Benchmarking the Titans: A Multi-Dimensional Empirical Evaluation of LLM Code Generation Quality in the .NET Ecosystem

Seyed Mohammad Mahdi Ghalandarian[1], Majid Bazargani[1] and Masoumeh Taromirad[1,*]

[1]*Amirkabir University of Technology, Iran*

Abstract

Evaluating Large Language Model (LLM) code generation quality requires examining not just whether the generated code is correct, but whether it is maintainable, efficient, and stylistically sound, all of which are qualities of direct importance to software engineering practitioners. Existing benchmarks reduce evaluation to a single Pass@k metric, which obscures critical trade-offs between functional correctness and structural quality. A further limitation is the near-exclusive focus on Python, leaving enterprise-relevant ecosystems such as C# and .NET without dedicated evaluation. This paper presents an automated, multi-dimensional evaluation framework for C# code generation, applying it to four state-of-the-art LLMs: GPT, Gemini, Claude, and Grok. We conduct a controlled experiment across 85 algorithmic tasks derived from HumanEval, generating and evaluating 340 solutions in total, in which each solution is assessed across three independent dimensions: functional correctness via automated unit testing, static code quality via Roslyn AST analysis, and runtime efficiency via adversarial BenchmarkDotNet profiling. Our central finding reveals a substantial gap between correctness and quality attributes (Pearson r = 0.075), demonstrating that Pass@k rankings systematically misrepresent the full LLM performance profile in software engineering contexts. We further characterize GPT's bimodal failure behavior.

Keywords

LLM Code Generation, Software Quality, Empirical Evaluation, C#

CEUR-WS.org/Vol-4338/LLM4SE-paper2.pdf

1. Introduction

The integration of Large Language Models into software development workflows has accelerated rapidly [1], with tools such as GitHub Copilot, Cursor, and Claude Code now widely adopted in both academic and industrial settings [2], placing new demands on evaluation. Industry reports and empirical studies have documented recurring challenges with LLM-generated code in practice, including subtle bugs, security vulnerabilities, and maintainability debt that only becomes apparent after integration [3, 4]. The relevant question is therefore no longer only whether LLM-generated code passes tests, but whether it meets the quality standards that practitioners are expected to maintain.

Current evaluation practices, dominated by benchmarks such as HumanEval [5] and MBPP [6], reduce performance to Pass@k, the probability that at least one of k generated samples passes all unit tests. While Pass@k is well-defined, reproducible, and widely used [5, 6, 7, 8], it is blind to code quality properties that directly affect maintainability, readability, and long-term engineering cost. For example, a function with cyclomatic complexity of 19 that passes all tests and a function with complexity of 2 passing the same tests are indistinguishable under Pass@k, yet they represent meaningfully different artifacts from an engineering standpoint. This constitutes the first fundamental limitation of current LLM code generation evaluation: the reduction of a multi-dimensional quality problem to a single binary correctness signal.

Another limitation is the near-exclusive focus on Python, as evidenced by the dominant benchmarks in the field [5, 6, 8, 7]. The C#/.NET ecosystem, heavily used in enterprise software development, financial systems, and game development, lacks a dedicated evaluation infrastructure that unifies functional

Joint Proceedings of the STAF 2026 Workshops: AgileMDE, GCM, ICMM, LLM4SE, TTC. Rennes, France, June 29–July 3, 2026

*Corresponding authors.

mohammadmehdi.ghn@aut.ac.ir (S. M. M. Ghalandarian); Majidbazargani@aut.ac.ir (M. Bazargani); m.taromirad@aut.ac.ir (M. Taromirad)

© 2026 Copyright for this paper by its authors. Use permitted under Creative Commons License Attribution 4.0 International (CC BY 4.0).

CEUR Workshop Proceedings ceur-ws.org ISSN 1613-0073

published 2026-11-26

correctness testing, runtime profiling, and static AST analysis into a single automated pipeline for LLM-generated code [9, 8]. While tools such as NDepend, SonarQube, and Roslynator provide static analysis capabilities for C# source code, none integrates all three evaluation dimensions—correctness, structural quality, and runtime efficiency—into a unified automated benchmarking harness targeted at LLM outputs. Practitioners working in .NET environments are therefore unable to draw reliable, ecosystem-specific conclusions from existing benchmark results.

This paper addresses both limitations by introducing a fully automated end-to-end C# evaluation framework applied to GPT, Gemini, Claude, and Grok. The framework operates in four sequential phases (dataset preparation, code generation, dynamic evaluation, and result collection), integrating functional correctness testing via .NET Reflection, Roslyn-based[1] AST static analysis, and adversarial BenchmarkDotNet [10] runtime profiling into a unified harness, with all results consolidated into a structured dataset for cross-dimensional analysis. Our central contribution is demonstrating, through this framework, that functional correctness and structural code quality are empirically orthogonal—a finding that directly challenges the sufficiency of Pass@k as the sole evaluation criterion for LLMs in software engineering contexts.

2. Related Work

Code generation evaluation has evolved considerably since the introduction of early function-level benchmarks, expanding from simple correctness metrics toward richer assessments of repository-level complexity, evaluation integrity, and code quality. This section situates our work within these three threads of the literature.

HumanEval [5] and MBPP [6] established Pass@k as the dominant metric for LLM code generation, providing a reproducible and well-defined correctness signal. However, more recent work argues that isolated function generation does not reflect real-world complexity. DevEval [11] introduced 2,690 samples drawn from 119 real-world projects, while HumanEvo [12] demonstrated that existing evaluation methods substantially overestimate LLM performance at repository level, with performance overestimations ranging from 10% to 61% when evolution-aware evaluation is applied. Jiao et al. [13] demonstrated that existing benchmarks are insufficient to evaluate LLMs across different code translation complexity levels, and Paul et al. [9] provided a critical review confirming that current metrics fail to capture the full capability spectrum of LLMs in code generation. Despite these advances, C# and the .NET ecosystem remain substantially under-represented across all existing benchmark suites.

A parallel line of work has focused on evaluation integrity. Dou et al. [14] showed that LLM-generated code frequently contains structural and logical deficiencies beyond simple correctness failures, with models tending to produce shorter yet more complex code compared to canonical solutions—a pattern that already suggests correctness and structural quality need not co-vary. Zhang et al. [15] demonstrated that hallucinations and contextual dependency failures are a primary source of runtime errors in LLM-generated code, a finding that directly motivates our decision to build a full .NET solution structure that resolves project-level dependencies during dynamic compilation.

Closest to our quality-measurement objectives, TASTY [16] predicts algorithmic complexity of C++ and Python code using a Transformer classifier. However, it predicts complexity classes rather than empirically measuring them, and does not assess runtime efficiency or naming quality. To our knowledge, no prior work integrates runtime benchmarking, static analysis, and functional correctness testing into a single automated pipeline for the .NET ecosystem.

3. Methodology and Framework Architecture

This section describes the design and implementation of the evaluation framework in full detail. The framework assesses LLM-generated C# code across three independent quality dimensions: (1) *functional*

[1] Microsoft Roslyn is the open-source .NET compiler platform that exposes a full API for code analysis and transformation. Available at: https://github.com/dotnet/roslyn

correctness, measuring whether generated solutions pass all unit tests; (2) *static code quality*, capturing structural and stylistic properties via AST analysis; and (3) *runtime efficiency*, measuring execution time and memory allocation under adversarial profiling. The research questions below are organized around these three dimensions.

3.1. Research Questions

To structure our empirical investigation, we define the following research questions:

RQ1 (Correctness): How does functional correctness compare across GPT, Gemini, Claude, and Grok when generating C# solutions?

RQ2 (Task Complexity): To what extent does inherent task difficulty, rather than model-specific behavior, drive the structural complexity (cyclomatic complexity and nesting depth) of generated code?

RQ3 (Orthogonality): Is there a statistically significant relationship between functional correctness and composite code quality, or are these independent evaluation dimensions?

RQ4 (Performance): How do LLM-generated C# solutions compare in runtime execution efficiency and memory allocation under adversarial BenchmarkDotNet profiling?

3.2. Evaluation Metrics

The metrics were selected to directly address each research question. For RQ1, functional correctness is the Success Rate (%), i.e., the proportion of unit tests passed per solution. For RQ2, cross-model pairwise Pearson correlations on cyclomatic complexity and nesting depth across identical tasks quantify how much task difficulty—rather than model behavior—drives structural complexity [17, 18]. For RQ4, BenchmarkDotNet [10] yields median Execution Time (ns) and heap Memory Allocation (bytes/operation). For RQ3, a Composite Quality Score aggregates five normalized sub-components (correctness, complexity, nesting, style, and performance) into a single 0–100 scale value (formula defined in Phase 4).

3.3. The Evaluation Framework

The framework was built as a fully automated, end-to-end pipeline that takes a benchmark dataset as input and produces a structured multi-dimensional evaluation result for given LLMs under study. The pipeline consists of four major phases, namely dataset preparation, code generation, dynamic evaluation, and result collection. The overall architecture is illustrated in Figure 1.

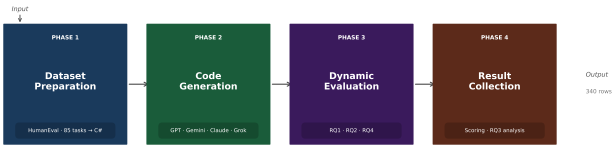


Figure 1: High-level overview of the four-phase evaluation pipeline and their mapping to RQ1–RQ4.

Phase 1: Data Preparation. Each HumanEval [5] record contains five fields — `task_id`, `description`, `prompt`, `entry_point`, and `test` — transformed into a structured Excel spreadsheet (one authoritative row per problem). A representative record is shown in Table 1; the `test` column shows a single representative assertion for brevity—each task contains multiple assertions from the original HumanEval test suite.

Two additional columns were computed per problem. The `solution_prompt` column is the field submitted to each LLM API on every individual run; it embeds the task fields into an engineered system

Table 1
Representative prepared record for task `csharp_1`.

Field	Content
`task_id`	`csharp/1`
`description`	Given two strings s and t, return true if t is an anagram of s
`prompt`	`public static bool IsAnagram(string s, string t)`
`entry_point`	`IsAnagram`
`test`	`Assert.AreEqual(true, Solver.IsAnagram("anagram","nagaram"))`
`solution_prompt`	Role: Expert C# Developer, class Solver, match signature, no namespace
`benchmark_prompt`	BenchmarkRunner, worst-case anagram inputs, N = 100/1,000/10,000

prompt that enforces a `public class Solver` output structure, prohibiting namespace declarations and `Main` methods injected later by the engine—any structural deviation would break automated compilation [19]. The `benchmark_prompt` generates a BenchmarkDotNet harness using worst-case inputs designed to prevent early-exit optimizations [20, 10]. Deliberately, harnesses were generated by an independent DeepSeek [21] instance rather than any evaluated model, preventing input bias and ensuring all four solvers were profiled under identical conditions [22, 23]. The prepared dataset is illustrated in Figure 2.

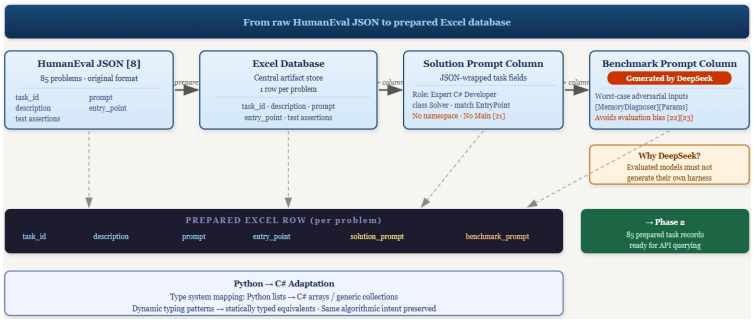


Figure 2: Phase 1: Data Preparation – HumanEval records transformed into an Excel database.

Phase 2: Code Generation. The framework queried all four LLMs in parallel via their respective APIs, submitting the `solution_prompt` for each of the 85 tasks at temperature = 0 to enforce deterministic decoding and eliminate sampling variance, consistent with the Pass@1 evaluation setting [5, 6]. Models are detailed in Section 4.1. All four outputs per task are compiled into a shared .NET 8.0 project comprising three files: the generated solutions (each `Solver` class isolated in its own namespace), the benchmark harness, and a test runner with a 5,000 ms timeout, as illustrated in Figure 3. Since LLM outputs frequently contain formatting inconsistencies that cause regex-based parsers to extract malformed method bodies [19, 15], the framework implements the ExtractMethodBodyRobust algorithm, which uses brace-counting to correctly locate method boundaries regardless of nesting depth or formatting variance.

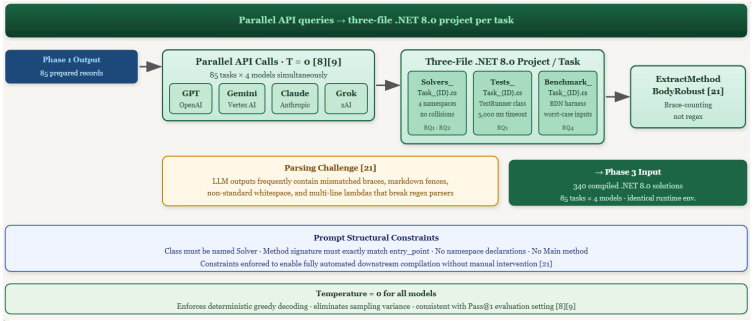


Figure 3: Phase 2: Code Generation – parallel API queries assembled into a shared three-file .NET 8.0 project per task.

Phase 3: Evaluation. The evaluation is carried out in three passes, namely functional correctness testing, static code quality analysis, and runtime performance profiling, each addressing a dedicated

research question as defined in Section 3.1.

1. Correctness. Each solver was invoked via .NET Reflection (model-specific `Run_{AI}` method), executing the full set of unit assertions from the HumanEval test field [5]. A thread-local score accumulator computes the Success Rate (%) as the proportion of assertions passed. Deep equality via CompareNetObjects handles complex return types (lists, arrays) to avoid false negatives.

2. Static Quality. The second pass assessed static code quality (addressing RQ2) using Microsoft Roslyn, which exposes each solution's full Abstract Syntax Tree for programmatic inspection. Four metrics are extracted: *Cyclomatic Complexity* [17], counting linearly independent execution paths (all `if`, `while`, `for`, `switch`, `&&`, `||` constructs), where higher values indicate harder-to-maintain code; *Nesting Depth*, the maximum depth of nested control structures, correlated with cognitive load [17, 18]; *Naming Violations*, counting deviations from standard C# casing conventions (PascalCase for public members, camelCase for locals),[2] which were not stated in the prompt—making this metric a measure of implicit convention adherence rather than instruction-following; and a *Professionalism Score* capturing XML documentation comments and explicit null-checks expected in production C# codebases.

3. Runtime Performance. The third evaluation pass (addressing RQ4) subjected all compiled solutions to runtime profiling using BenchmarkDotNet, one of the most widely used .NET microbenchmarking libraries. All benchmark methods (85 tasks $\times$ 4 models) were executed in a single joined run. For each task, the `BenchmarkRunner` class constructed during the preparation phase was executed across three input sizes (N = 100, 1,000, and 10,000) using the worst-case data generated by DeepSeek, ensuring that all execution paths within each solver were fully exercised. Two performance metrics were recorded per model per task: *median Execution Time* in nanoseconds and *total heap Memory Allocation* in bytes per operation. The large iteration count is necessary to produce stable, low-variance estimates and to amortize the overhead of .NET's JIT compilation across measurements [20, 24].

Phase 4: Result Collection and Composite Scoring. Upon completion of all three evaluation passes, the results for all 340 solution–model pairs (85 tasks $\times$ 4 models) were consolidated into a structured Excel file, with one row per pair and the following columns: Task ID, AI Model, Correctness (%), Quality Score, Time (ns), Memory (Bytes), Complexity, Nesting, and Naming Violations.

To enable the cross-dimensional analysis required by RQ3, a Composite Quality Score was also computed for each row, aggregating five normalized sub-components into a single value on a 0–100 scale:

$$\text{Composite Quality Score} = \frac{\text{Norm_C} + \text{Norm_Cx} + \text{Norm_N} + \text{Norm_S} + \text{Norm_P}}{5} \times 100 \quad (1)$$

where **Norm_C** = normalized correctness, **Norm_Cx** = normalized cyclomatic complexity (inverted), **Norm_N** = normalized nesting depth (inverted), **Norm_S** = normalized naming/style score, and **Norm_P** = normalized professionalism score.

Complexity and Nesting are inversely normalized so that simpler, flatter code scores higher, consistent with established software quality principles [17, 18, 11, 13, 14]. Performance is normalized against a 10,000 ns baseline using a decay curve that penalizes disproportionately slow solutions [13, 9]. All five components are weighted equally [11, 14]; we acknowledge that the equal weighting and decay curve parameters are design choices that may not generalize, and calibration via expert study is identified as future work.

4. Experimental Results

This section presents the experimental results of the evaluation of four well-known LLMs, namely GPT-4 [25], Gemini 1.5 Pro [26], Claude 3.5 Sonnet [27], and Grok 3 [28] (OpenAI, Google, Anthropic, and xAI respectively), applied to the C# code generation framework described in Section 3.

[2] Microsoft C# Coding Conventions: https://learn.microsoft.com/en-us/dotnet/csharp/fundamentals/coding-style/coding-conventions

4.1. Experimental Setup

All experiments were conducted on a Windows 11 machine equipped with an AMD Ryzen 9 5900X processor and 16 GB RAM, running the .NET 8.0 SDK. The four models evaluated were GPT-4 (OpenAI API), Gemini 1.5 Pro (Google Vertex AI), Claude 3.5 Sonnet (Anthropic API), and Grok 3 (xAI API). All models were queried with API temperature set to 0 to enforce deterministic decoding, consistent with the Pass@1 evaluation setting described in Section 3; all four providers document greedy decoding at temperature = 0 as deterministic for a fixed model version, which mitigates, though does not entirely eliminate, run-to-run variance [5, 6]. BenchmarkDotNet harnesses were generated independently using DeepSeek-V3 [21], deliberately excluded from the evaluation pool to prevent bias in worst-case profiling inputs [22, 23]. Runtime profiling was performed using BenchmarkDotNet across three input sizes ($N = 100, 1,000$, and $10,000$) per task per model to ensure stable, low-variance performance estimates [10, 20, 24].

4.2. Results Overview

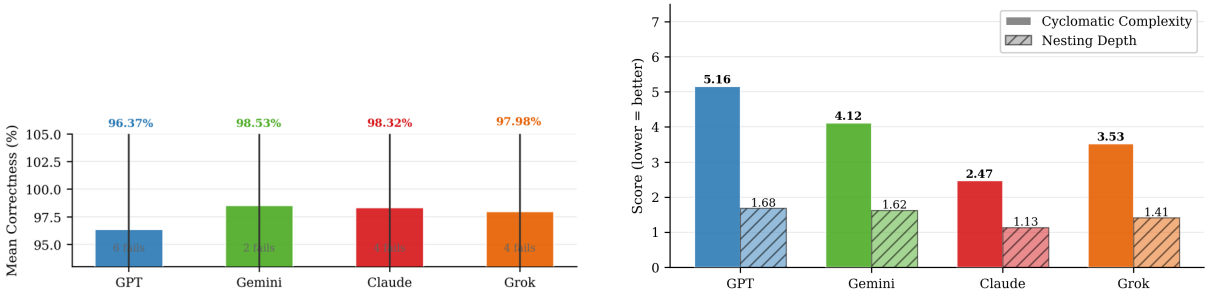
Table 2 summarises per-model outcomes across all metrics. RQ1: Sec. 4.3; RQ2: Sec. 4.4; RQ4: Sec. 4.5. RQ3 is addressed jointly: ranking reversal evidence in Sec. 5.2; full Pearson analysis in Sec. 5.1.

Table 2
Per-model summary (Med. time and memory used, see Sec. 4.5).

Model	Correct.	Quality	Cmplx.	Nest.	Time (ns)	Mem. (MB)	Fails
GPT	96.37	38.31	5.16	1.68	4,823	11.7	6
Gemini	98.53	40.05	4.12	1.62	9,034	28.9	2
Claude	98.32	37.69	2.47	1.13	12,859	156.6	4
Grok	97.98	40.80	3.53	1.41	9,136	6.0	4

4.3. Functional Correctness (RQ1)

Addressing RQ1, Figure 4a shows mean correctness with standard deviation error bars and failure counts. All four models achieve a median correctness of 100%, but meaningful differences emerge in mean and variance. Gemini leads at 98.53% (2 failures), followed by Claude at 98.32% (4 failures), Grok at 97.98% (4 failures), and GPT at 96.37% (6 failures) with the highest variance (SD = 15.37 vs Claude's 9.46), indicating a bimodal tendency toward fully correct or substantially failed solutions. Two tasks were universally challenging: `csharp_57` exposed a shared model limitation (Gemini scored 0%), and `csharp_64` produced an identical 75.0% across all four models, suggesting shared training data overlap.



(a) Mean Correctness (%) with ±1 SD error bars and failure counts per model.

(b) Cyclomatic Complexity and Nesting Depth per model. Claude generates the simplest code (2.47 vs GPT's 5.16).

Figure 4: Evaluation of model performance: Correctness rates and code complexity metrics.

4.4. Task Complexity and Model Behavior (RQ2)

Addressing RQ2, Figure 4b shows cyclomatic complexity and nesting depth across models. Claude generates the structurally simplest code (complexity 2.47, nesting 1.13): its cyclomatic complexity is approximately half that of GPT (5.16), and its nesting depth is roughly one third lower (1.13 vs. 1.68). However, the more consequential finding concerns the source of these differences. Cross-model

complexity agreement on identical tasks yields Pearson correlations of $r = 0.63$ to 0.69 across all model pairs, indicating that a substantial portion of the complexity observed in any model's output is attributable to the inherent difficulty of the task rather than to model-specific generation behavior. Problems requiring recursive logic or multi-branch control force elevated complexity across all four models simultaneously.

4.5. Runtime Performance (RQ4)

Addressing RQ4, Figure 5 shows median execution time and median memory allocation. Median values are used rather than means due to extreme outliers in Claude's data. **GPT** is the fastest at median (4,823 ns). Memory allocation varies substantially across models: **Grok** allocates the least (6.0 MB), followed by **GPT** (11.7 MB), **Gemini** (28.9 MB), and **Claude** (156.6 MB)—the latter being roughly 26× higher than Grok and 13× higher than GPT. **Claude**'s median execution time (12,859 ns) and memory allocation are both the highest, driven primarily by two anomalous tasks identified during profiling: `csharp_12`, in which Claude's execution time reached approximately 2.56×10^{11} ns—roughly 2,500× slower than the next worst model on the same task—and `csharp_16`, in which Claude allocated approximately 3.94 GB of heap memory while all other models allocated near zero, suggesting a fundamentally different and pathological algorithmic strategy for these specific tasks.

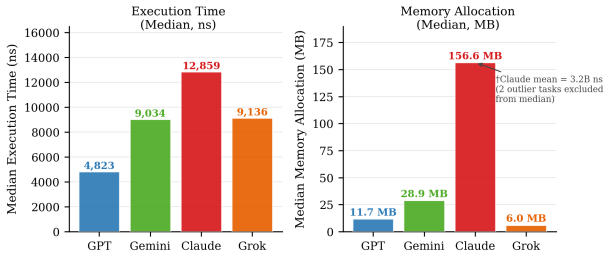


Figure 5: Median execution time (left) and memory allocation (right). Median used due to outliers in Claude's data.

5. Discussion

5.1. Inter-Metric Correlation Analysis

Figure 6 presents the full Pearson correlation matrix [29] across all 340 records. Two off-diagonal cells are highlighted as the key findings of this study: the Complexity–Nesting cell ($r = 0.706$) and the Correctness–Quality cell ($r = 0.075$).

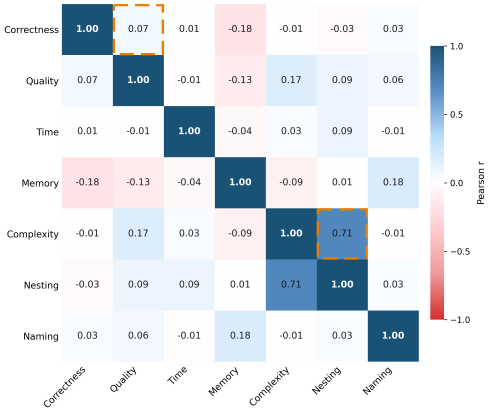


Figure 6: Pearson Correlation Matrix ($n = 340$): Complexity–Nesting $r = 0.706$; Correctness–Quality $r = 0.075$.

The strongest pairwise correlation is Complexity–Nesting ($r = 0.706$), confirming that these structural metrics are not independent, as models generating complex logic also produce deeply nested code. More consequentially for benchmark design, Correctness is near-zero correlated with all other

metrics ($|r| < 0.18$ throughout), including Quality Score ($r = 0.075$). We note that because Norm_C (normalized correctness) is itself one of the five components of the Composite Quality Score (Equation 1), the reported $r = 0.075$ specifically reflects the near-zero association between raw correctness and the four remaining structural and performance components, which together pull against any upward bias. This orthogonality is the central empirical result: correctness and structural quality must be evaluated as separate, independent dimensions.

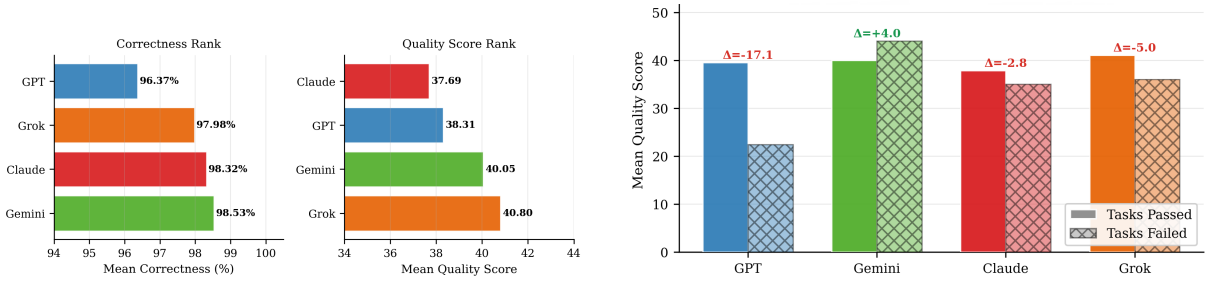
5.2. The Correctness–Quality Orthogonality Problem

The near-zero correlation ($r = 0.075$) means Pass@k rankings are statistically uninformative about code quality. Prior work has implicitly assumed that a model capable of generating functionally correct code would also tend to produce cleaner, more maintainable solutions [14]—an expectation grounded in the view that both correctness and quality reflect general coding competence. Our results empirically disconfirm this assumption: a model atop a correctness leaderboard may produce code harder to maintain, more deeply nested, and more complex than a lower-ranked competitor. For SE practitioners deploying LLMs in production, technical debt from AI-generated code accumulates incrementally with real engineering cost [3, 14]. Evaluation frameworks must treat correctness and quality as distinct, co-equal dimensions. Figure 7a illustrates this reversal directly, comparing model rankings on correctness against rankings on quality score.

> **Answer to RQ3:** Functional correctness and composite code quality are empirically orthogonal (Pearson $r = 0.075$, $n = 340$) within this benchmark (85 C# HumanEval tasks, four LLMs). The relationship is not statistically significant. This confirms that Pass@k rankings are insufficient as a sole evaluation criterion for LLMs in software engineering contexts; correctness and structural quality must be reported as independent dimensions. Whether this orthogonality generalises to other programming languages or task types remains an open question for future work.

5.3. GPT's Bimodal Failure Behavior

Figure 7b stratifies Quality Score by task outcome, revealing GPT's pronounced quality collapse on failed tasks. GPT's quality on failed tasks (22.44) is less than 57% of its quality on successful tasks (39.51), and just over half of Gemini's quality on failures (44.00). When GPT fails a task, the generated code exhibits broad structural deficiencies alongside the functional failure, representing a systemic rather than marginal breakdown. The other three models degrade more gracefully.



(a) Model rankings by Correctness vs. Quality Score, showing complete reversal ($r = 0.075$).

(b) Quality Score on passed vs. failed tasks. GPT collapses 17.1 points on failures.

Figure 7: Overall comparison of model rankings and bimodal quality scores.

> **Answer to RQ1:** Gemini achieves the highest mean correctness (98.53%, 2 failures) and is the most reliable model on this benchmark. Claude (98.32%) and Grok (97.98%) follow closely. GPT records the lowest mean (96.37%) and the highest variance (SD = 15.37), indicating a bimodal distribution between fully correct and substantially failed solutions. All models converge on identical partial solutions for two tasks (`csharp_57`, `csharp_64`), suggesting shared limitations across model families.

5.4. Task Difficulty as a Complexity Confound

Cross-model complexity agreement on identical tasks ($r = 0.63$ to 0.69 across model pairs) shows that inherent task difficulty, rather than model-specific behavior, is the primary driver of cyclomatic complexity. Problems requiring recursive logic or multi-branch control force elevated complexity across all models. Benchmarks comparing raw complexity across heterogeneous task sets without controlling for task difficulty will produce biased results. Practitioners and benchmark designers should therefore compare structural complexity metrics only within the same task, or normalise against a task-level baseline.

> **Answer to RQ2:** The cross-model complexity correlation analysis ($r = 0.63$–0.69 across all model pairs on identical tasks) confirms that inherent task difficulty is the primary driver of structural complexity in LLM-generated code, with model-specific differences being secondary. Benchmarks that compare raw cyclomatic complexity or nesting depth across models without controlling for task-level difficulty will produce biased conclusions.

5.5. Holistic Performance Profiles

Figure 8 provides a normalized radar view across all six evaluation dimensions simultaneously, making each model's trade-off profile immediately visible. No model dominates across all dimensions. Grok has the most balanced overall profile. Practitioners should select models based on the quality dimension their context prioritizes, rather than a single composite rank.

A noteworthy trade-off emerges from Claude's results: despite generating the structurally simplest code (complexity 2.47, nesting 1.13), Claude incurs the highest memory footprint (156.6 MB median). This illustrates that low structural complexity does not imply runtime efficiency—simpler control flow can be achieved through data-intensive algorithmic strategies that trade branch depth for memory cost. Correctness, structural complexity, and runtime efficiency should therefore be treated as complementary rather than redundant evaluation dimensions.

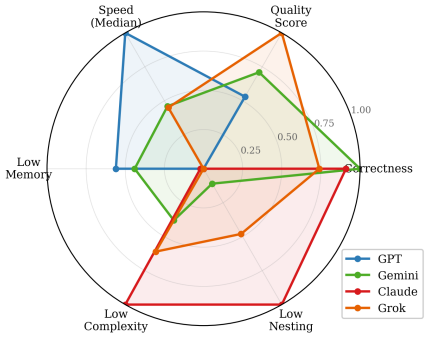


Figure 8: Normalized multi-dimensional radar across all six dimensions. Grok presents the most balanced profile.

> **Answer to RQ4:** GPT is the fastest model (median 4,823 ns); memory profile: Grok 6.0 MB, GPT 11.7 MB, Gemini 28.9 MB, Claude 156.6 MB. Claude's outlier values in both dimensions are driven primarily by two anomalous tasks (`csharp_12` and `csharp_16`). All models remain within one order of magnitude on median execution time.

6. Threats to Validity

We structure threats to validity following Wohlin et al. [30].

Internal validity. LLM API non-determinism was mitigated by setting temperature = 0; all four providers document greedy decoding as deterministic for a fixed model version. Multi-run variance analysis under non-zero temperature is identified as future work. BenchmarkDotNet outliers in Claude's

runtime data were addressed by adopting the median as the primary performance metric, with anomalous tasks (`csharp_12`, `csharp_16`) explicitly identified.

Construct validity. The equal weighting of the five Composite Quality Score sub-components may not reflect real-world priorities; all decomposed metrics are reported individually to allow independent interpretation. Since Norm_C is one of the five components, the $r = 0.075$ result is partly self-referential, specifically reflecting the near-zero association between raw correctness and the four remaining structural and performance components; a robustness check excluding Norm_C is identified as future work. The naming and professionalism metrics capture implicit C# convention adherence rather than instruction-following, as style constraints were not included in the solution prompt; investigating explicit style injection is also identified as future work.

External validity. The 85-task HumanEval subset does not reflect complex, multi-file enterprise .NET architectures, but provides a necessary controlled baseline. The Python-to-C# adaptation preserves algorithmic intent; potential systematic difficulty bias is acknowledged as an additional concern. The orthogonality finding (RQ3) should not be assumed to generalise unconditionally to other languages or task types. The framework is dataset-agnostic; applying it to enterprise repositories is a key future direction.

Conclusion validity. The 85-task dataset may limit statistical power; this was mitigated by performing the Pearson correlation analysis over all 340 solution records and reporting standard deviations throughout.

7. Conclusion and Future Work

This paper presents an automated, multi-dimensional evaluation framework for C# code generation. Our central contribution is demonstrating a systematic gap between an LLM's ability to generate functionally correct code and its capacity to produce structurally sound, maintainable, and efficient code. By confirming that correctness and quality are virtually orthogonal ($r = 0.075$, within this C# HumanEval benchmark), our findings directly challenge the sufficiency of Pass@k benchmarks for selecting models in production software engineering.

Evaluation of four state-of-the-art LLMs revealed distinct performance profiles: Gemini leads in functional reliability (98.53% correctness); Grok achieves the highest, most balanced composite quality score; Claude generates the structurally simplest code but incurs the highest memory footprint (156.6 MB median), illustrating that structural simplicity and runtime efficiency are independent properties; and GPT writes the fastest-executing code but exhibits the highest structural complexity and a severe, bimodal quality collapse on failed tasks. Furthermore, identical partial solutions across all models suggest shared biases in their underlying training distributions.

Future work will expand this framework to enterprise-representative C# patterns (e.g., asynchronous programming, LINQ pipelines), empirically calibrate Quality Score weights via expert studies, investigate prompt-injected style and naming constraints, conduct multi-run variance analysis, and validate the RQ3 orthogonality finding with a four-component quality score that excludes correctness.

Replication. The evaluation harness, adapted task specifications, and raw results dataset are publicly available on GitHub.[3] All experiments are fully reproducible given the documented hardware configuration and API temperature settings.

Declaration on Generative AI

During the preparation of this work, the author(s) used Claude for writing assistance, grammar and spelling correction, and LaTeX formatting. The author(s) reviewed and edited the content as needed and take full responsibility for the publication's content.

[3] https://github.com/mohammadmehdighalandarian/llm4se.git

References

[1] A. Svyatkovskiy, S. K. Deng, S. Fu, N. Sundaresan, IntelliCode Compose: Code Generation Using Transformer, in: Proceedings of the 28th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE), ACM, 2020, pp. 1433–1443. doi:10.1145/3368089.3417058.

[2] A. Ziegler, E. Kalliamvakou, X. A. Li, A. Rice, D. Rifkin, S. Simister, G. Sittampalam, E. Aftandilian, Productivity Assessment of Neural Code Completion, in: Proceedings of the 6th ACM SIGPLAN International Symposium on Machine Programming (MAPS), ACM, 2022, pp. 21–29.

[3] N. Perry, M. Srivastava, D. Kumar, D. Boneh, Do Users Write More Insecure Code with AI Assistants?, in: Proceedings of the ACM SIGSAC Conference on Computer and Communications Security (CCS), ACM, 2023, pp. 2785–2799.

[4] H. Pearce, B. Ahmad, B. Tan, B. Dolan-Gavitt, R. Karri, Asleep at the Keyboard? Assessing the Security of GitHub Copilot's Code Contributions, in: Proceedings of the IEEE Symposium on Security and Privacy (S&P), IEEE, 2022, pp. 754–768.

[5] M. Chen, J. Tworek, H. Jun, Q. Yuan, H. P. d. O. Pinto, J. Kaplan, H. Edwards, Y. Burda, N. Joseph, G. Brockman, et al., Evaluating Large Language Models Trained on Code, arXiv preprint arXiv:2107.03374 (2021).

[6] J. Austin, A. Odena, M. Nye, M. Bosma, H. Michalewski, D. Dohan, E. Jiang, C. Cai, M. Terry, Q. Le, et al., Program Synthesis with Large Language Models, arXiv preprint arXiv:2108.07732 (2021).

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

[8] Q. Zheng, X. Xia, X. Zou, Y. Dong, S. Wang, Y. Xue, Z. Wang, L. Shen, A. Wang, Y. Li, et al., CodeGeeX: A Pre-Trained Model for Code Generation with Multilingual Benchmarking on HumanEval-X, in: Proceedings of the 29th ACM SIGKDD Conference on Knowledge Discovery and Data Mining, ACM, 2023, pp. 5673–5684.

[9] D. G. Paul, H. Zhu, I. Bayley, Benchmarks and Metrics for Evaluations of Code Generation: A Critical Review, in: Proceedings of the IEEE International Conference on Artificial Intelligence Testing (AITest), IEEE, 2024, pp. 87–94. doi:10.1109/AITest62860.2024.00019.

[10] A. Akinshin, Pro .NET Benchmarking: The Art of Performance Measurement, Apress, New York, NY, USA, 2019.

[11] J. Lian, et al., DevEval: A Manually-Annotated Code Generation Benchmark Aligning with Real-World Code Repositories, in: Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (ACL), ACL, 2024.

[12] D. Zheng, Y. Wang, E. Shi, R. Zhang, Y. Ma, H. Zhang, Z. Zheng, HumanEvo: An Evolution-Aware Benchmark for More Realistic Evaluation of Repository-Level Code Generation, in: Proceedings of the 47th IEEE/ACM International Conference on Software Engineering (ICSE), IEEE/ACM, 2025, pp. 1372–1384. doi:10.1109/ICSE55347.2025.00228.

[13] M. Jiao, T. Yu, X. Li, G. Qiu, B. Gu, B. Shen, On the Evaluation of Neural Code Translation: Taxonomy and Benchmark, in: Proceedings of the 38th IEEE/ACM International Conference on Automated Software Engineering (ASE), IEEE/ACM, 2023, pp. 1529–1541. doi:10.1109/ASE56229.2023.00114.

[14] S. Dou, H. Jia, S. Wu, H. Zheng, W. Zhou, M. Wu, M. Chai, J. Fan, C. Huang, Y. Tao, Y. Liu, E. Zhou, M. Zhang, Y. Zhou, Y. Wu, R. Zheng, M. Wen, R. Weng, J. Wang, X. Cai, T. Gui, X. Qiu, Q. Zhang, X. Huang, What Is Wrong with Your Code Generated by Large Language Models? An Extensive Study, Science China Information Sciences 69 (2025). doi:10.1007/s11432-025-4632-8.

[15] Z. Zhang, C. Wang, Y. Wang, E. Shi, Y. Ma, W. Zhong, J. Chen, M. Mao, Z. Zheng, LLM Hallucinations in Practical Code Generation: Phenomena, Mechanism, and Mitigation, Proceedings of the ACM on Software Engineering 2 (2024) 481–503. doi:10.1145/3728894.

[16] K. Moudgalya, A. Ramakrishnan, V. Chemudupati, X. H. Lu, TASTY: A Transformer based Approach to Space and Time Complexity, arXiv preprint arXiv:2305.05379 (2023). doi:10.48550/arXiv.2305.05379.

[17] T. J. McCabe, A Complexity Measure, IEEE Transactions on Software Engineering SE-2 (1976) 308–320.

[18] T. J. McCabe, C. W. Butler, Design Complexity Measurement and Testing, Communications of the ACM 32 (1989) 1415–1425. doi:10.1145/76380.76382.

[19] Q. Huang, J. Sun, X. Xu, Q. Hu, J. Liu, X. Luo, An Empirical Study on Challenges and Issues of Code Generation with Large Language Models, arXiv preprint arXiv:2310.06218 (2023).

[20] T. Kalibera, R. Jones, Rigorous Benchmarking in Reasonable Time, in: Proceedings of the ACM SIGPLAN International Symposium on Memory Management (ISMM), ACM, 2013, pp. 63–74.

[21] DeepSeek-AI, DeepSeek-V3 Technical Report, arXiv preprint arXiv:2412.19437 (2024). doi:10.48550/arXiv.2412.19437.

[22] P. Liang, R. Bommasani, T. Lee, D. Tsipras, D. Soylu, M. Yasunaga, Y. Zhang, D. Narayanan, Y. Wu, A. Kumar, et al., Holistic Evaluation of Language Models, arXiv preprint arXiv:2211.09110 (2022).

[23] Y. Chang, X. Wang, J. Wang, Y. Wu, L. Yang, K. Zhu, H. Chen, X. Yi, C. Wang, Y. Wang, et al., A Survey on Evaluation of Large Language Models, ACM Transactions on Intelligent Systems and Technology 15 (2024) 1–45.

[24] A. Georges, D. Buytaert, L. Eeckhout, Statistically Rigorous Java Performance Evaluation, in: Proceedings of the 22nd ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA), ACM, 2007, pp. 57–76.

[25] OpenAI, GPT-4 Technical Report, arXiv preprint arXiv:2303.08774 (2023). doi:10.48550/arXiv.2303.08774.

[26] Gemini Team, Gemini 1.5: Unlocking Multimodal Understanding Across Millions of Tokens of Context, arXiv preprint arXiv:2403.05530 (2024). doi:10.48550/arXiv.2403.05530.

[27] Anthropic, Claude 3.5 Sonnet Model Card, Technical Report, Anthropic, 2024. URL: https://www.anthropic.com/claude/sonnet.

[28] xAI, Grok-3 Technical Report, Technical Report, xAI, 2025. URL: https://x.ai/news/grok-3.

[29] B. A. Kitchenham, S. L. Pfleeger, L. M. Pickard, P. W. Jones, D. C. Hoaglin, K. El Emam, J. Rosenberg, Preliminary Guidelines for Empirical Research in Software Engineering, IEEE Transactions on Software Engineering 28 (2002) 721–734.

[30] C. Wohlin, P. Runeson, M. Höst, M. C. Ohlsson, B. Regnell, A. Wesslén, Experimentation in Software Engineering, Springer, Berlin, Heidelberg, 2012.