Proc. of the European Safety and Reliability Conference (ESREL 2026)
Edited by José C. Matos, Paulo B. Lourenço, Michael Beer, Daniel Oliveira, Edoardo Patelli, Hélder S. Sousa, Rui A. Silva and Monica Santamaria
© 2026 ESREL 2026 Organizers. *Published by* Research Publishing, Singapore.
doi: 10.3850/ESREL2026061419_esrel26-p27650-cd

A Coding Benchmark for Reliability Engineering with LLMs: Design and Baselines

Solal Baudoin
Laboratoire de Génie Industriel, CentraleSupélec, Université Paris-Saclay, France.
E-mail: solal.baudoin@student-cs.fr

Jean Meunier-Pion
Laboratoire de Génie Industriel, CentraleSupélec, Université Paris-Saclay, France.
E-mail: jean.meunier-pion@centralesupelec.fr

Zhiguo Zeng
Chair on Risk and Resilience of Complex Systems, Laboratoire de Génie Industriel, CentraleSupélec, Université Paris-Saclay, France. E-mail: zhiguo.zeng@centralesupelec.fr

Modern reliability engineering increasingly relies on computer codes for modeling and analysis. As a result, reliability engineers routinely face repetitive, error-prone programming under tight time and resource constraints. Large Language Model (LLM)-based coding assistants have shown great potential to improve efficiency on general-purpose coding tasks. Yet, reliability engineering coding tasks remain largely absent from current LLM coding benchmarks, limiting our ability to assess whether these assistants can reliably support domain-specific work.
To address this gap, we introduce HumanEval-Rel, a HumanEval-inspired suite of hand-written Python coding tasks with executable unit tests that directly measure domain-specific coding ability on foundational reliability engineering skills. HumanEval-Rel contains 40 tasks targeting four areas: (i) reliability metrics and evaluation, (ii) reliability modeling, (iii) maintenance of repairable systems, and (iv) estimation and data analysis. We report baselines from seven widely-used LLMs, covering both proprietary and open-source ones. Benchmark results demonstrate that this dataset is challenging for general-purpose LLMs: the average pass@1 score is 43.8% (ranging from 7.7% to 60.2% depending on the models), and 22.5% of questions are never solved by any tested model. A deeper analysis shows that the tested LLMs perform well on tasks related to reliability metrics and calculations, but poorly on problems requiring multi-step reasoning and maintenance optimization.
By grounding evaluation in the actual computations reliability engineers perform, HumanEval-Rel turns reliability analysis into an executable benchmark, offering the research community a concrete way to measure whether code-generation models understand the math and modeling logic behind real reliability engineering work. It marks a first step toward an AI assistant that can help with reliability engineering coding tasks.
We release questions, solutions, and evaluation scripts at `https://github.com/sonic160/HumanEval-Rel`.

Keywords: Reliability engineering, LLM benchmark, code generation, pass@k

1. Introduction

Reliability engineering and safety assessment increasingly rely on computer codes for modeling, simulation, and data-driven decision support, *e.g.,* estimating lifetime distribution parameters, running Monte Carlo simulations, and evaluating the reliability of complex systems (Rausand et al., 2020; Lawless, 2003; Nakagawa, 2005). As Large Language Model (LLM)-based coding assistants have shown strong performance on general-purpose code generation tasks (Chen, 2021; Rozière, 2023), they represent a promising tool to reduce repetitive programming effort in reliability workflows. However, general-purpose code LLMs may be limited on reliability-specific tasks, since reliability code often embeds domain knowledge (definitions, modeling assumptions, and numerical conventions) and must satisfy stricter requirements on safety criteria, testability, and traceability than what is emphasized in generic coding benchmarks (Rausand et al., 2020; Barlow and Proschan, 1981). This motivates the need for domain-specific coding assistants tailored to reliability engineering. Such

domain-specific coding assistants, however, remain nascent (Zhang et al., 2025) despite high scientific and economic leverage.

As a first step towards developing domain-specific coding assistants for reliability, we aim at establishing an evaluation baseline for reliability engineering code generation. More specifically, we propose in this paper:

- A HumanEval-inspired benchmark (HumanEval-Rel) focused on reliability engineering primitives and workflows *with executable unit tests* and pass@k reporting (Chen, 2021).
- Baseline evaluations across a set of accessible LLMs under a constrained compute budget, revealing consistent strengths and weaknesses across topics.
- Analyses of difficulty and coverage across the benchmark space, with qualitative guidance for engineering practice and future training data.
- A fully reproducible artifact including questions, unit tests, and evaluation scripts.

Through the work developed in this paper, we intend to answer the following research questions:
RQ1: How well do general-purpose code LLMs perform on reliability-oriented tasks?

RQ2: Why general-purpose code LLMs fail in reliability-specific coding tasks?

RQ3: What error patterns dominate across models?

The rest of this paper is organized as follows. The related work is reviewed in Sect. 2. Section 3 describes the benchmark design and evaluation protocol. Section 4 presents baseline results, qualitative observations, and difficulty analysis. We conclude this paper in Sect. 5 with implications for practice and future evaluation work.

2. Related Work

2.1. *Code generation benchmarks*

General-purpose code generation benchmarks have shaped evaluation methodology for programming LLMs. HumanEval introduced executable tests and the pass@k estimator for functional correctness (Chen, 2021). Subsequent work addressed test rigor and data contamination: EvalPlus strengthens test suites for better generalization assessment (Liu, 2023), LiveCodeBench uses temporal filtering to prevent contamination (Jain et al., 2024), and BigCodeBench evaluates compositional reasoning with function calls (Zhuo, 2024). While these resources catalyzed rapid progress, they evaluate generic algorithmic ability rather than domain-specific analytical workflows.

2.2. *Domain-specific evaluation*

Domain-specific benchmarks move beyond single-function prompts. DS-1000 focuses on data-science code generation grounded in library usage (Lai, 2022), while SciCode evaluates research-level code generation with 338 subproblems from 80 challenging scientific problems (Tian, 2024). Despite these advances, none explicitly target canonical reliability engineering computations (e.g., CDF↔reliability↔hazard transforms, mean residual life), system structure functions (series/parallel/k-out-of-n, minimal cut sets), or maintenance/availability calculations.

2.3. *Reliability engineering foundations*

Reliability engineering rests on well-established mathematical foundations spanning component/system reliability, maintenance theory, and survival analysis (Barlow and Proschan, 1981). Recent interest in LLMs for industrial applications has emerged, including predictive maintenance and fault diagnosis (Zhang et al., 2025), yet executable benchmarks for reliability-specific code generation remain absent.

2.4. *Positioning of HumanEval-Rel*

Our benchmark, HumanEval-Rel, stands at the intersection of code generation evaluation and reliability engineering. Following HumanEval-style suites, HumanEval-Rel uses executable unit tests and pass@k for objective scoring.

3. The HumanEval-Rel dataset

3.1. *Motivation and goals*

The benchmark targets core reliability competencies and pairs conceptual tasks with strict implementation requirements. Questions are presented as concise prompts with a function signature and reference tests. The evaluation spans four domains, and multiple subthemes, as presented in Table 1. Tasks are grounded in standard reliability patterns used in practice and education, avoiding domain-specific shortcuts. As detailed in Table 1, the required computational methods span varied difficulty levels—from direct formula application to advanced techniques.

3.2. *Content and structure*

The benchmark dataset contains 40 items in HumanEval-style. Each question provides a function signature, a docstring describing the task, and example input-output pairs. Figure 1 shows an example question from the benchmark. The model must generate a complete implementation that passes the hidden unit tests. Correctness is verified via unit tests.

All tasks and unit tests are manually authored by reliability domain experts, tasks reflect realistic reliability workflows, they balance symbolic manipulation (e.g., transformations among the CDF, PDF, reliability, and failure rate), numerical precision (tolerances), and robust handling of edge cases.

3.3. *Evaluation protocol*

We report pass@k following Chen (2021). For each question, we sample up to k completions and mark the question as solved if any completion passes all unit tests. We estimate pass@k using the unbiased estimator of Chen (2021):

$$\widehat{\text{pass@}k}(\text{model}, \text{question}) = 1 - \frac{\binom{n-c}{k}}{\binom{n}{k}},$$

where c is the number of correct completions from the model among n samples (with $0 \leq k \leq n$). We summarize performance by averaging pass@k across questions and also report per-question results.

```
def cal_series_sys_rel(comp_rel: np.
    array) -> float:
    '''
    You will be given a numpy array
        comp_rel.
    Your task is to calculate the
        system
    reliability of a series system,
        where the
    reliability of the components in
         the system
    is given in comp_rel.

    Examples:
    >>> cal_series_sys_rel(np.array
        ([.9]))
    .9
    >>> cal_series_sys_rel(np.array
        ([.9, .8]))
    .72
    '''
```

Fig. 1. Example question from the Reliability Modeling category: computing series system reliability.

We also report maximum pass@k, defined for any question q as

$$\text{max_pass@}\,k(q) = \max_{m \in \text{models}} \left[\widehat{\text{pass@}k}(m, q)\right]$$

which captures the best performance achieved by any model on that question, highlighting question difficulty.

For each model and each question we sample $n = 50$ independent completions, and report those metrics for $k \in \{1, 2, 5, 10, 20\}$. Because LLM code generation is non-deterministic, sampling multiple completions captures this variability and avoids relying on a single attempt.

4. Results and discussions

4.1. *Model selection*

Often, a practical project faces constraints from limited computational resource and budget. To take this into account, when choosing the LLMs for the benchmark study, we prioritized models that could either run locally with manageable computational resource (one Nvidia A100), or could be accessed through API with acceptable price (maximum 1.6$ /millions tokens from Open-

Table 1. Benchmark taxonomy.

Category	Subthemes	Count
Reliability metrics & evaluation (Metrics)	Basic reliability concepts (CDF, reliability, failure rate, Mean residual life (MRL), etc.); Empirical estimators from samples; Lifetime distributions and mixtures of distributions.	14
Reliability modeling (Modeling)	Structure functions: series, parallel, k-out-of-n (G/F); Minimal cut-set aggregation; Time-dependent reliability modeling: system MTTF, median life, failure rate; Importance measures: Birnbaum and criticality.	9
Estimation & Data analysis (Estimation)	Kaplan-Meier estimation, Censoring, Maximum likelihood, Bayesian analysis, Degradation Models.	6
Maintenance of repairable systems (Maintenance)	Renewal processes, Availability, Age based replacement strategies, Limit & interval availability.	11

Router). Within these constraints, we selected the models with best performance on coding from HuggingFace's OpenLLM Leaderboard Fourrier et al. (2024) as of 06/2025. Based on these criteria, we selected seven LLMs as baselines, as shown in Table 2.

4.2. *Main results*

Table 2 shows pass@1 ranges from 7.7% (LLaMA-3-8B) to 60.2% (QwQ-32B), with mean 43.8%. This aggregate masks substantial variance: nine questions (22.5%) were never solved by any model across all 350 attempts, while no task achieved 100% pass rate across all models.

Figure 2 shows the pass@1 across different category of questions (see Table 1 for details). It can be seen that different category of questions has diverse performance: Metrics (53.6%) $>$ Modeling (48.0%) $>$ Estimation (46.6%) $>$ Maintenance (26.3%). The main reason for this is that the questions in Maintenance and Estimation usually requires multi-step reasoning before the coding agent can generate the correct code, while the other categories are more direct formula applications that can be directly transformed into codes.

Performance distribution and improvement with multiple attempts

The variance in model performance is substantial. Figure 3 reveals the distribution of pass@1 success rates across the 40 questions, showing that while the mean is 43.8%, individual question difficulty ranges from 0% (nine never-solved tasks) to near-universal success. This heterogeneity reflects the range of cognitive demand: s e metric transformations (CDF-to-reliability conversions) versus multi-step estimation and optimization problems.

Beyond pass@1, the question arises: does allowing multiple generation attempts improve performance? Figure 4 addresses this by plotting pass@k curves (the probability of solving a question with k independent attempts). QwQ-32B dominates across all k values, approaching 80% at k=20, while other models plateau earlier. The shape of these curves reveals an important distinction.

Generation variability versus algorithmic understanding.

When a model knows the correct algorithmic approach but generates implementation variants (e.g., different loop structures, forgotten imports, off-by-one errors), increasing k allows the randomness of generation to eventually produce a passing solution. This is evident in tasks requiring direct formula application: Metrics and basic Modeling questions show steady improvement with k.

Conversely, for tasks requiring a specific algorithmic pipeline (e.g., solving renewal integrals, implementing censored MLE optimization), the curves flatten. If a model lacks the conceptual bridge between mathematical formulation and numerical implementation, additional attempts with the same underlying model do not recover the missing method. This reflects a fundamental asymmetry: pass@k can compensate for *execution variance* but not for *algorithmic ignorance*.

Fig. 5 shows the efficiency trade-off. GPT-4.1-mini is the most token-efficient at 1,163 tokens per

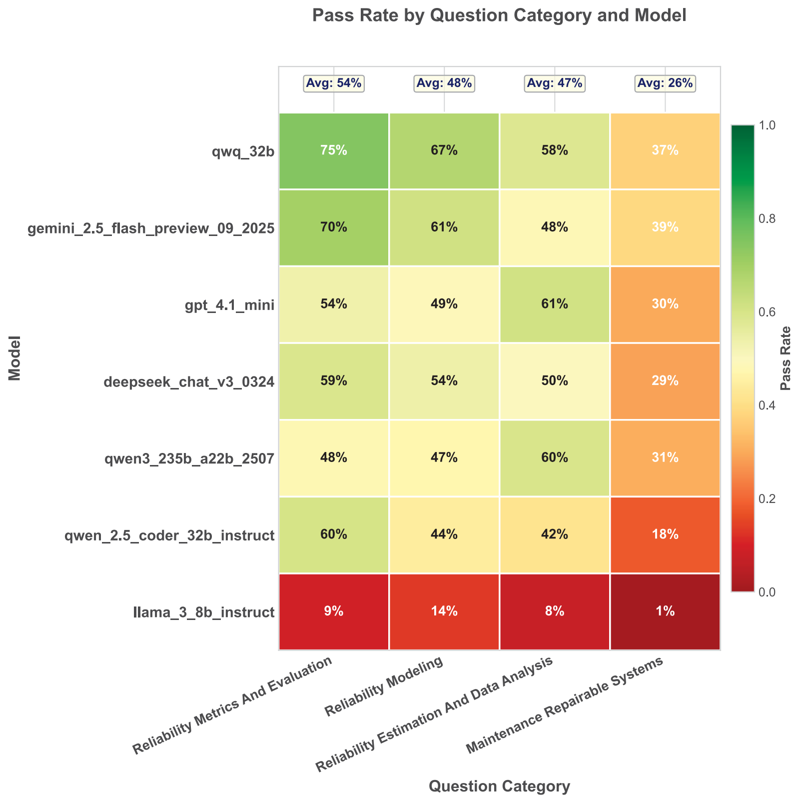

Fig. 2. Pass@1 heatmap across models and questions.

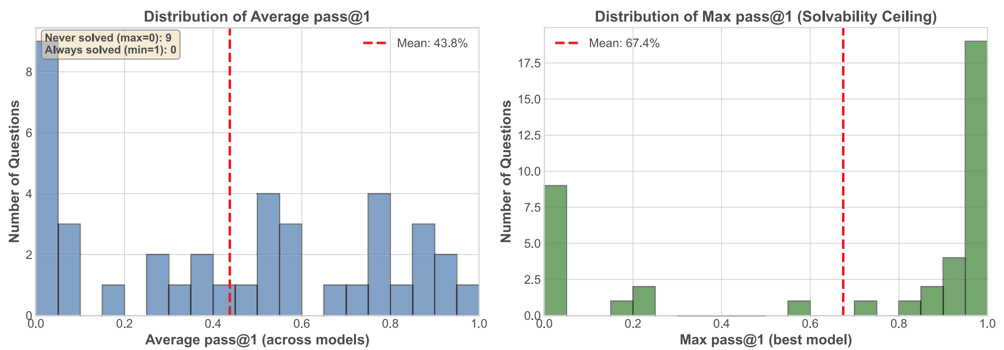

Fig. 3. Distributions of average and maximum pass@1 across questions. The mean average pass@1 is 43.8 %.

correct answer, while QwQ-32B requires 10,770 (9× more) for 13 percentage points of additional accuracy. However, token counts and monetary costs diverge: based on OpenRouter API pricing (right panel), Qwen3-235B becomes the most cost-effective model, while Gemini-2.5-Flash—3rd in token efficiency—is the most expensive.

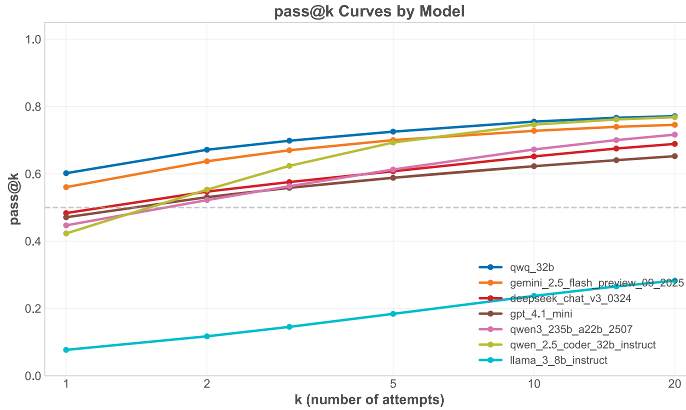

Fig. 4. Pass@k curves showing how average success rate improves with more attempts (k). QwQ-32B achieves the highest performance across all k values, reaching ≈80% at k=20.

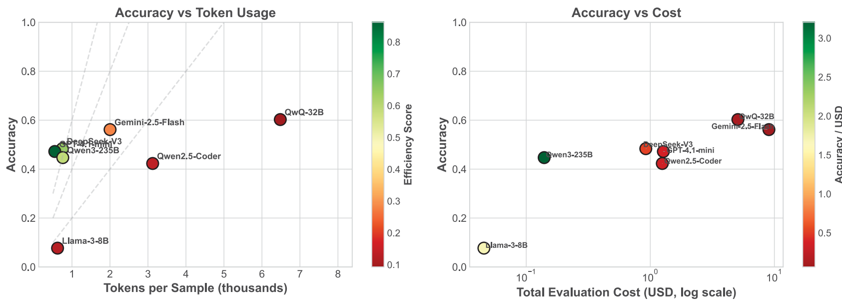

Fig. 5. Accuracy vs. token usage (left) and vs. monetary cost (right, OpenRouter pricing, log scale). Model rankings diverge between the two views.

4.3. *Observations and discussions*

We conducted a detailed qualitative inspection of the 14,000 generated code samples to understand the failure patterns. Our analysis reveals distinct failure patterns across questions and models.

Error Type Distribution.

Table 3 summarizes the error type distribution across models. The dominant error type is *wrong answer* (21–39%), where syntactically correct code produces incorrect numerical results. *No Output*, corresponds to no correctly-formatted code snippet is produced by the LLMs. `NameErrors` (“lazy coding”), corresponds to models calling helper functions like `math.exp` or `scipy.integrate.quad` without importing them.

GPT-4.1-mini exhibits the cleanest behavior (0% No Output, 0.2% NameError), while Qwen-2.5-coder shows a striking 13.9% No Output rate. QwQ-32B also shows elevated No Output (7.6%), suggesting reasoning models sometimes fail to produce complete code despite stronger overall accuracy.

The elevated No Output rates for reasoning models stem from token budget constraints. Reasoning-focused models like QwQ-32B generate lengthy chain-of-thought traces before producing code, consuming a substantial portion of the available token budget. Two failure modes emerge: (1) the model’s internal reasoning process exceeds token limits mid-generation, truncating output before code generation begins, and (2) rea-

Table 2. Evaluated models and pass@1 performance on HumanEval-Rel.

Model	Creator	Type	Access	Size	pass@1	Std. Error	Tokens
qwq-32b	Alibaba	Open	Local	32B	**60.2%**	$\pm$6.6%	6,483
gemini-2.5-flash-preview	Google	Proprietary	OpenRouter	Unknown	56.2%	$\pm$6.6%	2,003
deepseek-chat-v3-0324	Deepseek	Open	OpenRouter	671B-a37B	48.4%	$\pm$6.9%	748
gpt-4.1-mini	OpenAI	Proprietary	OpenRouter	Unknown	47.2%	$\pm$7.0%	548
qwen3-235b-a22b	Alibaba	Open	OpenRouter	235B-a22B	44.7%	$\pm$6.7%	758
qwen-2.5-coder-32b	Alibaba	Open	Local	32B	42.3%	$\pm$5.5%	3,123
llama-3-8b-instruct	Meta	Open	Local	8B	7.7%	$\pm$2.8%	621
Mean	–	–	–	–	**43.8%**	–	2,040

soning traces enter infinite loops or excessively deep recursion, consuming tokens without forward progress. In contrast, direct-prediction models like GPT-4.1-mini produce code more directly, bypassing these reasoning-induced bottlenecks.

4.4. *Failure analysis: Deep dive on hard questions*

Having identified error distributions across models, we now examine the failure modes of the most-difficult questions in detail. A detailed case-by-case analysis of the nine never-solved tasks is available in the extended version of this paper with appendices, provided in our GitHub repository. The following synthesis extracts common patterns and actionable insights from these cases.

In summary, failures stem from three sources: (1) *numerical instability* in optimization-integration loops, (2) *overcomplication* of statistical formulas, and (3) *implementation gaps* where theoretical knowledge does not translate to working code for specialized reliability computations.

The Implementation Gap.

A key finding is that difficulty stems primarily from the lack of high-level library support. In standard data science, models leverage `scikit-learn` or `statsmodels`. For reliability computations—renewal processes, censored likelihoods, availability calculations—specialized functions are absent from pre-installed libraries. Models must implement these from scratch, revealing weaknesses in numerical stability (e.g., overflow in survival function tails) and edge-case handling (e.g., empty censor sets, degenerate distributions).

The nine never-solved questions cluster in specific subtopics: *Maintenance* (4 tasks: optimal replacement intervals, renewal density, repairable system availability), *Modeling* (2 tasks: competing risks, dependent components), *Metrics* (2 tasks: mean residual life under non-standard distributions), and *Estimation* (1 task: MLE under interval censoring). These share a common pattern: they require implementing algorithms not available in standard libraries (e.g., renewal equation solvers, censored likelihood maximizers).

Semantic vs. Syntactic Difficulty.

Unlike general benchmarks that primarily test algorithmic logic, HumanEval-Rel tests *domain fidelity*. A model can produce syntactically correct Python that is mathematically meaningless—for instance, treating a hazard rate as a probability or confusing reliability with availability. This “semantic difficulty” makes the benchmark a rigorous test for engineering-oriented AI agents where domain correctness is paramount.

5. Conclusion

We present HumanEval-Rel, a HumanEval-style benchmark for reliability engineering code generation, and report baseline results on seven accessible LLMs. Overall, current general-purpose code LLMs show solid performance on direct reliability computations, but they remain brittle on tasks that require multi-step reasoning, numerical optimization, and maintenance-policy logic.

The evaluation also exposes recurring failure modes beyond “wrong answers,” including incomplete outputs and missing imports (`NameError`). These observations suggest that future work should focus on:

Retrieval-Augmented Generation (RAG): Pro-

Table 3. Distribution of errors across representative models.

Error Type	GPT-4.1-mini	DeepSeek-v3	QwQ-32B	Qwen-2.5-coder
Pass	47.2	48.4	60.2	42.3
Wrong answer	38.6	30.6	21.1	30.6
No Output†	0.0	2.1	7.6	13.9
NameError‡	0.2	2.7	0.2	0.5
TypeError	3.0	3.1	3.0	4.5
Timeout	2.5	2.6	4.2	2.1
Other	8.6	10.4	3.5	6.2

†Model did not output the required function.
‡"Lazy coding": calls undefined helper functions.

viding models with access to verified reliability libraries or domain documentation to reduce hallucination of non-existent functions.

Agentic Workflows: Allowing models to execute code iteratively, observe runtime errors like `NameError`, and self-correct through multiple refinement steps.

Domain-Specific Fine-Tuning: Training on high-quality reliability engineering codebases to internalize the implementation details of specialized algorithms for censored data, renewal processes, and optimization.

Finally, our cost analysis shows that token efficiency and monetary cost yield different model rankings, since per-token pricing varies widely across providers. Deployment decisions should therefore consider actual API costs alongside token counts.

We encourage evaluation on stronger frontier models to track progress toward dependable reliability engineering assistants.

References

Barlow, R. E. and F. Proschan (1981). *Statistical Theory of Reliability and Life Testing: Probability Models*. Holt, Rinehart and Winston.

Chen, M. e. a. (2021). Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374*.

Fourrier, C., N. Habib, A. Lozovskaya, K. Szafer, and T. Wolf (2024). Open llm leaderboard v2. `https://huggingface.co/spaces/open-llm-leaderboard/open_llm_leaderboard`.

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

Lai, Y. e. a. (2022). Ds-1000: A natural and reliable benchmark for data science code generation. *arXiv preprint*.

Lawless, J. F. (2003). *Statistical Models and Methods for Lifetime Data* (2 ed.). Wiley.

Liu, Z. e. a. (2023). Evalplus: Towards reliable evaluation of code generation models. *arXiv preprint*.

Nakagawa, T. (2005). *Maintenance Theory of Reliability*. Springer.

Rausand, M., A. Barros, and A. Høyland (2020). *System Reliability Theory: Models, Statistical Methods, and Applications*. John Wiley & Sons, Inc.

Rozière, B. e. a. (2023). Code Llama: Open foundation models for code. *arXiv preprint arXiv:2308.12950*.

Tian, M. e. a. (2024). SciCode: A research coding benchmark curated by scientists. *arXiv preprint arXiv:2407.13168*.

Zhang, Y., W. Zhang, H. Liu, and X. Chen (2025). Large language models in reliability systems engineering: A comprehensive review. *Reliability Engineering & System Safety*. In press.

Zhuo, T. Y. e. a. (2024). BigCodeBench: Benchmarking code generation with diverse function calls and complex instructions. *arXiv preprint arXiv:2406.15877*.