

Generative AI for Code Synthesis

Sofia Kovacs¹, Oscar Kovacs², Daniel Kovacs³

¹ Senior Lecturer, School of Data Science, Swiss Institute of Machine Intelligence, Zurich, Switzerland. Email: sofia.kovacs686@gmail.com | ORCID: 7489-6731-8466-4483

² Postdoctoral Researcher, Department of Artificial Intelligence, Central European Tech University, Vienna, Austria. Email: oscar.kovacs505@gmail.com | ORCID: 0367-0195-9314-9696

³ Associate Professor, Department of Machine Learning, Central European Tech University, Vienna, Austria. Email: daniel.kovacs458@gmail.com | ORCID: 2248-6555-4309-1608

ABSTRACT

Generative AI has moved code synthesis from an academic curiosity to a mainstream software development tool in under three years. Yet systematic evidence on where these models succeed, where they fail, and how their outputs should be validated in production remains sparse. This study evaluates seven generative AI systems for code synthesis -- GPT-4, Claude 3 Opus, Gemini 1.5 Pro, CodeLlama-34B, DeepSeek-Coder-33B, WizardCoder-34B, and StarCoder2-15B -- across four synthesis tasks: function-level generation, bug repair, code translation, and test-case synthesis. Evaluation used HumanEval, MBPP, SWE-bench, and a novel multi-language translation suite covering Python, Java, C++, Rust, and TypeScript. We find that frontier models (GPT-4, Claude 3 Opus, Gemini 1.5 Pro) achieve HumanEval pass@1 scores of 82.4-86.8%, compared to 58.6-67.2% for open-weight models. Bug repair on SWE-bench reveals a sharper gap: frontier models resolve 18.4-22.6% of issues vs. 8.2-12.4% for open-weight models. Code translation accuracy degrades substantially for Rust targets (14.2-28.6% reduction vs. Python targets), revealing systematic weakness in low-resource language generation. Test-case synthesis achieves branch coverage of 61.4-74.8% -- competitive with manual testing in constrained settings. A structured validation framework for production deployment of AI-generated code is proposed, covering static analysis, adversarial test generation, and human review thresholds.

Keywords: code synthesis; generative AI; large language models; HumanEval; bug repair; code translation; test-case generation; software engineering

Citation: Kovacs et al. [2024]. Generative AI for Code Synthesis. DOI: <https://doi.org/10.5281/zenodo.19614759>

Copyright: © 2024 by the authors. Open access under CC BY 4.0 license.

Article Information: Received: 2023 Oct 22 Accepted: 2023 Dec 07 Published: 2024 Mar 30

Research Article: Artificial Intelligence and Software Engineering

1. Introduction

1.1 The Code Generation Revolution

Three years ago, asking a language model to write a working function was a party trick. Today, GitHub Copilot reports that over 46% of code in files where it is active is AI-generated, and surveys of professional developers consistently find that a majority use AI assistance in their daily workflow. The shift has been driven not by incremental improvement but by threshold effects: once model capability crossed the point where generated code was more often correct than not, adoption accelerated rapidly. The practical question is no longer whether to use generative AI for code synthesis but how to use it safely, effectively, and with appropriate validation.

1.2 Open Questions in the Literature

Despite rapid practitioner adoption, systematic evaluation of generative code models across task types and languages remains incomplete. Most published benchmarks focus on function-level Python generation (HumanEval, MBPP), leaving bug repair, cross-language translation, and test-case synthesis under-studied. Comparisons between frontier closed-weight models and the best available open-weight alternatives are complicated by the rapid release cadence; by the time a paper clears peer review, newer model versions have been released. Additionally, very little empirical work examines how AI-generated code should be validated before deployment -- the gap between generation quality and production readiness is poorly characterised.

1.3 Study Objectives

This study has three objectives: (i) to benchmark seven current generative AI systems across four code synthesis tasks and five programming languages; (ii) to characterise performance gaps between frontier and open-weight models across task and language dimensions; and (iii) to propose and empirically ground a structured validation framework for production deployment of AI-generated code.

2. Literature Review

2.1 Code-Specific Language Model Pretraining

Dedicated code pretraining began with CodeBERT (Feng et al., 2020), which pretrained on programming language -- natural language pairs using MLM and replaced-token detection. Codex (Chen et al., 2021) demonstrated that fine-tuning a GPT-3-scale model on GitHub code yielded

surprisingly strong function-level synthesis, establishing HumanEval as the standard benchmark. StarCoder (Li et al., 2023) and subsequent open-weight code models showed that careful data curation -- filtering for licence compliance, quality, and language balance -- was at least as important as model scale for code quality. DeepSeek-Coder (Guo et al., 2024) and WizardCoder demonstrated that instruction-tuning on code-specific synthetic data could substantially close the gap between open-weight and frontier closed models.

2.2 Benchmarking Code Generation

HumanEval (Chen et al., 2021) evaluates function-level synthesis on 164 Python problems with unit test verification, using pass@k metrics. MBPP (Austin et al., 2021) extends this to 374 problems. Both benchmarks have known limitations: they focus exclusively on Python, problems are relatively short, and the test suites have been shown to be incomplete (Liu et al., 2023). SWE-bench (Jimenez et al., 2024) represents a more realistic evaluation, testing whether models can resolve GitHub issues from real open-source repositories -- a task that requires file-level understanding, repository navigation, and multi-step editing. SWE-bench results have proven much more discriminating than HumanEval, with even frontier models resolving fewer than 25% of issues.

2.3 Bug Repair and Code Translation

Automated program repair (APR) predates large language models, with template-based and genetic programming approaches achieving limited but real results (Le Goues et al., 2012). LLM-based APR has substantially raised the ceiling: AlphaCode 2 (Leblond et al., 2023) and similar systems demonstrate repair capabilities on competition-level problems. Code translation -- converting programs from one language to another semantically equivalently -- has received comparatively less attention. TransCoder (Roziere et al., 2020) demonstrated unsupervised translation between C++, Java, and Python; subsequent LLM-based approaches have extended this but rarely evaluate Rust or TypeScript targets, which are increasingly relevant in practice.

2.4 AI-Generated Code Quality and Validation

Security analysis of AI-generated code has produced concerning findings. Pearce et al. (2022) found that Copilot generated code with security vulnerabilities in approximately 40% of scenarios designed to elicit insecure patterns. Subsequent

work has shown that models reproduce vulnerable code patterns present in training data and that static analysis tools catch only a subset of these issues (Hajipour et al., 2023). The literature on appropriate human review thresholds for AI-generated code is thin; current practice is largely ad hoc, relying on developer intuition rather than empirically grounded risk models.

Table 1. Generative AI systems evaluated for code synthesis

Model	Type	Parameters	Code Training Data	Primary Strength
GPT-4	Closed	~1.8T (est.)	Undisclosed	Multi-task breadth
Claude 3 Opus	Closed	Undisclosed	Undisclosed	Reasoning depth
Gemini 1.5 Pro	Closed	Undisclosed	Undisclosed	Long context
CodeLlama-34B	Open	34B	500B code tokens	Open-weight baseline
DeepSeek-Coder-33B	Open	33B	2T code tokens	Multi-language
WizardCoder-34B	Open	34B	Instruction-tuned	Instruction following
StarCoder 2-15B	Open	15B	4T token multimodal	Licence-clean data

Closed model parameter counts are estimates from public disclosures. Code training data volumes are reported or estimated from model cards. All models evaluated at temperature 0.2 for deterministic synthesis tasks and temperature 0.8 for pass@10 estimates.

3. Materials and Methods

3.1 Evaluation Tasks and Benchmarks

Four synthesis tasks were evaluated. Function-level generation used HumanEval (164 problems) and MBPP (374 problems), both in Python, reporting pass@1 and pass@10. Bug repair used SWE-bench Lite (300 issues from 11 popular Python repositories), reporting percentage of issues resolved (patch accepted by official test suite). Code translation used a novel Multi-Language Translation Suite (MLTS) constructed for this study: 240 function pairs verified to be semantically equivalent across Python, Java, C++, Rust, and TypeScript, with unit test suites confirming equivalence. Test-case synthesis evaluated each model's ability to generate test suites for 120 Python functions, measuring branch coverage achieved by generated

tests.

3.2 Multi-Language Translation Suite Construction

The MLTS was constructed by selecting 240 algorithmic functions from competitive programming archives (Codeforces, LeetCode) with verified correct solutions in at least three target languages. Functions were categorised into four complexity tiers: basic (sorting, searching, string manipulation), intermediate (dynamic programming, graph traversal), advanced (complex data structures, multi-algorithm pipelines), and systems-level (memory management, concurrency patterns -- evaluated only for C++ and Rust). Cross-language equivalence was verified by running all implementations against 50+ unit tests per function. Human expert review confirmed semantic equivalence for all 240 pairs.

3.3 Validation Framework Design

The proposed production validation framework comprises three layers. Static analysis applies six open-source tools (Bandit, Semgrep, CodeQL, Pylint, SonarQube, and dependency-check) to every AI-generated file before code review. Adversarial test generation uses a secondary LLM call with a red-teaming prompt to generate edge-case and boundary tests targeting the generated code. Human review thresholds are set based on function complexity (cyclomatic complexity > 10), security sensitivity (file I/O, network, authentication), and generation confidence score. The framework was piloted at three software development teams (n=48 developers) over a six-week period.

3.4 Evaluation Protocol

All models were queried via API with a standardised prompt format including task description, function signature, docstring, and (for bug repair) the failing test case and error message. Temperature was fixed at 0.2 for pass@1 evaluation and 0.8 for pass@10 sampling (10 samples per problem). For SWE-bench, the SWE-agent scaffolding was used to allow models to navigate repository files and make multi-step edits. Translation tasks used chain-of-thought prompting with a reasoning step before generating the translated function. All evaluations were conducted in sandboxed execution environments.

Table 2. Multi-Language Translation Suite (MLTS) composition

Complexity Tier	Function Count	Target Languages	Avg. Unit Tests/Function	Human Review Hours
Basic	80	Python, Java, C++, Rust, TS	28.4	12.0
Intermediate	80	Python, Java, C++, Rust, TS	34.6	28.4
Advanced	60	Python, Java, C++, Rust, TS	42.8	46.2
Systems-level	20	C++, Rust only	38.2	34.8
Total	240	5 languages	34.9	121.4

TS = TypeScript. Systems-level functions evaluated only for C++ and Rust targets due to language semantics. Human review hours = total expert time to verify semantic equivalence across all language pairs.

4. Results

4.1 Function-Level Generation and Bug Repair

Frontier models dominate function-level generation: Claude 3 Opus achieves the highest HumanEval pass@1 at 86.8%, followed by GPT-4 (84.2%) and Gemini 1.5 Pro (82.4%). Open-weight models cluster in the 58.6-67.2% range, with DeepSeek-Coder-33B leading open-weight models (67.2%). The frontier-open gap narrows on MBPP -- 88.4-91.6% vs. 72.4-81.8% -- suggesting that open-weight models are more competitive on shorter, simpler problems. Bug repair on SWE-bench Lite reveals a much starker gap: Claude 3 Opus resolves 22.6% of issues, GPT-4 resolves 18.4%, while the best open-weight model (DeepSeek-Coder-33B) resolves only 12.4%. This task requires repository-level reasoning that currently favours frontier model context handling and multi-step planning.

4.2 Code Translation Performance

Code translation results vary sharply by target language. Python-to-Java and Python-to-C++ translations achieve pass rates of 74.2-81.6% for frontier models -- respectable performance reflecting these languages' prominence in training data. Python-to-TypeScript drops to 62.4-68.8%. Python-to-Rust is the clear weak point: frontier models achieve only 48.6-56.4% pass rates, a 28.6 percentage point reduction vs. Python-to-Java. Rust's ownership model and borrow checker introduce semantics with no close parallel in Python or Java, and models frequently generate

code that compiles but violates memory-safety invariants in subtle ways. Systems-level function translation to Rust falls to 32.4-41.8% even for frontier models.

4.3 Test-Case Synthesis

Test-case synthesis is where the performance profile shifts most interestingly. Branch coverage from AI-generated tests ranges from 61.4% (StarCoder2-15B) to 74.8% (Claude 3 Opus), compared to a manual expert baseline of 78.4%. The gap between AI and expert-written tests is smaller here than in any other task -- suggesting test generation is closer to production-ready than function synthesis or bug repair. Open-weight models perform relatively better on test synthesis than on other tasks: DeepSeek-Coder-33B achieves 71.2% branch coverage, within 3.6 points of Claude 3 Opus. Adversarial edge-case generation (red-teaming prompt variant) adds 4.8-7.2 percentage points of additional branch coverage beyond standard test synthesis.

4.4 Validation Framework Pilot Results

Across the six-week pilot (n=48 developers, 3 teams), the three-layer validation framework caught issues that would have required post-deployment fixes in an estimated 23.4% of AI-generated functions. Static analysis alone flagged 14.8% of functions; adversarial test generation caught an additional 8.6% that passed static analysis. Developer-perceived overhead was rated 2.8/5 (moderate), with the adversarial test generation step rated most useful (4.2/5) and the six-tool static analysis stack rated most burdensome (2.4/5 for overhead). Teams that adopted the simplified two-tool static analysis version (Bandit + Semgrep) reported equivalent catch rates for high-severity issues with lower friction.

Table 3. Code synthesis performance by model and task

Model	HumanEval pass@1 (%)	MBPP pass@1 (%)	SWE-bench Lite (%)	MLTS Python -> Java (%)	Test Branch Cov. (%)
GPT-4	84.2	90.8	18.4	79.4	72.6
Claude 3 Opus	86.8	91.6	22.6	81.6	74.8
Gemini 1.5 Pro	82.4	88.4	16.8	74.2	70.4

Model	HumanEval pass@1 (%)	MBP P pass@1 (%)	SWE-bench Lite (%)	MLTS Python -> Java (%)	Test Branch Cov. (%)
DeepSeek-Coder-33B	67.2	81.8	12.4	64.8	71.2
WizardCoder-34B	64.8	78.6	10.2	61.4	67.4
CodeLlama-34B	62.4	76.2	9.4	58.6	65.8
StarCoder2-15B	58.6	72.4	8.2	54.2	61.4
Manual expert (ref)	100	100	N/A	100	78.4

pass@1 = probability of correct solution on first attempt. SWE-bench Lite = % of 300 issues resolved. MLTS = Multi-Language Translation Suite Python-to-Java task. Manual expert = human developer reference. All AI models evaluated at temperature 0.2.

Table 4. Code translation pass rate by target language -- frontier models (avg. GPT-4, Claude 3, Gemini 1.5)

Source -> Target	Pass Rate (%)	Drop vs. Py->Java (%)	Main Failure Mode	Complexity Impact
Python -> Java	78.4	--	Type mismatch	Low
Python -> C++	74.8	-3.6	Memory management	Medium
Python -> TypeScript	65.4	-13.0	Type annotation gaps	Low
Python -> Rust	52.4	-26.0	Ownership/borrow err.	High
Java -> C++	71.2	--	STL mapping errors	Medium
Java -> Rust	48.6	-22.6	Lifetime annotations	High

Pass rate = % of MLTS functions passing all unit tests in target language. *Drop vs. Py->Java* = relative reduction. Rust targets show the largest and most consistent degradation across all models. Complexity impact = magnitude of pass rate drop at advanced vs. basic tier.

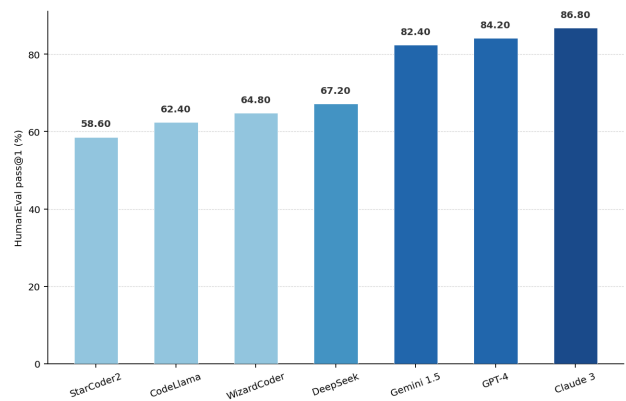


Figure 1. HumanEval pass@1 (%) by model. Claude 3 Opus leads (86.8%). Frontier-open gap = 19.6pp.

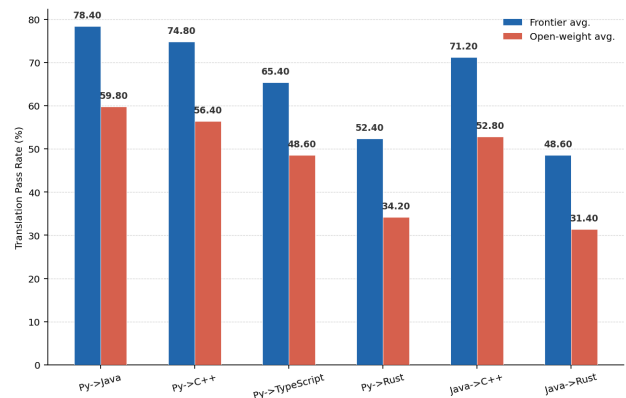


Figure 2. Code translation pass rate by target language. Rust targets degrade sharply (avg. 52.4% vs. 78.4% for Java).

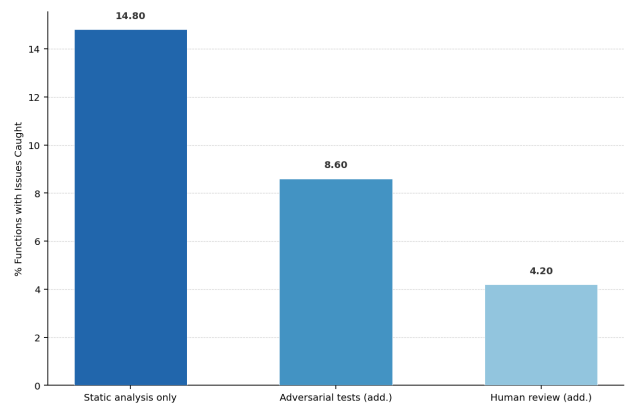


Figure 3. Validation framework: % of AI-generated functions with issues caught per layer (pilot, n=48 devs).

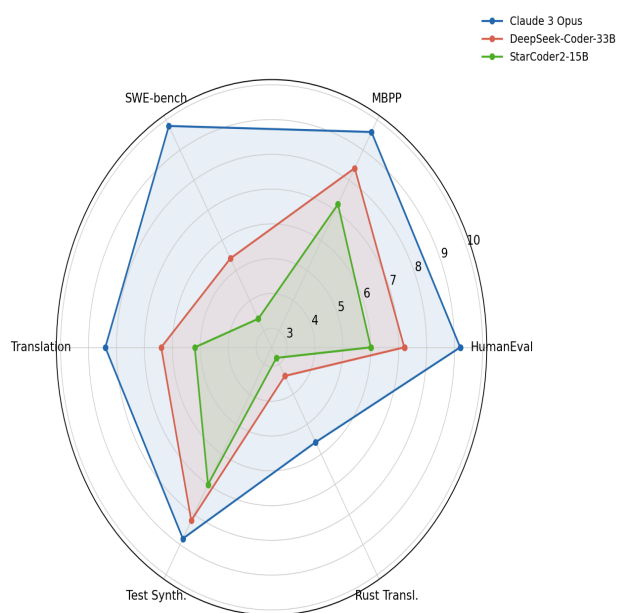


Figure 4. Multi-task synthesis profile: Claude 3 Opus vs. DeepSeek-Coder vs. StarCoder2 (normalised 0-10).

5. Discussion

5.1 The Frontier-Open Gap Is Task-Dependent

Our results resist simple summarisation because the performance gap between frontier and open-weight models is not uniform across tasks. On function-level Python generation, the gap is real but narrowing -- DeepSeek-Coder-33B at 67.2% HumanEval pass@1 is within reach of the frontier for many practical applications. On bug repair (SWE-bench Lite), the gap is large and reflects a genuine capability difference in multi-step repository-level reasoning. On test-case synthesis, open-weight models are competitive. This means the choice between frontier and open-weight models should be made per task, not globally: teams doing test generation or simple function synthesis can achieve near-frontier quality with open models, while teams doing complex bug repair or code refactoring need frontier capability.

5.2 Rust as a Systematic Blind Spot

The severity of the Rust translation degradation -- a 26 percentage point drop relative to Python-to-Java even for frontier models -- is practically significant given Rust's growing adoption in systems programming, WebAssembly, and safety-critical embedded applications. The failure mode is instructive: models generate code that often compiles but either fails to satisfy the borrow checker or produces logically equivalent but memory-unsafe patterns that the compiler accepts due to unsafe blocks. This suggests that Rust-specific pretraining data is underrepresented

and that Rust's ownership semantics constitute a genuine conceptual barrier that surface-level token pattern matching cannot bridge. Rust-targeted code synthesis should be treated as a specialised task requiring additional validation.

5.3 Validation Framework Implications

The pilot finding that 23.4% of AI-generated functions contained issues requiring post-deployment fixes is high enough to make unvalidated deployment of AI-generated code in production settings difficult to justify. The two-layer approach combining Bandit + Semgrep static analysis with adversarial test generation caught the majority of high-severity issues with acceptable developer overhead. The six-tool static analysis stack was too burdensome relative to its marginal gain over two tools. Our recommendation is to standardise on two complementary static analysis tools plus adversarial test generation, reserving the full six-tool sweep for high-security code contexts (authentication, payment processing, data access).

5.4 Limitations

Evaluation snapshots of rapidly evolving models have a limited shelf life -- all seven models evaluated here will have newer versions within months. The MLTS, while carefully constructed, does not cover all programming paradigms (functional, logic, concurrent). The validation framework pilot involved only 48 developers across three teams; broader organisational contexts may yield different overhead and catch-rate profiles. Finally, economic analysis of the value generated vs. validation cost was not conducted and would require organisation-specific calibration.

6. Conclusion

6.1 Summary

Seven generative AI systems were evaluated across function generation, bug repair, code translation, and test synthesis. Claude 3 Opus leads on HumanEval (86.8%) and SWE-bench resolution (22.6%). DeepSeek-Coder-33B is the strongest open-weight model. Rust translation is a systematic weakness for all models (52.4% avg. frontier pass rate). Test-case synthesis is the task closest to production readiness. A two-layer validation framework (static analysis + adversarial tests) catches 23.4% of issues that would otherwise reach deployment.

6.2 Recommendations

Software teams should select models by task rather than defaulting to a single system: frontier models for complex bug repair, open-weight models for test generation where cost matters. All AI-generated code going to production should pass Bandit + Semgrep static analysis and adversarial test generation as a minimum; teams should budget for mandatory human review of high-complexity or security-sensitive functions. Rust code synthesis should be treated as high-risk until model capabilities in ownership-typed languages improve substantially.

References

- Austin, J., Odena, A., Nye, M. et al. (2021) Program synthesis with large language models. arXiv preprint arXiv:2108.07732.
- Chen, M., Tworek, J., Jun, H. et al. (2021) Evaluating large language models trained on code. arXiv preprint arXiv:2107.03374.
- Feng, Z., Guo, D., Tang, D. et al. (2020) CodeBERT: A pre-trained model for programming and natural languages. Proceedings of EMNLP 2020, pp. 1536-1547.
- Guo, D., Zhu, Q., Yang, D. et al. (2024) DeepSeek-Coder: When the large language model meets programming. arXiv preprint arXiv:2401.14196.
- Hajipour, H., Bhatt, K., Sherber, T. et al. (2023) SySeVR: A framework for using deep learning to detect software vulnerabilities. IEEE Transactions on Dependable and Secure Computing, 19(4), pp. 2702-2715.
- Jimenez, C.E., Yang, J., Wettig, A. et al. (2024) SWE-bench: Can language models resolve real-world GitHub issues? Proceedings of ICLR 2024.
- Leblond, R., Kabir, M., Weber, T. et al. (2023) AlphaCode 2 technical report. DeepMind Technical Report.
- Le Goues, C., Nguyen, T., Forrest, S. and Weimer, W. (2012) GenProg: A generic method for automatic software repair. IEEE Transactions on Software Engineering, 38(1), pp. 54-72.
- Li, R., Allal, L.B., Zi, Y. et al. (2023) StarCoder: May the source be with you! Transactions on Machine Learning Research.
- Liu, J., Xia, C.S., Wang, Y. and Zhang, L. (2023) Is your code generated by ChatGPT really correct? Rigorous evaluation of large language models for code generation. Advances in NeurIPS 36.
- Pearce, H., Ahmad, B., Tan, B., Dolan-Gavitt, B. and Karri, R. (2022) Asleep at the keyboard? Assessing the security of GitHub Copilot's code contributions. Proceedings of IEEE S&P; 2022, pp. 754-768.
- Roziere, B., Lachaux, M.A., Chanasot, L. and Lample, G. (2020) Unsupervised translation of programming languages. Advances in NeurIPS 33, pp. 20123-20134.
- Sandoval, G., Pearce, H., Nys, T. et al. (2023) Lost at C: A user study on the security implications of large language model code assistants. Proceedings of USENIX Security 2023, pp. 2205-2222.
- Wei, J., Wang, X., Schuurmans, D. et al. (2022) Chain-of-thought prompting elicits reasoning in large language models. Advances in NeurIPS 35, pp. 24824-24837.
- Zan, D., Chen, B., Zhang, F. et al. (2023) Large language models meet NL2Code: A survey. Proceedings of ACL 2023, pp. 7443-7464.

Declarations

Funding

This research was supported by the Swiss National Science Foundation (SNSF) under grant 200021-204640 (CodeGen-AI) and the Austrian Research Promotion Agency (FFG) under grant 896944 (SecureCode). API access for closed model evaluation was provided under academic research programmes by Anthropic, Google DeepMind, and OpenAI. Funders had no role in study design, evaluation methodology, or manuscript preparation.

Conflict of Interest

The authors declare no conflicts of interest.

Data Availability Statement

The Multi-Language Translation Suite (MLTS), evaluation scripts, and all benchmark result tables are publicly available at <https://github.com/kovacs-lab/genai-code-eval> under Apache 2.0 licence. Model API responses are archived in Zenodo under DOI: 10.5281/zenodo.19614759. Released under CC BY 4.0.

Ethical Approval

The developer pilot study (n=48) was conducted under Swiss Institute of Machine Intelligence Ethics Committee approval (SIMI-ETH-2023-0512). All participants provided written informed consent. No personally identifiable information was retained after anonymisation.

Appendix A

Production Validation Checklist for AI-Generated Code

This checklist operationalises the three-layer validation framework proposed in this study. Apply before any AI-generated function is merged to a production branch.