Function extraction from LLM responses proceeds in up to two steps: (1) code is extracted from the first Markdown code block, specifically from the text enclosed between the ````python` opening delimiter and the closing ````` (that is, the code is expected to be located within these code fences); or (2) if the previous step fails, the entire response is assessed as potential Python code. If neither approach yields valid code, the attempt is logged and assigned a score of 1, as illustrated in Figure 1. Both prompts explicitly instruct the LLMs to output a single code block containing only the function implementation; nonetheless, these extraction steps provide tolerance for minor deviations from reply formatting. Upon successful extraction, the function is saved as a Python file for subsequent execution.

Extracted code is then executed in a controlled environment. In Experiment 1 (*ParShift*), the extracted `do_parshift_exp()` function is tested using three CSV files, collecting the respective turn-usurping proportions. In Experiment 2, `do_clustering_exp()` is executed with $n = 30$ samples, with results collected as a *Pandas* data frame. If execution fails due to syntax or runtime errors, the error in question is logged and a score of 2 is assigned.

Successful execution leads to an automated validation of the function's output type and format. For Experiment 1, the output must be a list of three floats; for Experiment 2, the expected output is a data frame with specific columns and dimensions. Mismatches in type or structure result in a score of 3. If the output format is correct, values are compared to pre-computed baselines: a 1% numerical tolerance is allowed for Experiment 1, while statistical indistinguishability is required for Experiment 2 (see Section 3.5 for details). Outputs deviating from the baseline receive a score of 4; otherwise, a perfect score of 5 is assigned.

This evaluation procedure is summarized as follows:

1. No Python code could be extracted (score 1).
2. Code fails to run due to syntax or runtime errors (score 2).
3. Code runs but returns an output of incorrect type or structure (score 3).
4. Code runs and returns the correct type/structure, but with incorrect results (numerically/statistically different from baseline; score 4).
5. Code runs and returns the correct, baseline-matching result (score 5).

All outcomes, scores, and error logs are recorded for further analysis. This systematic, automated pipeline enables robust and reproducible benchmarking of the LLM-generated code.

Table 3. Prompt used in Experiment 2 (*pyclugen*). The table presents the prompt text without the included API docstrings; the complete version with all API docstrings can be found in the supplementary material [24].

Prompt 2

Create a Python function named `do_clustering_exp()` that runs a comprehensive clustering experiment. This function accepts a single `n_samples` argument of type `int`, and returns a pandas DataFrame.

Specifically, the experiment involves generating synthetic clustered datasets using the `clugen` function from the *pyclugen* library, whose API documentation is provided below in the form the respective source docstrings:

```
```python
{{DOCSTRINGS HERE}}
```
```

The computational experiment requirements are as follows:

- The dimensionality (`num_dims`) should be set to 3.
- The number of clusters (`num_clusters`) should be 10.
- Each dataset should contain exactly 10,000 points (`num_points`).
- The average cluster direction vector (`direction`) is set to a 3-dimensional vector of ones.
- The standard deviation of cluster direction angles (`angle_disp`) should be $\pi/16$ radians.
- The average separation between cluster centers (`cluster_sep`) should be fixed at 50 for each dimension.

Additionally, perform a parameter sweep over different values of the average cluster-supporting line length (`llength`):

- The average lengths (`llength`) should range from 0 to 800, inclusive, with steps of 25 (i.e., `[0, 25, 50, ..., 800]`).

Finally:

- The length dispersion of cluster-supporting lines (`llength_disp`) should be set to $0.2 * llength$.
- The cluster lateral dispersion (`lateral_disp`) should be set to $0.5 * llength$.

For each dataset generated, apply the following clustering algorithms from scikit-learn:

- K-means++ (`KMeans` with `init='k-means++'`)
- Gaussian Mixture Model clustering (`GaussianMixture`)
- Agglomerative Hierarchical Clustering (`AgglomerativeClustering`) with the following linkage methods:
 - Ward linkage
 - Single linkage
 - Complete linkage
 - Average linkage (with Euclidean distance)

Please note that in recent version of scikit-learn, the `AgglomerativeClustering` class no longer accepts the `affinity` parameter. This parameter is now called `metric` which is set to `euclidean` by default.

Evaluate the clustering quality for each run using the V-measure (`v_measure_score`) provided by scikit-learn, comparing predicted cluster labels to the ground truth labels provided by `clugen`.

For every run, record the following into a pandas DataFrame:

- `run`: Sample run index (from 0 to `n_samples - 1`)
- `algorithm`: Name of the clustering algorithm (e.g., `"kmeans++"`, `"em"`, `"ahc-ward"`, `"ahc-single"`, `"ahc-complete"`, `"ahc-avg"`)
- `length`: The current average line length (`llength`)
- `vmeas`: Calculated V-measure score
- `seed`: Random seed used for generating the dataset

The `do_clustering_exp()` function should return this pandas DataFrame.

The `do_clustering_exp()` function should strictly follows these requirements:

- The function must be self-contained. In other words, all variables, constants, and/or helper functions must be defined within the `do_clustering_exp()` function.
- Beyond the Python standard library, only the *pyclugen*, *numpy*, *pandas*, *sklearn*, and *scipy* libraries are allowed.
- The function should be compatible with Python 3.9 and above and follow the PEP 8 style guidelines, with 4-space indentation.
- Do not include print statements, plots, or additional output.

Please respond with a single code block only. Inline comments within the code are fine, but do not include any explanation or text outside the code block.

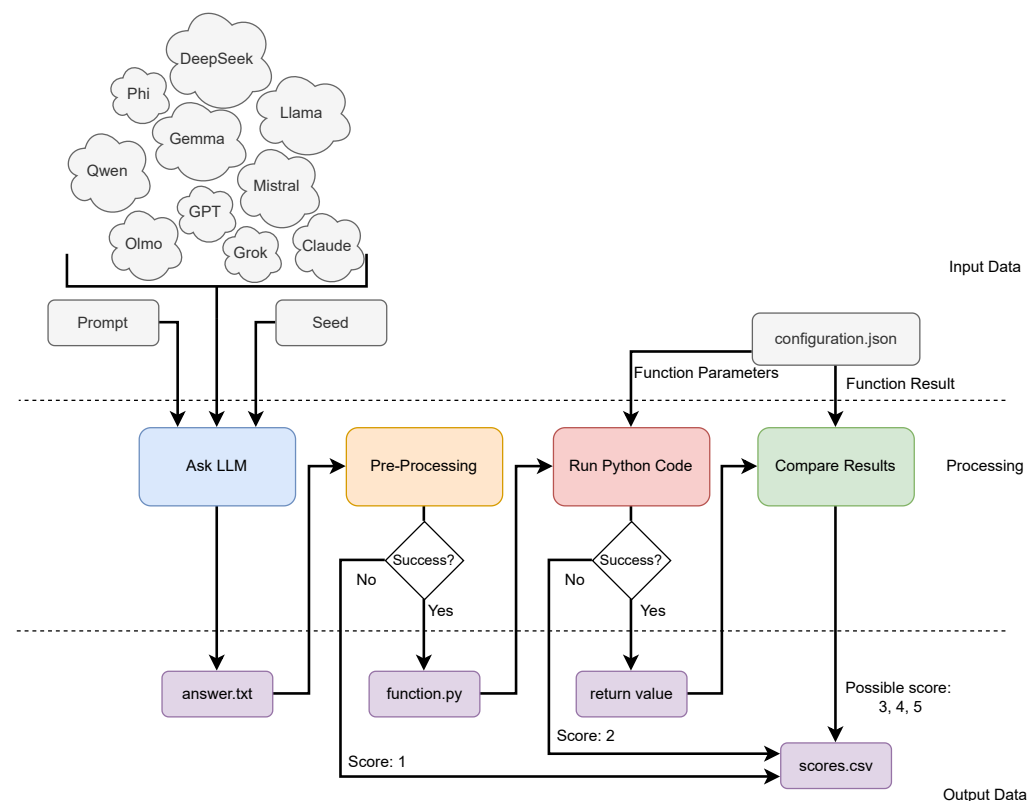


Figure 1. Evaluation pipeline for assessing LLM-generated Python code. Each prompt is submitted to an LLM with a random number generator seed, followed by code extraction, execution, and validation. Outputs are scored based on success at each stage, from parsing to result accuracy, enabling reproducible and structured performance assessment.

3.3. LLMs Considered

The language models evaluated in this work, summarized in Table 4, were selected for their impact in Artificial Intelligence research, diverse methodological approaches, and demonstrated strengths in code generation, reasoning, and efficiency. Geographic diversity was also prioritized to ensure broad representation of current LLM development. Model sizes (parameters, tokens, etc.) are indicated in millions (M), billions (B), or trillions (T), with uppercase letters used throughout for consistency.

Claude 3.7 Sonnet [27], developed by Anthropic, is a state-of-the-art multimodal LLM capable of processing both textual and visual input. The model supports both rapid-response and extended-reasoning modes and is optimized for handling complex instruction-following, including coding tasks and natural language understanding. Claude 3.7 Sonnet is a representative advanced general-purpose LLM, demonstrating strong performance in both standard and zero-shot Python code synthesis tasks.

DeepSeek Coder-V2 [28], R1 [29], and V3 [30] belong to DeepSeek’s open source LLM lineup. Coder-V2 is an Mixture-of-Experts (MoE) model specialized for code and mathematics, supporting 338 programming languages. V3 is DeepSeek’s 671B MoE general-purpose model, employing multi-head latent attention and reinforcement learning (RL) from human feedback for multilingual reasoning. R1 is a dense reasoning model derived from V3 using multi-stage RL—initially via pure RL and later with cold-start supervised data—to improve math, logic, and code reasoning. The inclusion of these models demonstrates how conditional computation (via MoE) and scale benefit Python code generation, especially in zero-shot scenarios with sparse online code examples.

Table 4. LLMs tested in this study. ‘Patch/Date’ indicates the model’s exact release patch, if available, otherwise displaying the patch release date. ‘Size’ indicates the number of parameters in billions (B), when available. ‘Mode’ shows whether the model was executed offline via Ollama in the author’s institution infrastructure, or if it was experimented with online through the respective parent company’s infrastructure. ‘Tag’ corresponds to how each specific model is mentioned in the figures and tables of the results section.

| Family | Model/Version | Ref. | Patch/Date | Size | Mode | Tag |
|----------|---------------|------|------------|------|---------|-------------------|
| Claude | Sonnet 3.7 | [27] | 20250219 | ★ | Online | claude-3.7-sonnet |
| DeepSeek | coder-v2 | [28] | 2024-09-06 | 16B | Offline | deepseek-coder-v2 |
| | R1 | [29] | 2025-01-20 | 671B | Online | deepseek-r1 |
| | V3 | [30] | 0324 | 671B | Online | deepseek-v3 |
| Gemma | code 3.0 | [31] | 2024-07-18 | 7B | Offline | codegemma |
| | | [32] | 2025-04-18 | 27B | Offline | gemma3 |
| GPT | 4o | [33] | 2024-08-06 | ★ | Online | gpt-4o |
| | 4.1 | [34] | 2025-04-14 | ★ | Online | gpt-4.1 |
| Grok | 3-beta | [35] | 2025-02-17 | ★ | Online | grok-3-beta |
| LLaMA | 3.3 | [36] | 2024-12-06 | 70B | Offline | llama3.3 |
| | code | [37] | 2024-07-18 | 70B | Offline | code llama |
| Mistral | codestral 0.1 | [38] | 2024-09-03 | 22B | Offline | codestral |
| | Large 2.1 | [39] | 24.11 | 123B | Online | mistral-large |
| Olmo | 2 | [40] | 2025-01-11 | 13B | Offline | olmo2 |
| Phi | 4.0 | [41] | 2025-01-08 | 14B | Offline | phi4 |
| Qwen | 2.5-coder | [42] | 2025-05-28 | 32B | Offline | qwen2.5-coder |
| | 3.0 | [43] | 2025-05-29 | 32B | Offline | qwen3 |

★ Not publicly disclosed.

CodeGemma [31] and Gemma3 [32] are based on Google’s lightweight Gemma architecture. CodeGemma (7B) is fine-tuned specifically for code completion and delivers strong performance on Python benchmarks, while Gemma3 (27B) is a general-purpose, multimodal model featuring long context windows and instruction tuning. These models allow examination of how compact, optimized open weight architectures perform in zero-shot code generation, especially when integrating lesser-known library APIs.

GPT-4o [33] and GPT-4.1 [34] are successive generations of OpenAI’s flagship LLMs. GPT-4o (“omni”) is a unified multimodal model for text, image, audio, and video, trained end-to-end and achieving leading text and code performance with improved multilingual and latency characteristics. GPT-4.1 is oriented towards developer applications, with a million-token context window and substantial improvements in code generation. Including both enables direct comparison between generalist multimodal (GPT-4o) and developer-optimized (GPT-4.1) architectures in zero-shot Python code synthesis with uncommon libraries.

Grok 3 [35], developed by xAI, is notable for its exceptionally large context window, officially claimed to support up to 1M tokens. This extended context capacity enables the model to capture long-range dependencies, which is valuable for complex code generation tasks. The model’s advanced reasoning abilities and long-context handling make it a relevant benchmark for zero-shot application of novel libraries.

Llama 3.3 [36] and CodeLlama [37], both from Meta, share a decoder-only transformer backbone but diverge in specialization. Llama 3.3 is a multilingual, instruction-tuned model with strong general-language and reasoning capabilities, while CodeLlama is fine-tuned for code—particularly Python—demonstrating strong performance at infilling and instruction-following. Their comparison illustrates the effect of generalist versus code-specialized tuning on Python code generation with rarely used APIs.

Codestral [38] and Mistral Large 2.1 [39], from Mistral AI, represent recent advances in both code-focused and general-purpose transformer models from Europe. Codestral is a compact (22B parameter) model designed specifically for code generation, supporting over 80 programming languages and offering competitive performance and latency on code benchmarks. Mistral Large 2.1 is a large (123B parameter), dense model with a 128K token context window, demonstrating strong reasoning and code generation capabilities. Including both allows assessment of architectural diversity and regional competitiveness in LLM-based code generation.

Olmo2 [40] (13B), from the Allen Institute for AI, is a fully open source dense transformer trained on approximately 5T tokens. Its rigorous engineering and transparent training pipeline distinguish it among open models, supporting robust reasoning and general language understanding. Olmo2 offers a benchmark for transparent, research-driven LLMs in zero-shot Python code synthesis with less-common libraries.

Phi-4 [41], a 14B parameter model from Microsoft Research, exemplifies a data-centric training approach, leveraging high-quality synthetic data throughout pretraining and post-training to advance reasoning, mathematical problem-solving, and code generation. Phi-4's strong performance relative to larger models—as also observed in a previous study from our group [4]—underlines the influence of training methodology over sheer parameter count, and its compact size makes it well-suited for resource- or latency-constrained deployment.

Qwen2.5-Coder [42] and Qwen3 [43], developed by Alibaba Cloud, expand the study's geographic and architectural breadth. Qwen2.5-Coder, a dense model with a 128K-token context window, is reported to perform well in code generation tasks through synthetic and code-centric pretraining. Qwen3 offers both dense and MoE variants, dual-mode inference, adaptive compute, and multilingual reasoning, achieving competitive results in code and logic tasks. Together with the DeepSeek models, the inclusion of Alibaba's LLMs provides insight into the capabilities of open-weight LLMs from Asia in Python code generation when guided by prompts that include explicit API docstrings.

While every effort was made to include a broad range of state-of-the-art models, certain recent releases were excluded due to practical limitations in time, budget, hardware, or technical access. For example, Gemini models were omitted due to payment system issues with Google during the study period, and Llama4 exceeded available local hardware capacity. The rapid proliferation of LLMs precludes exhaustive coverage in a single study. Nonetheless, the selection here captures a representative cross-section of contemporary architectures, parameter scales, technical strategies, and geographic origins, providing a robust foundation for evaluating Python code generation—particularly in the context of novel Python libraries.

3.4. Experimental Setup

To evaluate the capabilities of the models described in Section 3.3, the prompts detailed in Section 3.1 were submitted to each LLM in six separate runs, using distinct pseudo-random number generator seeds where supported. While seed control is intended to improve reproducibility, not all LLMs provide this functionality (e.g., Claude 3.7 Sonnet did not support seeding at the time of writing), and even when seeding is available, strict reproducibility cannot be assured. Sources of non-determinism include parallel computation on heterogeneous hardware (such as multi-GPU configurations), small discrepancies in floating-point arithmetic across hardware types and architectures, and potential variability arising from system drivers or firmware. Further, instability in the software environment—including differences in library versions, system configuration, and computational resource allocation—can also impact reproducibility [44,45]. Accordingly, this study focuses on assessing the general statistical consistency of model outputs over six runs per configuration,

rather than pursuing exact reproducibility throughout all models and conditions. As a consequence, random seeds are used on a best-effort basis, with the recognition that exact reproducibility may not be achievable under these conditions.

The temperature parameter for sampling was set to 10% of each model's maximum supported temperature (e.g., $T = 0.1$ if $T_{\max} = 1.0$), as this is generally considered adequate to allow for some stochasticity while still maintaining accurate problem-solving capabilities in code generation [4,46]. It should be noted, however, that not all models expose or respect temperature controls. For example, DeepSeek R1 (accessed via its online API) ignored temperature settings at the time of experimentation [47]. Furthermore, the documentation for some models is ambiguous regarding the maximum temperature value. For instance, Mistral's official sampling guide illustrates temperatures up to 3.2 [48], but in the present study a value of 0.2 was used, following Mistral's recommendations and to maintain consistency with other models employing a 0.0–2.0 range.

Other model parameters, such as `top_p` (which restricts next-token sampling to the most probable candidates summing to a specified cumulative probability), were left at their default values. Offline models were executed locally via Ollama [49] on the authors' institutional infrastructure, while online models were accessed through their official APIs.

For models supporting external tools—such as sandboxed code execution or web search (e.g., GPT-4o and GPT-4.1)—these functionalities were not enabled or activated. All models were therefore evaluated solely on their intrinsic capabilities in processing the prompt and associated API docstrings, without access to additional external information sources beyond what was explicitly provided within each prompt.

3.5. Data Collection and Analysis

For each experiment, the scores assigned to each LLM-generated function (ranging from 1 to 5) were analyzed to provide an overview of model performance over runs with different random seeds. Score distributions were visualized using histograms, allowing for a clear depiction of the relative frequency of each outcome per model over six runs. In addition to these distributions, the percentage of top scores (i.e., the proportion of runs yielding a perfect score of 5) was recorded, as this directly reflects the frequency with which each model succeeded in generating a fully correct solution.

To compare the proportion of correct answers between models, pairwise statistical testing was performed using Fisher's exact test [50]. For each model, its proportion of perfect scores (score equal to 5) was compared against that of each other model, and the number of statistically significant "wins" was recorded. Statistical significance is defined as a p -value below the conventional threshold of $\alpha = 0.05$. To account for multiple comparisons, p -values were adjusted using the Benjamini–Hochberg procedure for controlling the false discovery rate (FDR) [51], applied per model (i.e., to each model's set of pairwise comparisons).

In Experiment 2, to further investigate the quality of runnable LLM-generated functions—specifically, those able to return properly formatted outputs—their results were statistically compared to those of the baseline *pyclugen* experiment. Following the output comparison approach described in [52], the outputs of each function (V -measure as a function of average cluster length for each clustering algorithm, with centered and scaled results concatenated by algorithm) were subjected to principal component analysis (PCA). Statistical similarity was assessed using the Mann–Whitney U test [53] on the first principal component scores. In addition, PCA score plots were visualized to provide a qualitative assessment of potential differences between LLM-generated and baseline outputs.

In addition to functional correctness, the subset of fully correct code snippets (score 5) was further analyzed for code quality using four software engineering metrics: cyclomatic

complexity (c_c), maintainability index (m_i), number of type errors per 100 source lines of code ($e_t/100$), and number of F errors per 100 source lines of code ($e_F/100$). Source lines of code (s_{loc}), defined as the number of non-comment, non-blank lines, were also calculated to characterize code size and to normalize error counts; these values were obtained using *Radon* [54], a static analysis tool for Python. Cyclomatic complexity, also computed with *Radon*, quantifies the number of independent paths through a program and provides a measure of structural complexity [55], which is particularly relevant for assessing whether generated code remains simple and usable. The maintainability index, m_i , likewise measured with *Radon*, is a composite indicator on a 0–100 scale (higher is better) derived from c_c , s_{loc} , percentage of comment lines, and various measures of operators and operands. In this context, it provides an aggregate view of readability and long-term maintainability. Type errors were identified using *mypy* [56], a static type checker for Python, with results expressed as $e_t/100$ to account for code size and allow comparison between models; these errors indicate violations of type annotations and potential runtime failures. F errors were determined using *Ruff* [57], a high-performance Python linter that extends and replaces *Flake8* [58], detecting a broad range of issues including unused imports, undefined variables, duplicate definitions, and standard style violations. These issues are categorized under the traditional F error codes, reflecting their role in capturing logical and correctness problems beyond type checking or complexity analysis. As with type errors, results were normalized as $e_F/100$ to mitigate differences in code size. For all metrics, emphasis was placed on median values, which are more robust to skewed distributions and outliers, while box plots were used to visualize overall distributions and highlight variability. This enabled a quantitative summary of correct LLM-generated code quality, allowing comparisons not only between models but also across experiments, providing insights into both model-specific tendencies and differences in code quality associated with task complexity. Although there is some overlap between the metrics employed (e.g., cyclomatic complexity is one of the components of the maintainability index, and certain F errors may coincide with type-related issues), their joint use nevertheless provides a deeper assessment of code quality than any single metric alone.

Additional analyses are provided in the supplementary material [24] for interested readers. These include further descriptive statistics (e.g., mean, standard deviation, among others), as well as pairwise model comparisons of full score distributions using the Mann–Whitney U test (note that the main statistical analysis in the paper addresses the proportion of fully correct responses, score = 5, not the full score distribution). In addition, the supplementary material contains non-aggregated results and individual counts of type and F errors, allowing a more fine-grained inspection of model behavior. These supplementary analyses are not further discussed in the main text since: (1) descriptive statistics such as means or standard deviations may obscure important aspects of discrete, skewed, or multi-modal distributions; (2) while overall scores are useful to understand how and when models fail, emphasizing statistically significant differences in the complete score distribution between models may distract from the primary criterion of interest: the consistent generation of fully correct code; and, (3) detailed per-instance code quality results, although useful for in-depth inspection, extend beyond the central focus of this work on functional correctness.

Data analysis was conducted primarily in Python, using several established scientific computing libraries. *Pandas* [59] and *NumPy* [60] were employed for general data manipulation and numerical computations, while *SciPy* [61] provided statistical testing functionalities. The *statsmodels* package [62] was used to perform p -value corrections. Additionally, some analyses were performed in the R environment [63]. In particular, the *micompr* package [64] was used for statistically comparing multidimensional outputs from

the LLM-generated functions in Experiment 2 against a baseline, enabling differentiation between functions scoring 4 and those scoring 5. The specific versions of these packages are listed in Table A1.

4. Results

Table 5 summarizes the aggregated evaluation results for the 17 LLMs tested in the two experiments. Detailed pairwise model comparisons are provided in Table A2 for Experiment 1 and Table A3 for Experiment 2. Additionally, to distinguish between scores of 4 and 5 for Experiment 2, Table A4 provides the statistical comparison of outputs from LLM-generated functions against a predefined baseline.

Table 5. Distribution and significance of scores (1–5) for the 17 tested models in the two experiments. ‘Hist.’ shows the histogram (score distribution); ‘Top’ reports the percentage of top scores (i.e., scores of 5); and ‘Sig.’ indicates the number of models for which this model had a statistically significant higher proportion of top scores (Fisher’s exact test [50], $\alpha < 0.05$, with p -values corrected for FDR using the Benjamini–Hochberg method [51]; additional test details in Table A2 for Experiment 1, and Table A3 for Experiment 2).

| Model | Experiment 1 | | | Experiment 2 | | |
|-------------------|--------------|--------|------|--------------|--------|------|
| | Hist. | Top | Sig. | Hist. | Top | Sig. |
| claude-3.7-sonnet | | 33.3% | 0 | | 66.7% | 10 |
| codegemma | | 0.0% | 0 | | 0.0% | 0 |
| codellama | | 0.0% | 0 | | 0.0% | 0 |
| codestral | | 66.7% | 0 | | 0.0% | 0 |
| deepseek-coder-v2 | | 0.0% | 0 | | 0.0% | 0 |
| deepseek-r1 | | 33.3% | 0 | | 66.7% | 10 |
| deepseek-v3 | | 66.7% | 0 | | 33.3% | 0 |
| gemma3 | | 0.0% | 0 | | 0.0% | 0 |
| gpt-4.1 | | 100.0% | 12 | | 100.0% | 11 |
| gpt-4o | | 66.7% | 0 | | 50.0% | 0 |
| grok-3-beta | | 100.0% | 12 | | 83.3% | 10 |
| llama3.3 | | 0.0% | 0 | | 0.0% | 0 |
| mistral-large | | 33.3% | 0 | | 100.0% | 11 |
| olmo2 | | 0.0% | 0 | | 0.0% | 0 |
| phi4 | | 0.0% | 0 | | 0.0% | 0 |
| qwen2.5-coder | | 0.0% | 0 | | 0.0% | 0 |
| qwen3 | | 0.0% | 0 | | 0.0% | 0 |

To clarify the nature of execution failures that resulted in a score of 2, the corresponding error messages were grouped into three categories (Table 6). **Logic** errors refer to basic Python or reasoning flaws, such as missing imports or referencing variables before assignment. **Established API** errors arise from the incorrect use of widely adopted libraries such as *Pandas* or *scikit-learn*. Finally, **novel API** errors correspond to issues in handling the libraries under evaluation, specifically *ParShift* in Experiment 1 and *pyclugen* in Experiment 2. This categorization condenses a wide range of raw error messages into a small

set of recurring patterns, highlighting the main reasons for LLM-generated code failure in these experiments.

To complement the functional correctness analysis, code quality indicators for the subset of runs that produced fully correct code (score 5) are also reported. Table 7 summarizes, for each model and experiment, the medians of s_{loc} , c_c (cyclomatic complexity), m_i (maintainability index), $e_t/100$ (type errors per 100 s_{loc}), and $e_F/100$ (F errors per 100 s_{loc}), with n indicating the number of successful instances. The last row (“All instances”) reports experiment-wide medians (and total n). Figure 2 complements this table by displaying standard box plots of these metrics per model and experiment, providing a clearer view of distributional spread and potential outliers.

Table 6. Source of errors when code fails to run due to syntax or runtime errors (score 2) for the 17 tested models in the two experiments. ‘Logic’ errors include basic Python or reasoning errors such as missing imports or use of variable before assignment; ‘Est. API’ errors are due to incorrect use of established APIs such as *Pandas* or *scikit-learn*; finally, ‘Nov. API’ errors occur when misusing the novel API under test, i.e., *ParShift* in Experiment 1 and *pyclugen* in Experiment 2.

| Model | Experiment 1 | | | Experiment 2 | | |
|-------------------|--------------|----------|----------|--------------|----------|----------|
| | Logic | Est. API | Nov. API | Logic | Est. API | Nov. API |
| claude-3.7-sonnet | – | – | 4 | 2 | – | – |
| codegemma | – | – | 6 | – | 6 | – |
| codellama | – | – | 4 | – | 1 | 5 |
| codestral | – | – | 2 | – | 6 | – |
| deepseek-coder-v2 | – | – | 6 | – | 6 | – |
| deepseek-r1 | – | – | 4 | 2 | – | – |
| deepseek-v3 | 1 | – | 1 | – | – | – |
| gemma3 | – | – | – | – | 6 | – |
| gpt-4.1 | – | – | – | – | – | – |
| gpt-4o | – | – | 2 | – | 1 | – |
| grok-3-beta | – | – | – | – | 1 | – |
| llama3.3 | – | – | 6 | – | 6 | – |
| mistral-large | – | – | 4 | – | – | – |
| olmo2 | 6 | – | – | – | 6 | – |
| phi4 | – | – | 4 | – | 1 | – |
| qwen2.5-coder | – | – | 6 | – | 6 | – |
| qwen3 | – | – | 3 | 2 | – | 2 |

Table 7. Median code quality metrics per model for each experiment for runs that produced fully correct code (score 5): s_{loc} —source lines of code, i.e., non-comment lines of code; c_c —cyclomatic complexity, a measure of independent code paths (lower is better); m_i —maintainability index, a composite software quality measure (0–100, higher is better); $e_t/100$ —type errors per 100 s_{loc} , reflecting type inconsistencies; and, $e_F/100$ — F errors per 100 s_{loc} , reflecting potential code quality issues. ‘All instances’ shows experiment-wide medians of each metric across all models, with n equal to the total number of successful instances.

| Model | Experiment 1 | | | | | | Experiment 2 | | | | | |
|-------------------|--------------|-----------|-------|-------|-----------|-----------|--------------|-----------|-------|-------|-----------|-----------|
| | n | s_{loc} | c_c | m_i | $e_t/100$ | $e_F/100$ | n | s_{loc} | c_c | m_i | $e_t/100$ | $e_F/100$ |
| claude-3.7-sonnet | 2 | 17.5 | 2.0 | 91.7 | 23.03 | 28.78 | 4 | 95.0 | 3.0 | 72.8 | 2.11 | 1.05 |
| codestral | 4 | 13.0 | 2.5 | 90.9 | 0.00 | 30.77 | – | – | – | – | – | – |
| deepseek-r1 | 2 | 19.5 | 4.0 | 60.2 | 22.50 | 17.89 | 4 | 56.0 | 5.0 | 54.9 | 2.73 | 0.89 |
| deepseek-v3 | 4 | 24.0 | 6.0 | 82.7 | 4.17 | 16.67 | 2 | 56.0 | 5.0 | 75.1 | 1.79 | 0.00 |
| gpt-4.1 | 6 | 23.0 | 5.0 | 86.0 | 8.89 | 17.79 | 6 | 62.0 | 5.0 | 73.2 | 1.67 | 0.00 |
| gpt-4o | 4 | 19.0 | 3.0 | 90.4 | 5.26 | 20.53 | 3 | 59.0 | 5.0 | 77.1 | 3.39 | 0.00 |
| grok-3-beta | 6 | 32.0 | 6.0 | 80.6 | 0.00 | 12.51 | 5 | 57.0 | 5.0 | 78.6 | 3.51 | 1.75 |
| mistral-large | 2 | 17.5 | 2.0 | 79.5 | 0.00 | 20.39 | 6 | 57.0 | 5.0 | 72.9 | 0.00 | 0.00 |
| All instances | 30 | 21.0 | 4.0 | 86.0 | 3.60 | 17.91 | 30 | 58.0 | 5.0 | 73.5 | 1.84 | 0.00 |

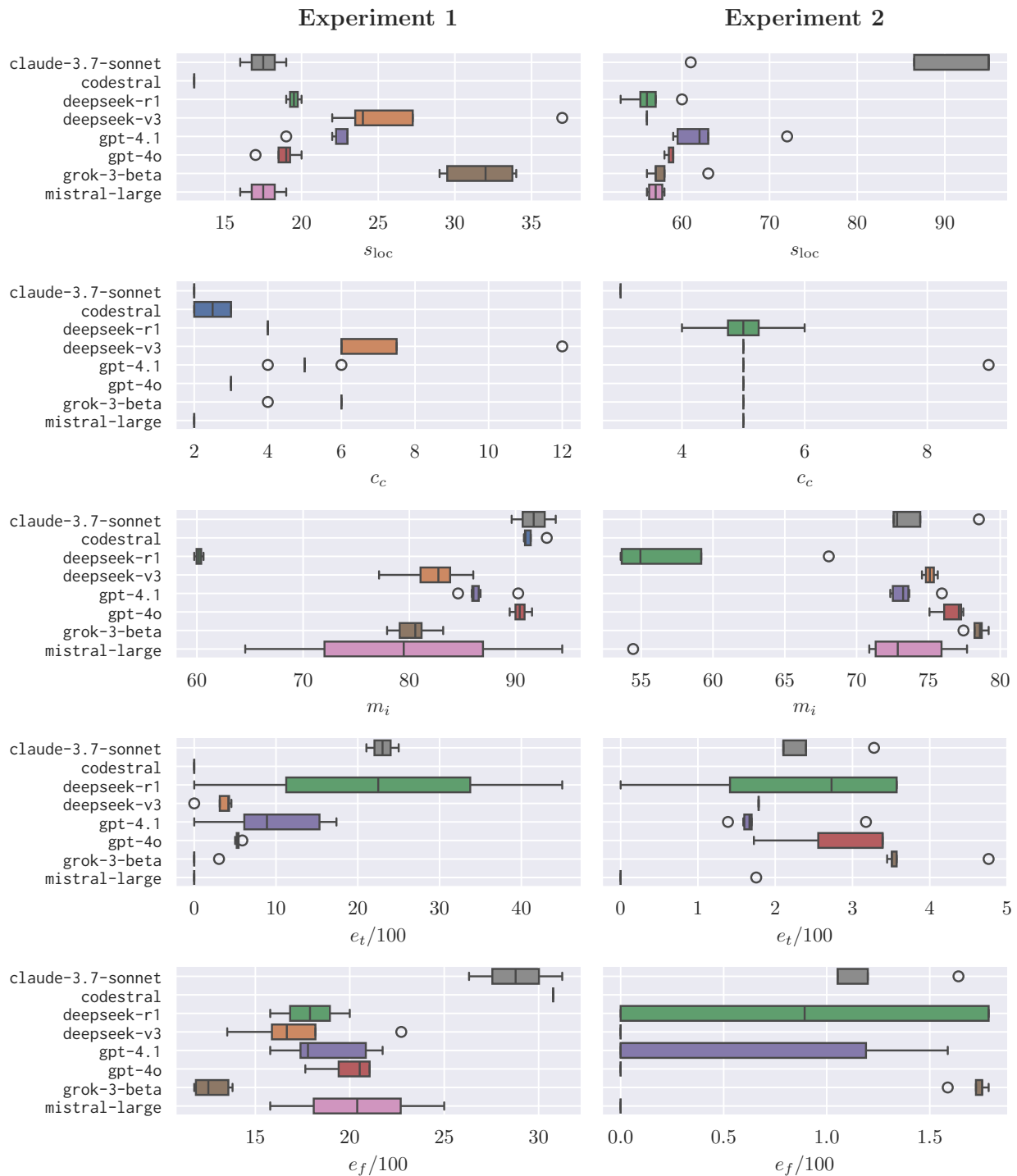


Figure 2. Standard box plots of code quality metrics per model per experiment, restricted to code achieving score = 5. **Left column:** Experiment 1; **Right column:** Experiment 2 (note that axis scales differ substantially between experiments and should not be compared directly). No box plot is shown for Codestral in Experiment 2, since this model did not generate working code. Rows correspond to different metrics: s_{loc} —source lines of code, i.e., non-comment lines of code; c_c —cyclomatic complexity, a measure of independent code paths (lower is better); m_i —maintainability index, a composite software quality measure (0–100, higher is better); $e_t/100$ —type errors per 100 s_{loc} , reflecting type inconsistencies; and $e_f/100$ — F errors per 100 s_{loc} , reflecting potential code quality issues.

Complete experimental data, including all generated outputs, extracted functions, and error logs, are available in the supplementary material [24].

4.1. Experiment 1: ParShift

The first task involved generating Python code using the *ParShift* library to compute the proportion of utterance pairs classified as “Turn Usurping” from CSV conversation files. As indicated in Table 5, the models exhibited a wide range of outcomes. GPT-4.1 and Grok 3 received a perfect score of 5 in every evaluated run, generating executable Python code that returned the expected output in all cases. These two models also achieved a statistically significant higher proportion of perfect scores than other models in 12 out of 16 pairwise comparisons (see Table A2).

Other high-performing models include Codestral, DeepSeek-V3, and GPT-4o, which generated correct solutions in four out of six runs. Claude 3.7 Sonnet, DeepSeek-R1, and Mistral Large achieved score 5 results in two out of six runs each. All large models executed online achieved at least one perfect result. Among offline models, only Codestral reached a perfect score, doing so in 66.7% of runs. Other offline models did not produce any perfect scores in this experiment.

Interestingly, Gemma3 always produced code that executed without errors, but returned an incorrect—although properly formatted—result in every run (score 4). As summarized in Table 6, most execution failures (score 2) were novel API errors. These typically manifested as *KeyErrors* when attempting to access non-existent *ParShift* fields, observed in models such as Llama 3.3, Phi-4, Qwen2.5-Coder, DeepSeek-R1, Mistral Large, and Codestral. In several cases, exceptions surfaced from *Pandas* within *ParShift*’s I/O pipeline (mainly in DeepSeek-Coder-V2), but these were triggered by incorrect use of *ParShift*’s API and were therefore classified as novel API errors. A secondary source of failures involved logic errors, including missing or incorrect imports, as was always the case in Olmo2. There were no direct established API errors in this experiment.

As can be observed in Table 7 and Figure 2, the characteristics of the LLM-generated code that achieved score 5 differed substantially across models in Experiment 1. Regarding size, Codestral-generated code was concise and uniform (median $s_{loc} = 13$, without variance), while code from Grok 3 was comparatively longer (typically > 30 lines). Structural complexity was low for code produced by Claude 3.7 Sonnet and Mistral Large (median $c_c = 2$), while code from Grok 3 and DeepSeek-V3 showed higher branching (median $c_c > 5$), including an outlier for DeepSeek-V3 with $c_c = 12$. Maintainability was high for code generated by Claude 3.7 Sonnet, Codestral, and GPT-4o (median $m_i > 90$), whereas code from DeepSeek-R1 showed lower maintainability (median $m_i \approx 60$). Type checking outcomes varied: several models yielded zero or near-zero $e_t/100$, but code from Claude 3.7 Sonnet and DeepSeek-R1 exhibited higher medians (> 20 type errors per 100 s_{loc}). *F* errors were mostly related with unused inputs and a few inconsequential type redefinitions. Code from Grok 3 had the lowest median (≈ 12.5 per 100 s_{loc}), while code from Claude 3.7 Sonnet and Codestral showed higher medians (roughly 28–31 per 100 s_{loc}). Focusing on models with 100% success rate, GPT-4.1 stands out by always being on the average of other models in all these metrics, and Grok 3 by producing somewhat verbose and complex code, although with very few type and *F* errors.

4.2. Experiment 2: pylugen

The second experiment required the generation of a function to perform a clustering experiment using the *pylugen* and *scikit-learn* libraries and return a *Pandas* data frame with computed clustering metrics. We hypothesize that this task is more complex than that of Experiment 1 due to the higher dimensionality of the data and the need to properly apply multiple clustering algorithms.

The GPT-4.1 and Mistral Large models produced correct, executable code that returned the expected output in every run, obtaining a perfect score of 5 in all cases. The Grok

3 model produced correct output in five out of six runs, while Claude 3.7 Sonnet and DeepSeek-R1 succeeded in four out of six runs. These five models showed a statistically significant higher proportion of top scores relative to most others (details in Table A3).

The remaining online models, GPT-4o and DeepSeek-V3, produced correct solutions in some runs, but none of the smaller, offline models succeeded in generating correct code in any evaluated run. This observation is in line with our hypothesis that Experiment 2 involves greater complexity than Experiment 1.

The Phi-4 model consistently produced code that executed successfully and returned a valid data frame in the correct format, but the results were statistically different from those provided by the reference implementation in all tested runs (see Table A4). This was due to the generated code applying clustering algorithms to the point projections (which *pyclugen* uses as an intermediate step to generate the final points) instead of the final data points themselves, resulting in structurally correct but semantically incorrect outputs.

As shown in Table 6, the most common code execution failures in Experiment 2 (score 2) were established API errors, mainly from incorrect use of *scikit-learn*. A recurring issue in the code generated by several models (CodeGemma, Codestral, DeepSeek Coder-V2, Gemma3, Llama3.3, Olmo2, and Qwen2.5-Coder) was importing GaussianMixture from *sklearn.cluster* instead of the correct *sklearn.mixture* module. Additional problems included passing deprecated/invalid parameters in *scikit-learn* methods (CodeLlama) and accessing unavailable attributes in *scikit-learn* objects (Grok 3), as well as *Pandas* usage incompatible with recent versions (GPT-4o and Phi-4). Novel API errors were less frequent but present, reflecting misuse of *pyclugen*, such as calling the *clugen()* function with missing required positional arguments (CodeLlama), or assuming non-existent functions or attributes (CodeLlama and Qwen3). Finally, a smaller set of logic errors was observed, including missing imports or symbols (Claude 3.7 Sonnet and DeepSeek-R1) and failure to generate the experiment's function with the specified name (Qwen3).

In addition to these execution failures, some models produced outputs with the wrong format (score 3), in particular returning a data frame with the column name "ahc-average" rather than the required "ahc-avg" (DeepSeek-V3 and GPT-4o).

Focusing only on fully functional generated code (score 5), broadly similar profiles were observed between models, with a few interesting differences, as shown in Table 7 and Figure 2. In terms of size, code from Claude 3.7 Sonnet was markedly more verbose (median $s_{loc} = 95$), whereas other models clustered around 56–62 lines. Cyclomatic complexity was stable (median $c_c = 5$) for all models except Claude 3.7 Sonnet (median $c_c = 2$); the combination of higher s_{loc} and lower c_c suggests longer but more linear implementations with fewer branching constructs. An isolated outlier was observed for GPT-4.1 with $c_c = 7$. Maintainability indices were generally similar between models (medians in the 70s), with lower values for code from DeepSeek-R1 (median $m_i \approx 55$) and an occasional low outlier for Mistral Large (below 55). Type checking results were uniformly strong: medians of $e_t/100$ were low for all models, including a zero median for Mistral Large; code from Grok 3 showed the highest median among models (≈ 3.5 per 100 s_{loc}), though still small in absolute terms. F errors per 100 s_{loc} ($e_F/100$) were also minimal, and consistently zero for DeepSeek-V3, GPT-4o, and Mistral Large. The few observed F errors were related to undefined types (with no impact on runtime execution) and some assigned variables that were never used. Code from Grok 3 showed the highest median (≈ 1.75 per 100 s_{loc}). Among models achieving perfect correctness across all runs, code from GPT-4.1 tended again to align with group medians across metrics, whereas code from Mistral Large frequently reached zero $e_t/100$ and $e_F/100$.

4.3. Aggregate Observations

In both experiments, GPT-4.1 was the only model to achieve a perfect score in every evaluated run. Grok 3 also performed strongly, with only one instance of a non-executable output in Experiment 2. At the other end of the scale, Qwen3 was the only model to produce score 1 outputs, failing to generate a Python function at all in some cases. Concurrently, score 4 was commonly assigned to Gemma3 (in Experiment 1) and Phi-4 (in Experiment 2) due to correct function execution but semantically incorrect results.

In both experiments, most outputs clustered around either score 2 (runtime or syntax error) or score 5 (correct result). Scores of 3 (e.g., incorrect return type or number of elements) and 4 (incorrect results with valid output) were less frequent. Within the score 2 outputs, Experiment 1 failures were largely due to novel API errors (mainly from incorrectly accessing *ParShift*'s data structures), whereas Experiment 2 was dominated by established API errors (misuse or hallucination of methods, mainly in *scikit-learn*).

Considering the last row of Table 7, both experiments yielded the same number of successful instances ($n = 30$). Code size increased substantially from Experiment 1 to Experiment 2 (median s_{loc} from 21 to 58). Median cyclomatic complexity rose only modestly (median c_c from 4 to 5) and was notably more stable in Experiment 2, with values tightly concentrated around 5. The maintainability index decreased between experiments (median m_i from 86 to 73.5), consistent with the greater breadth of the second task. Error rates per 100 s_{loc} were lower in Experiment 2: the median $e_t/100$ was roughly halved (1.84 vs. 3.60), and the median $e_F/100$ dropped from approximately 18 to 0.

5. Discussion

The results obtained from the two experiments provide insight into the capabilities and limitations of LLMs in generating Python code using third-party libraries, particularly when dealing with less commonly known APIs. Several important observations can be drawn from the analysis of results.

In Experiment 1, most LLMs encountered difficulties with the *ParShift* library, resulting predominantly from key errors when accessing its internal data structures. These failures can be traced in part to limitations in the clarity of the existing *ParShift* documentation. Indeed, during the preliminary phase of this study, the authors themselves faced challenges in clearly formulating the final prompt due to ambiguities and complexities in the library's documentation. A post-experiment assessment of *ParShift*'s documentation confirms that clearer structure and more explicit examples would likely reduce such errors. Consequently, improvements to *ParShift*'s documentation are planned, motivated directly by findings from this study.

In Experiment 2, the incorrect import of *GaussianMixture* was a common failure point for many models, reflecting a broader class of import and attribute errors. This issue highlights a fundamental characteristic of LLMs: their reliance on statistical patterns in training data to predict plausible tokens rather than verifying correctness through explicit validation. Since *GaussianMixture* appears contextually similar to algorithms within the *sklearn.cluster* module, models often incorrectly associated it with this module. This behavior resembles human guesswork based on contextual association rather than verified knowledge, reflecting a known limitation of current generative models [65]. In contrast, models such as GPT-4.1 and Mistral Large consistently avoided this mistake, suggesting that more rigorous fine-tuning with validated code samples, stronger internal consistency mechanisms, or filtering of incorrect token sequences for critical libraries such as *scikit-learn* may mitigate such errors.

Code quality results demonstrated that top performers are not identical once maintainability and static issues are considered. GPT-4.1 tended to sit near group medians across

metrics—a “steady default” that consistently produced working code, albeit not always optimizing for maintainability or strict style conformance. Its counterpart, GPT-4o, was also relatively average across metrics but showed higher maintainability when it produced correct outputs. Grok 3 was notably verbose in Experiment 1 yet exhibited very few type and F errors; conversely, in Experiment 2 it was among the less verbose models while showing comparatively higher static error rates among successful runs. A plausible explanation is that the simpler task in Experiment 1 involved a more idiosyncratic API (*ParShift*) that several models found harder to parse, whereas the more complex Experiment 2 may have benefited from clearer API affordances (e.g., *pyclugen* documentation), allowing some models to adapt structure to the increased task complexity, with only a modest reduction in maintainability. Importantly, Grok 3 almost always produced working code (score 5) across both tasks, trailing only GPT-4.1 in overall reliability. Claude 3.7 Sonnet presented the lowest and most stable c_c between experiments. It achieved this with relatively few s_{loc} in Experiment 1 but with substantially higher s_{loc} in Experiment 2 (approximately 95 versus 56–62 for peers), yielding longer, more linear code that may be easier to read, while maintaining reasonable m_i and a relatively higher static error rate in Experiment 1. Codestral, although only successful in Experiment 1, produced concise code with low c_c , high m_i , and zero type errors. However, it exhibited the highest median $e_F/100$, largely attributable to unused (often type-related) imports—suggesting a conservative import strategy or remnants of partially pruned templates. DeepSeek-R1 showed consistently lower m_i relative to peers in both experiments, indicating a higher refactoring burden even when outputs were correct. Its non-reasoning counterpart DeepSeek-V3 generated code with higher maintainability and fewer static issues, indicating that R1 did not surpass V3 in this study. Finally, while Mistral Large struggled in Experiment 1 (both in success rate and some quality indicators), it excelled in Experiment 2, producing correct code in all runs with uniformly strong quality metrics (e.g., zero type and F errors).

With the exception of Codestral, models that produced working code in one experiment also did so in the other. Across successful runs, Experiment 2 generally yielded longer code (higher s_{loc}), only slightly higher c_c , and a modest decrease in m_i —patterns consistent with increased task complexity. Interestingly, type and F errors were markedly fewer in Experiment 2 among correct solutions, which, together with the error taxonomy, strengthens the hypothesis that *ParShift*’s API in Experiment 1 posed interpretability challenges for several models. This observation complements the earlier conclusion regarding the value of improved documentation for *ParShift*.

During prompt development for Experiment 2, a separate issue arose when some LLMs provided `np.int64` types for the seed parameter in the `clugen()` function from the *pyclugen* library. Although conceptually correct, this caused runtime crashes due to type incompatibility with the previous implementation of *pyclugen*. To address this, *pyclugen* was updated prior to the experiment to accept any integer-compatible type, ensuring robustness to typical coding practices. This experience highlights a broader utility of employing LLMs for testing assumptions and identifying edge cases that might not surface even under comprehensive unit testing. In this particular instance, despite 100% unit test coverage, the practical use of LLM-generated code facilitated discovery and rectification of a subtle but impactful type-checking oversight.

Contrasting with the findings from a previous study [4], where smaller offline models, particularly Phi-4 and Llama3.3, performed comparably to larger models, the current results revealed significantly lower performance from these smaller models in generating code for computational experiments. The exception among offline models was Codestral, which achieved a few instances of correct results in Experiment 1. The diminished performance of Phi-4 and Llama3.3 compared to previous findings indicates that the complexity and

novelty of the APIs used in the present study significantly affect the effectiveness of these models.

Among all evaluated models, GPT-4.1 demonstrated a clear and consistent advantage, achieving perfect scores in both experiments and showing a strong capacity to interpret unfamiliar APIs and return syntactically and semantically correct code. This suggests that recent improvements in the GPT-4 family, such as improved reasoning, instruction following, and error correction mechanisms, translate directly into more reliable code synthesis, especially for complex or lesser-known libraries. Both Grok 3 and Mistral Large also performed at a high level, ranking among the few models able to solve both tasks with high consistency. These results indicate that, at present, only a handful of top-tier models can reliably perform advanced code generation tasks that involve integrating novel APIs in end-to-end experiment pipelines.

It is also relevant that DeepSeek-R1, a model explicitly marketed for improved reasoning, did not clearly outperform its generalist counterpart, DeepSeek-V3, in these scenarios, especially in Experiment 1. Both models achieved similar levels of performance and operated in a comparable range to Claude 3.7 Sonnet and GPT-4o, which are also positioned as high-capacity but general-purpose LLMs. This outcome suggests that, despite recent advances in LLM architectures and training strategies for specialized reasoning, practical gains in challenging code synthesis tasks may remain limited unless accompanied by targeted fine-tuning and extensive exposure to domain-specific code samples. The high variance observed even among top models further underlines the need for ongoing evaluation and benchmarking as LLM technology continues to evolve.

For settings where multiple models achieve similar top-score rates, code quality metrics can guide selection according to context: models with lower $e_t/100$ and $e_F/100$ may be preferable when static typing and automated code checks (continuous integration linting gates that block errors before code is merged) are critical. Higher s_{loc} with lower c_c may be acceptable (or even desirable) when readability by less experienced developers is prioritized; and “median-leaning” profiles can be advantageous when consistency across tasks and codebases is valued.

Overall, these findings highlight the importance of rigorous API documentation and model-specific fine-tuning when employing LLMs for automated generation of computational experiment code. Additionally, these results demonstrate the utility of such automated code generation tasks not only for evaluating model capabilities but also for exposing assumptions and deficiencies within libraries themselves, thus contributing positively to their further development and robustness. The integration of code quality metrics highlights that, beyond attaining functional correctness, the sustainability and maintainability of LLM-generated code should be factored into model selection and deployment decisions.

6. Limitations

While the present study provides important insights into the capabilities of current LLMs for the automated synthesis of computational experiment code using less common Python libraries, several limitations should be acknowledged.

First, the evaluation was limited to two specific computational experiments involving the *ParShift* and *pyclugen* libraries. While these libraries are representative of real-world, lesser-known scientific software, the generalizability of findings to other domains or libraries cannot be guaranteed. It is possible that the observed failure patterns or success rates may differ when different APIs, programming languages, or experimental paradigms are used. Additionally, because the prompts included real docstrings from the target libraries, we cannot fully exclude the possibility that these materials were part of some LLMs’ pretraining corpora, potentially influencing their performance.

Second, all models were evaluated under a zero-shot prompt setting, which, while arguably reflecting realistic scenarios faced by many users, does not capture the potential benefits of in-context learning, prompt tuning, or multi-turn interactions. Many recent studies show that LLMs can substantially improve code quality when provided with examples, clarifying questions, or iterative feedback [1–3]. The single-pass prompt structure employed here, while controlled, does not fully exploit these capabilities.

Third, the evaluation infrastructure did not simulate the availability of external resources (e.g., internet search, code validation tools, or documentation lookup) that are increasingly integrated into commercial LLMs and coding assistants. All models were evaluated using only their pretrained knowledge. Therefore, the results may underestimate the real-world performance of models when used in modern coding environments.

Fourth, model variability was assessed by running each prompt with several seeds and temperature slightly above zero to guarantee some randomness; however, this does not exhaustively capture the full space of model stochasticity, especially in models for which temperature or seed control was not available or did not function as expected. Thus, some sources of non-determinism and rare failure modes may not have been observed.

Finally, the study included a limited set of models, selected for breadth and relevance but necessarily omitting some recent or proprietary releases due to practical access constraints. The results may not reflect the capabilities of models released after the completion of this study, nor those of specialized, commercial, or domain-adapted variants.

These limitations point to important avenues for future research, such as expanding the diversity of tested libraries and tasks, incorporating iterative and in-context prompting, and benchmarking LLMs in environments augmented with external tools and resources.

7. Conclusions

This study systematically evaluated the ability of state-of-the-art LLMs to generate functional Python code for realistic computational experiments involving less widely known scientific libraries. By presenting models with structured, zero-shot prompts using the *ParShift* and *pyclugen* libraries, the experiments probed both code synthesis accuracy and robustness when integrating novel APIs.

The results reveal that only a small subset of current models, particularly GPT-4.1, Grok 3, and Mistral Large, consistently produced correct and executable code across both experiments. These models demonstrated strong capabilities not only in generating syntactically correct code, but also in interpreting prompt requirements and accurately handling library-specific details. Among runs that achieved perfect functional scores, analysis of code quality indicators (source lines of code, cyclomatic complexity, maintainability index, and static error rates) further differentiated models and exposed practical trade-offs in readability and maintainability, highlighting the value of assessing quality beyond pass/fail correctness. Interestingly, the specialized reasoning model DeepSeek-R1 did not clearly surpass the generalist DeepSeek-V3, and both performed similarly to Claude 3.7 Sonnet and GPT-4o. Conversely, the majority of smaller or open weight offline models struggled, often failing at basic integration steps or producing code with semantic errors.

Beyond model assessment, this study also highlighted the role of LLM-based experiments in uncovering both documentation gaps (as seen with *ParShift*) and latent software bugs (as demonstrated by the type handling issue in *pyclugen*), suggesting the mutual benefit of LLM evaluation for both model developers and scientific software authors.

While the results are encouraging regarding the best-performing LLMs, the observed limitations and failure cases indicate that reliably automating scientific experiment pipelines with LLMs remains challenging—particularly for models lacking targeted fine-tuning or advanced context handling. To broaden the applicability of LLMs in specialized research

domains, continued progress is needed not only in model training and prompt engineering, but also in improving the documentation of third-party libraries. Clearer and more complete API documentation can help LLMs interpret unfamiliar tools more accurately, reducing errors and improving overall performance.

Future work should extend these benchmarks to a wider range of libraries, languages, and interaction modes, as well as to real-world use cases requiring code maintenance, interpretability, and safe deployment. As LLMs continue to advance, systematic and transparent evaluation will remain critical for their responsible and effective integration into scientific research workflows.

Author Contributions: Conceptualization, N.F., D.F., C.M.F. and J.P.M.-C.; methodology, N.F. and D.F.; software, N.F., D.F. and B.D.F.-S.; validation, N.F., D.F., C.M.F., B.D.F.-S. and J.P.M.-C.; formal analysis, N.F.; investigation, N.F. and C.M.F.; resources, N.F. and J.P.M.-C.; data curation, N.F.; writing—original draft preparation, N.F. and C.M.F.; writing—review and editing, N.F., D.F., C.M.F., B.D.F.-S. and J.P.M.-C.; visualization, N.F. and B.D.F.-S. All authors have read and agreed to the published version of the manuscript.

Funding: This research was partially funded by the Fundação para a Ciência e a Tecnologia (FCT, <https://ror.org/00snfq58>) under Grants UIDB/04111/2020, UIDB/00066/2020, UIDB/00408/2020, UID/00408/2025, and CEECINST/00002/2021/CP2788/CT0001, as well as by the Instituto Lusófono de Investigação e Desenvolvimento (ILIND), Portugal, under Project COFAC/ILIND/COPELABS-/1/2024.

Data Availability Statement: The data generated by this study and its respective analysis are available at <https://doi.org/10.5281/zenodo.16582736> under the CC-BY license.

Conflicts of Interest: The authors declare no conflicts of interest.

Abbreviations

The following abbreviations are used in this manuscript:

| | |
|-----|-----------------------------------|
| API | Application Programming Interface |
| FDR | False Discovery Rate |
| LLM | Large Language Model |
| MoE | Mixture-of-Experts |
| PCA | Principal Component Analysis |
| RL | Reinforcement Learning |

Appendix A

This appendix contains four supplementary tables. Table A1 provides the versions of the software used for running the LLM-generated code and performing the data analysis. The remaining tables provide detailed statistical analyses of the experimental results. Specifically, Table A2 presents pairwise p -values for top-score proportions in Experiment 1, Table A3 presents the corresponding results for Experiment 2, and Table A4 provides a comparison of model outputs for Experiment 2 against the reference baseline.

| Software | Execution | Analysis |
|---------------------|-----------|----------|
| Python | 3.10.12 | 3.12.3 |
| <i>mypy</i> | — | 1.17.1 |
| <i>NumPy</i> | 1.26.4 | 2.3.2 |
| <i>Pandas</i> | 2.2.3 | 2.3.1 |
| <i>ParShift</i> | 1.0.1 | — |
| <i>pyclugen</i> | 1.1.4 | — |
| <i>Radon</i> | — | 6.0.1 |
| <i>Ruff</i> | — | 0.12.9 |
| <i>scikit-learn</i> | 1.4.2 | — |
| <i>SciPy</i> | — | 1.16.1 |
| <i>statsmodels</i> | — | 0.14.5 |
| R | — | 4.5.1 |
| <i>micompr</i> | — | 1.2.0 |
| <i>rmarkdown</i> | — | 2.29 |

[illegible]

Table A3. Experiment 2: p -values from Fisher’s exact test [50] comparing the proportion of top scores between models (rows vs. columns). The null hypothesis H_0 states that the model in the row does *not* have a significantly higher proportion of top scores than the model in the column. p -values below the 0.05 significance threshold (highlighted in light grey) indicate rejection of H_0 , suggesting that the row model *does* have a significantly higher proportion of top scores. Multiple comparisons are corrected per row using the Benjamini–Hochberg procedure [51] over 16 comparisons.

| Model | claude-3.7-sonnet | codegemma | codellama | codestral | deepseek-coder-v2 | deepseek-r1 | deepseek-v3 | gemma3 | gpt-4.1 | gpt-4o | grok-3-beta | llama3.3 | mistral-large | olmo2 | phi4 | qwen2.5-coder | qwen3 |
|-------------------|-------------------|-----------|-----------|-----------|-------------------|-------------|-------------|--------|---------|--------|-------------|----------|---------------|-------|-------|---------------|-------|
| claude-3.7-sonnet | — | 0.048 | 0.048 | 0.048 | 0.048 | 0.895 | 0.412 | 0.048 | 1.000 | 0.667 | 1.000 | 0.048 | 1.000 | 0.048 | 0.048 | 0.048 | 0.048 |
| codegemma | 1.000 | — | 1.000 | 1.000 | 1.000 | 1.000 | 1.000 | 1.000 | 1.000 | 1.000 | 1.000 | 1.000 | 1.000 | 1.000 | 1.000 | 1.000 | 1.000 |
| codellama | 1.000 | 1.000 | — | 1.000 | 1.000 | 1.000 | 1.000 | 1.000 | 1.000 | 1.000 | 1.000 | 1.000 | 1.000 | 1.000 | 1.000 | 1.000 | 1.000 |
| codestral | 1.000 | 1.000 | 1.000 | — | 1.000 | 1.000 | 1.000 | 1.000 | 1.000 | 1.000 | 1.000 | 1.000 | 1.000 | 1.000 | 1.000 | 1.000 | 1.000 |
| deepseek-coder-v2 | 1.000 | 1.000 | 1.000 | 1.000 | — | 1.000 | 1.000 | 1.000 | 1.000 | 1.000 | 1.000 | 1.000 | 1.000 | 1.000 | 1.000 | 1.000 | 1.000 |
| deepseek-r1 | 0.895 | 0.048 | 0.048 | 0.048 | 0.048 | — | 0.412 | 0.048 | 1.000 | 0.667 | 1.000 | 0.048 | 1.000 | 0.048 | 0.048 | 0.048 | 0.048 |
| deepseek-v3 | 1.000 | 0.364 | 0.364 | 0.364 | 0.364 | 1.000 | — | 0.364 | 1.000 | 1.000 | 1.000 | 0.364 | 1.000 | 0.364 | 0.364 | 0.364 | 0.364 |
| gemma3 | 1.000 | 1.000 | 1.000 | 1.000 | 1.000 | 1.000 | 1.000 | — | 1.000 | 1.000 | 1.000 | 1.000 | 1.000 | 1.000 | 1.000 | 1.000 | 1.000 |
| gpt-4.1 | 0.260 | <0.01 | <0.01 | <0.01 | <0.01 | 0.260 | 0.044 | <0.01 | — | 0.121 | 0.533 | <0.01 | 1.000 | <0.01 | <0.01 | <0.01 | <0.01 |
| gpt-4o | 1.000 | 0.145 | 0.145 | 0.145 | 0.145 | 1.000 | 0.727 | 0.145 | 1.000 | — | 1.000 | 0.145 | 1.000 | 0.145 | 0.145 | 0.145 | 0.145 |
| grok-3-beta | 0.571 | 0.012 | 0.012 | 0.012 | 0.012 | 0.571 | 0.176 | 0.012 | 1.000 | 0.364 | — | 0.012 | 1.000 | 0.012 | 0.012 | 0.012 | 0.012 |
| llama3.3 | 1.000 | 1.000 | 1.000 | 1.000 | 1.000 | 1.000 | 1.000 | 1.000 | 1.000 | 1.000 | 1.000 | — | 1.000 | 1.000 | 1.000 | 1.000 | 1.000 |
| mistral-large | 0.260 | <0.01 | <0.01 | <0.01 | <0.01 | 0.260 | 0.044 | <0.01 | 1.000 | 0.121 | 0.533 | <0.01 | — | <0.01 | <0.01 | <0.01 | <0.01 |
| olmo2 | 1.000 | 1.000 | 1.000 | 1.000 | 1.000 | 1.000 | 1.000 | 1.000 | 1.000 | 1.000 | 1.000 | 1.000 | 1.000 | — | 1.000 | 1.000 | 1.000 |
| phi4 | 1.000 | 1.000 | 1.000 | 1.000 | 1.000 | 1.000 | 1.000 | 1.000 | 1.000 | 1.000 | 1.000 | 1.000 | 1.000 | 1.000 | — | 1.000 | 1.000 |
| qwen2.5-coder | 1.000 | 1.000 | 1.000 | 1.000 | 1.000 | 1.000 | 1.000 | 1.000 | 1.000 | 1.000 | 1.000 | 1.000 | 1.000 | 1.000 | 1.000 | — | 1.000 |
| qwen3 | 1.000 | 1.000 | 1.000 | 1.000 | 1.000 | 1.000 | 1.000 | 1.000 | 1.000 | 1.000 | 1.000 | 1.000 | 1.000 | 1.000 | 1.000 | 1.000 | — |

Table A4. Statistical comparisons of Experiment 2 baseline versus outputs from LLM-generated functions. The table shows the adjusted p -values (p -val.) resulting from the Mann-Whitney U test [53] applied to the first principal component, obtained via PCA, of the concatenated algorithm outputs (V -measure per average cluster length, centered and scaled). The p -values are adjusted for FDR per-model (across seeds) using the Benjamini-Hochberg procedure [51]. Significant results (at $\alpha = 0.05$) indicating statistical differences between baseline and LLM-generated outputs are highlighted with a light grey background. A significant difference implies assigning a score of 4 (statistically different), whereas non-significant results indicate assigning a score of 5 (no statistically significant difference detected). The ‘sp’ columns contain corresponding stylized score plots displaying the first two principal components to assist in evaluating statistical similarities (red dots correspond to the baseline, black dots to the LLM-generated function output). An $\times$ indicates cases where the generated function was either not runnable (scores 1–2) or did not return a properly structured output (score 3); hence, no comparison was possible for that seed. Models that only generated such functions are not shown. Identical score plots across different seeds indicate that the respective model produced functions with identical algorithmic processes and internal seeding behavior for two or more seeds.

| Model | Run/Seed | | | | | | | | | | | |
|-------------------|-----------|---|-----------|---|-----------|---|-----------|---|-----------|----------|-----------|----------|
| | 1/0862 | | 2/1924 | | 3/2090 | | 4/4745 | | 5/5417 | | 6/7259 | |
| | p -Val. | sp | p -Val. | sp | p -Val. | sp | p -Val. | sp | p -Val. | sp | p -Val. | sp |
| claude-3.7-sonnet | 0.398 |  | 0.398 |  | 0.398 |  | 0.398 |  | $\times$ | $\times$ | $\times$ | $\times$ |

Table A4. Cont.

| Model | Run/Seed | | | | | | | | | | | |
|---------------|----------------|----|----------------|----|----------------|----|----------------|----|----------------|----|----------------|----|
| | 1/0862 | | 2/1924 | | 3/2090 | | 4/4745 | | 5/5417 | | 6/7259 | |
| | <i>p</i> -Val. | sp | <i>p</i> -Val. | sp | <i>p</i> -Val. | sp | <i>p</i> -Val. | sp | <i>p</i> -Val. | sp | <i>p</i> -Val. | sp |
| deepseek-r1 | 0.256 | | × | × | 0.256 | | 0.256 | | 0.307 | | × | × |
| deepseek-v3 | × | × | 0.182 | | × | × | × | × | 0.182 | | × | × |
| gpt-4.1 | 0.224 | | 0.439 | | 0.224 | | 0.395 | | 0.633 | | 0.633 | |
| gpt-4o | 0.174 | | 0.174 | | × | × | × | × | × | × | 0.366 | |
| grok-3-beta | 0.336 | | 0.336 | | 0.398 | | 0.398 | | 0.398 | | × | × |
| mistral-large | 0.582 | | 0.582 | | 0.296 | | 0.296 | | 0.296 | | 0.296 | |
| phi4 | <0.01 | | <0.01 | | <0.01 | | <0.01 | | <0.01 | | × | × |

References

- Chen, M.; Tworek, J.; Jun, H.; Yuan, Q.; Pinto, H.P.D.O.; Kaplan, J.; Edwards, H.; Burda, Y.; Joseph, N.; Brockman, G.; et al. Evaluating large language models trained on code. *arXiv* **2021**, arXiv:2107.03374. [\[CrossRef\]](#)
- Jain, N.; Vaidyanath, S.; Iyer, A.; Natarajan, N.; Parthasarathy, S.; Rajamani, S.; Sharma, R. Jigsaw: Large language models meet program synthesis. In Proceedings of the ICSE '22: 44th International Conference on Software Engineering, Pittsburgh, PA, USA, 21–29 May 2022; Association for Computing Machinery: New York, NY, USA, 2022; pp. 1219–1231. [\[CrossRef\]](#)
- Gu, X.; Chen, M.; Lin, Y.; Hu, Y.; Zhang, H.; Wan, C.; Wei, Z.; Xu, Y.; Wang, J. On the effectiveness of large language models in domain-specific code generation. *ACM Trans. Softw. Eng. Methodol.* **2024**, *34*, 78. [\[CrossRef\]](#)
- Fernandes, D.; Matos-Carvalho, J.P.; Fernandes, C.M.; Fachada, N. DeepSeek-V3, GPT-4, Phi-4, and LLaMA-3.3 generate correct code for LoRaWAN-related engineering tasks. *Electronics* **2025**, *14*, 1428. [\[CrossRef\]](#)
- O'Donoghue, O.; Shtedritski, A.; Ginger, J.; Abboud, R.; Ghareeb, A.E.; Booth, J.; Rodrigues, S.G. BioPlanner: Automatic Evaluation of LLMs on Protocol Planning in Biology. *arXiv* **2023**. [\[CrossRef\]](#)
- Hong, S.; Lin, Y.; Liu, B.; Liu, B.; Wu, B.; Zhang, C.; Wei, C.; Li, D.; Chen, J.; Zhang, J.; et al. Data Interpreter: An LLM Agent for Data Science. *arXiv* **2024**, arXiv:2402.18679. [\[CrossRef\]](#)
- Charness, G.; Jabarian, B.; List, J.A. The next generation of experimental research with LLMs. *Nat. Hum. Behav.* **2025**, *9*, 345–357. [\[CrossRef\]](#)
- Amatriain, X. Prompt design and engineering: Introduction and advanced methods. *arXiv* **2024**, arXiv:2401.14423. [\[CrossRef\]](#)
- Spiess, C.; Gros, D.; Pai, K.S.; Pradel, M.; Rabin, M.R.I.; Alipour, A.; Jha, S.; Devanbu, P.; Ahmed, T. Calibration and correctness of language models for code. In Proceedings of the 2025 IEEE/ACM 47th International Conference on Software Engineering (ICSE), Ottawa, ON, Canada, 26 April–5 May 2025; pp. 540–552. [\[CrossRef\]](#)
- Ferreira-Saraiva, B.D.; Matos-Carvalho, J.P.; Fachada, N.; Pita, M. ParShift: A Python package to study order and differentiation in group conversations. *SoftwareX* **2023**, *24*, 101554. [\[CrossRef\]](#)
- Fachada, N.; de Andrade, D. Generating multidimensional clusters with support lines. *Knowl.-Based Syst.* **2023**, *277*, 110836. [\[CrossRef\]](#)
- Wei, J.; Wang, X.; Schuurmans, D.; Bosma, M.; Ichter, B.; Xia, F.; Chi, E.H.; Le, Q.V.; Zhou, D. Chain-of-thought prompting elicits reasoning in large language models. In Proceedings of the NIPS '22: 36th International Conference on Neural Information Processing Systems, New Orleans, LA, USA, 28 November–9 December 2022.
- Xiong, G.; Xie, E.; Shariatmadari, A.H.; Guo, S.; Bekiranov, S.; Zhang, A. Improving Scientific Hypothesis Generation with Knowledge Grounded Large Language Models. *arXiv* **2024**, arXiv:2411.02382. [\[CrossRef\]](#)

14. Pu, Y.; Lin, T.; Chen, H. PiFlow: Principle-aware Scientific Discovery with Multi-Agent Collaboration. *arXiv* **2025**, arXiv:2505.15047.
15. Jin, Y.; Zhao, Q.; Wang, Y.; Chen, H.; Zhu, K.; Xiao, Y.; Wang, J. AgentReview: Exploring Peer Review Dynamics with LLM Agents. *arXiv* **2024**, arXiv:2406.12708. [\[CrossRef\]](#)
16. Agarwal, S.; Sahu, G.; Puri, A.; Laradji, I.H.; Dvijotham, K.D.; Stanley, J.; Charlin, L.; Pal, C. LitLLM: A Toolkit for Scientific Literature Review. *arXiv* **2025**, arXiv:2402.01788.
17. Luo, Z.; Yang, Z.; Xu, Z.; Yang, W.; Du, X. LLM4SR: A Survey on Large Language Models for Scientific Research. *arXiv* **2025**, arXiv:2501.04306. [\[CrossRef\]](#)
18. Liang, J.; Huang, W.; Xia, F.; Xu, P.; Hausman, K.; Ichter, B.; Florence, P.; Zeng, A. Code as Policies: Language Model Programs for Embodied Control. In Proceedings of the 2023 IEEE International Conference on Robotics and Automation (ICRA), London, UK, 29 May–2 June 2023; pp. 9493–9500. [\[CrossRef\]](#)
19. Luo, H.; Wu, J.; Liu, J.; Antwi-Afari, M.F. Large language model-based code generation for the control of construction assembly robots: A hierarchical generation approach. *Dev. Built Environ.* **2024**, *19*, 100488. [\[CrossRef\]](#)
20. Yao, S.; Zhao, J.; Yu, D.; Du, N.; Shafran, I.; Narasimhan, K.; Cao, Y. ReAct: Synergizing Reasoning and Acting in Language Models. *arXiv* **2023**, arXiv:2210.03629. [\[CrossRef\]](#)
21. Yang, H.; Yue, S.; He, Y. Auto-GPT for Online Decision Making: Benchmarks and Additional Opinions. *arXiv* **2023**, arXiv:2306.02224. [\[CrossRef\]](#)
22. Brown, T.; Mann, B.; Ryder, N.; Subbiah, M.; Kaplan, J.D.; Dhariwal, P.; Neelakantan, A.; Shyam, P.; Sastry, G.; Askell, A.; et al. Language models are few-shot learners. In Proceedings of the NeurIPS 2020: 34th Conference on Neural Information Processing Systems, Online, 6–12 December 2020; Volume 33, pp. 1877–1901.
23. Vemprala, S.; Bonatti, R.; Buckner, A.; Kapoor, A. ChatGPT for robotics: Design principles and model abilities. *arXiv* **2023**, arXiv:2306.17582. [\[CrossRef\]](#)
24. Fachada, N.; Fernandes, D.; Fernandes, C.M.; Ferreira-Saraiva, B.; Matos-Carvalho, J. Supplementary material for “GPT-4.1 Sets the Standard in Automated Experiment Design Using Novel Python Libraries”. *Zenodo* **2025**. [\[CrossRef\]](#)
25. Pedregosa, F.; Varoquaux, G.; Gramfort, A.; Michel, V.; Thirion, B.; Grisel, O.; Blondel, M.; Prettenhofer, P.; Weiss, R.; Dubourg, V.; et al. Scikit-learn: Machine Learning in Python. *J. Mach. Learn. Res.* **2011**, *12*, 2825–2830.
26. Rosenberg, A.; Hirschberg, J. V-measure: A conditional entropy-based external cluster evaluation measure. In Proceedings of the 2007 Joint Conference on Empirical Methods in Natural Language Processing and Computational Natural Language Learning (EMNLP-CoNLL), Prague, Czech Republic, 28–30 June 2007; pp. 410–420.
27. Anthropic Team. *Claude 3.7 Sonnet System Card*; Technical Report; Anthropic PBC: San Francisco, CA, USA, 2025.
28. Zhu, Q.; Guo, D.; Shao, Z.; Yang, D.; Wang, P.; Xu, R.; Wu, Y.; Li, Y.; Gao, H.; Ma, S.; et al. Deepseek-coder-v2: Breaking the barrier of closed-source models in code intelligence. *arXiv* **2024**, arXiv:2406.11931. [\[CrossRef\]](#)
29. Guo, D.; Yang, D.; Zhang, H.; Song, J.; Zhang, R.; Xu, R.; Zhu, Q.; Ma, S.; Wang, P.; Bi, X.; et al. DeepSeek-R1: Incentivizing reasoning capability in llms via reinforcement learning. *arXiv* **2025**, arXiv:2501.12948. [\[CrossRef\]](#)
30. DeepSeek-AI; Liu, A.; Feng, B.; Xue, B.; Wang, B.; Wu, B.; Lu, C.; Zhao, C.; Deng, C.; Zhang, C.; et al. DeepSeek-V3 Technical Report. *arXiv* **2024**, arXiv:2412.19437. [\[CrossRef\]](#)
31. Zhao, H.; Hui, J.; Howland, J.; Nguyen, N.; Zuo, S.; Hu, A.; Choquette-Choo, C.A.; Shen, J.; Kelley, J.; Bansal, K.; et al. Codegemma: Open code models based on gemma. *arXiv* **2024**, arXiv:2406.11409. [\[CrossRef\]](#)
32. Kamath, A.; Ferret, J.; Pathak, S.; Vieillard, N.; Merhej, R.; Perrin, S.; Matejovicova, T.; Ramé, A.; Rivière, M.; Rouillard, L.; et al. Gemma 3 technical report. *arXiv* **2025**, arXiv:2503.19786. [\[CrossRef\]](#)
33. Hurst, A.; Lerer, A.; Goucher, A.P.; Perelman, A.; Ramesh, A.; Clark, A.; Ostrow, A.; Welihinda, A.; Hayes, A.; Radford, A.; et al. GPT-4o system card. *arXiv* **2024**, arXiv:2410.21276. [\[CrossRef\]](#)
34. OpenAI. Introducing GPT-4.1 Model Family. 2025. Available online: <https://openai.com/index/gpt-4-1/> (accessed on 9 July 2025).
35. xAI. Grok 3 Beta—The Age of Reasoning Agents. 2025. Available online: <https://x.ai/news/grok-3> (accessed on 9 July 2025).
36. Grattafiori, A.; Dubey, A.; Jauhri, A.; Pandey, A.; Kadian, A.; Al-Dahle, A.; Letman, A.; Mathur, A.; Schelten, A.; Vaughan, A.; et al. The Llama 3 Herd of Models. *arXiv* **2024**, arXiv:2407.21783. [\[CrossRef\]](#)
37. Roziere, B.; Gehring, J.; Gloeckle, F.; Sootla, S.; Gat, I.; Tan, X.E.; Adi, Y.; Liu, J.; Sauvestre, R.; Remez, T.; et al. Code llama: Open foundation models for code. *arXiv* **2023**, arXiv:2308.12950. [\[CrossRef\]](#)
38. Mistral AI Team. Codestral. 2024. Available online: <https://mistral.ai/news/codestral> (accessed on 9 July 2025).
39. Mistral AI Team. Large Enough. 2024. Available online: <https://mistral.ai/news/mistral-large-2407> (accessed on 9 July 2025).
40. Team OLMo; Walsh, P.; Soldaini, L.; Groeneveld, D.; Lo, K.; Arora, S.; Bhagia, A.; Gu, Y.; Huang, S.; Jordan, M.; et al. 2 OLMo 2 Furious. *arXiv* **2024**, arXiv:2501.00656. [\[CrossRef\]](#)
41. Abdin, M.; Aneja, J.; Behl, H.; Bubeck, S.; Eldan, R.; Gunasekar, S.; Harrison, M.; Hewett, R.J.; Javaheripi, M.; Kauffmann, P.; et al. Phi-4 technical report. *arXiv* **2024**, arXiv:2412.08905. [\[CrossRef\]](#)

42. Hui, B.; Yang, J.; Cui, Z.; Yang, J.; Liu, D.; Zhang, L.; Liu, T.; Zhang, J.; Yu, B.; Lu, K.; et al. Qwen2.5-Coder Technical Report. *arXiv* **2025**, arXiv:2409.12186. [\[CrossRef\]](#)
43. Yang, A.; Li, A.; Yang, B.; Zhang, B.; Hui, B.; Zheng, B.; Yu, B.; Gao, C.; Huang, C.; Lv, C.; et al. Qwen3 technical report. *arXiv* **2025**, arXiv:2505.09388. [\[CrossRef\]](#)
44. Semmelrock, H.; Ross-Hellauer, T.; Kopeinik, S.; Theiler, D.; Haberl, A.; Thalmann, S.; Kowald, D. Reproducibility in machine-learning-based research: Overview, barriers, and drivers. *AI Mag.* **2025**, *46*, e70002. [\[CrossRef\]](#)
45. Zhou, H.; Savova, G.; Wang, L. Assessing the Macro and Micro Effects of Random Seeds on Fine-Tuning Large Language Models. *arXiv* **2025**, arXiv:2503.07329. [\[CrossRef\]](#)
46. Renze, M.; Guven, E. The effect of sampling temperature on problem solving in large language models. In *Proceedings of the Findings of the Association for Computational Linguistics: EMNLP 2024, Miami, FL, USA, 12–16 November 2024*; Al-Onaizan, Y., Bansal, M., Chen, Y.N., Eds.; Association for Computational Linguistics: Stroudsburg, PA, USA, 2024; pp. 7346–7356. [\[CrossRef\]](#)
47. DeepSeek, Inc. Reasoning Model (Deepseek-Reasoner) | DeepSeek API Docs. Available online: https://api-docs.deepseek.com/guides/reasoning_model (accessed on 25 June 2025).
48. Mistral AI. Sampling Guide | Mistral AI Documentation. Available online: <https://docs.mistral.ai/guides/sampling/> (accessed on 12 June 2025).
49. Morgan, J.; Chiang, M. Ollama: Get Up and Running with Large Language Models. GitHub. 2023. Available online: <https://github.com/ollama/ollama> (accessed on 10 February 2025).
50. Fisher, R.A. On the Interpretation of χ^2 from Contingency Tables, and the Calculation of P. *J. R. Stat. Soc.* **1922**, *85*, 87–94. [\[CrossRef\]](#)
51. Benjamini, Y.; Hochberg, Y. Controlling the false discovery rate: A practical and powerful approach to multiple testing. *J. R. Stat. Soc. Ser. B (Methodol.)* **1995**, *57*, 289–300. [\[CrossRef\]](#)
52. Fachada, N.; Lopes, V.V.; Martins, R.C.; Rosa, A.C. Model-independent comparison of simulation output. *Simul. Model. Pract. Theory* **2017**, *72*, 131–149. [\[CrossRef\]](#)
53. Mann, H.B.; Whitney, D.R. On a test of whether one of two random variables is stochastically larger than the other. *Ann. Math. Stat.* **1947**, *18*, 50–60. [\[CrossRef\]](#)
54. Lacchia, M. Radon. 2012. Available online: <https://github.com/rubik/radon> (accessed on 24 August 2025).
55. McCabe, T.J. A complexity measure. *IEEE Trans. Softw. Eng.* **1976**, *SE-2*, 308–320. [\[CrossRef\]](#)
56. Lehtosalo, J.; van Rossum, G.; Levkivskiy, I.; Sullivan, M.J. mypy—Optional Static Typing for Python. 2014. Available online: <https://www.mypy-lang.org/> (accessed on 24 August 2025).
57. Astral Team. Ruff. 2022. Available online: <https://docs.astral.sh/ruff/> (accessed on 24 August 2025).
58. Ziadé, T.; Sottile, A.; Cordasco, I. Flake8: Your Tool for Style Guide Enforcement. 2016. Available online: <https://flake8.pycqa.org/> (accessed on 24 August 2025).
59. McKinney, W. pandas: A foundational Python library for data analysis and statistics. In *Proceedings of the PyHPC '11: Python for High Performance and Scientific Computing*, Seattle, WA, USA, 18 November 2011.
60. Harris, C.R.; Millman, K.J.; Van Der Walt, S.J.; Gommers, R.; Virtanen, P.; Cournapeau, D.; Wieser, E.; Taylor, J.; Berg, S.; Smith, N.J.; et al. Array programming with NumPy. *Nature* **2020**, *585*, 357–362. [\[CrossRef\]](#)
61. Virtanen, P.; Gommers, R.; Oliphant, T.E.; Haberland, M.; Reddy, T.; Cournapeau, D.; Burovski, E.; Peterson, P.; Weckesser, W.; Bright, J.; et al. SciPy 1.0: Fundamental Algorithms for Scientific Computing in Python. *Nat. Methods* **2020**, *17*, 261–272. [\[CrossRef\]](#)
62. Seabold, S.; Perktold, J. Statsmodels: Econometric and Statistical Modeling with Python. In *Proceedings of the 9th Python in Science Conference*, Austin, TX, USA, 28 June–3 July 2010; pp. 92–96. [\[CrossRef\]](#)
63. R Core Team. *R: A Language and Environment for Statistical Computing*; R Foundation for Statistical Computing: Vienna, Austria, 2025.
64. Fachada, N.; Rodrigues, J.; Lopes, V.V.; Martins, R.C.; Rosa, A.C. micompr: An R Package for Multivariate Independent Comparison of Observations. *R J.* **2016**, *8*, 405–420. [\[CrossRef\]](#)
65. Bender, E.M.; Koller, A. Climbing towards NLU: On meaning, form, and understanding in the age of data. In *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics, Online*, 5–10 July 2020; Jurafsky, D., Chai, J., Schluter, N., Tetreault, J., Eds.; Association for Computational Linguistics: Stroudsburg, PA, USA, 2020; pp. 5185–5198. [\[CrossRef\]](#)

Disclaimer/Publisher’s Note: The statements, opinions and data contained in all publications are solely those of the individual author(s) and contributor(s) and not of MDPI and/or the editor(s). MDPI and/or the editor(s) disclaim responsibility for any injury to people or property resulting from any ideas, methods, instructions or products referred to in the content.