DOI: https://doi.org/10.15276/aait.09.2026.20
UDC 004.42:004.8

Research on the quality of automated program code generation using neural networks

Viktor O. Speranskyy[1)]
ORCID: https://orcid.org/0000-0002-8042-1790; speranskyy@op.edu.ua. Scopus Author ID: 54401618900
Hlib E. Maksymenko[1)]
ORCID: https://orcid.org/0009-0002-9043-5376; maksimenkogleb@stud.op.edu.ua
Anastasiya Gromova[2)]
ORCID: https://orcid.org/0009-0001-6492-9172; anastasiya.gromova1@louisiana.edu
[1)] Odesa Polytechnic National University, 1, Shevchenko Avenue. Odesa, 65044, Ukraine
[2)] University of Louisiana at Lafayette, 104, E. University Circle. Lafayette, LA 70503, USA

ABSTRACT

Relevance. Rapid advancements in large language models have significantly impacted software engineering, necessitating a rigorous evaluation of their capabilities in automated code generation. **The aim of the study** is to develop and validate a methodology for comprehensive, multi-dimensional quality assessment of program code generated by modern large language models in production-grade environments. **Objectives.** The study conducts a comparative analysis of the code generation quality produced by five leading neural networks – GPT-5, Claude Opus 4.1, Gemini 3.1 Pro, Grok 4, and DeepSeek-V3.1 – within the context of modern web development, evaluating their capacity to generate a production-ready, standalone Angular 19 component featuring complex drag-and-drop functionality, smooth animations, and mock Hypertext Transfer Protocol service integration. **Methods.** A two-level evaluation methodology was employed, combining automatic quantitative metrics – such as build correctness, TypeScript and ESLint error rates, cyclomatic complexity, bundle size, and security auditing – with a qualitative expert assessment of architectural integrity, maintainability, and documentation completeness. An Integrated Model Quality Assessment Metric was derived to rank the models based on weighted factors, prioritizing correctness (35%) and maintainability (30%) over performance (20%), documentation (10%), and security (5%). **Scientific novelty.** The proposed methodology integrates automated static analysis with structured expert evaluation into a single comparable metric grounded in the ISO/IEC 25010 quality framework, addressing the gap left by existing benchmarks that evaluate only functional correctness on isolated tasks. **Practical significance.** The findings provide crucial empirical data for selecting artificial intelligence tools in development workflows and demonstrate that production-oriented quality assessment requires multi-dimensional evaluation beyond syntactic correctness. **Results.** The empirical analysis revealed significant disparities across the tested architectures. Claude Opus 4.1 achieved the highest Integrated Quality Score (0.636), demonstrating superior code structure and documentation standards. Gemini 3.1 Pro followed closely (0.620), excelling in performance optimization and build stability. GPT-5 (0.447), while syntactically accurate, suffered from performance optimization issues, while DeepSeek-V3.1 (0.530) required substantial manual debugging. Grok 4 scored the lowest (0.240), with its reliance on outdated module-based architectures resulting in systemic deficiencies. **Conclusions.** While modern large language models are capable of generating valid code, substantial human oversight remains essential to ensure production readiness. The Integrated Quality Score proved effective in differentiating models whose surface-level syntactic performance appears similar.

Keywords: Large language models; automated code generation; software quality assessment; Angular framework; ISO/IEC 25010 quality framework; static code analysis; integrated quality score; software engineering benchmarks

***For citation*: Speranskyy V. O., Maksymenko H. E., Gromova A. "Research on the quality of automated program code generation using neural networks". *Applied Aspects of Information Technology*. 2026; Vol.9 No.3: 296–309. DOI: https://doi.org/10.15276/aait.09.2026.20**

INTRODUCTION

Automated code generation systems driven by large language models (LLMs) are undergoing rapid evolution, expanding their utility across both academic instruction and industrial software engineering [23], [24]. Contemporary architectures – including OpenAI's GPT, Anthropic's Claude, Google DeepMind's Gemini, xAI's Grok, and DeepSeek – demonstrate the capacity to synthesize syntactically valid source code [4], [17] and resolve intricate applied problems leveraging contextual awareness [21].

Notwithstanding these advancements, evaluating generated code quality remains complex; viable solutions must extend beyond mere syntactic correctness to ensure comprehensive alignment with functional specifications, runtime efficiency, fault tolerance, and seamless integration with legacy software infrastructure.

The primary objective of this study is to perform a multi-dimensional investigation into the quality of software components synthesized by these state-of-the-art neural architectures. This evaluation is executed by subjecting each model to an identical, concise zero-shot prompt, thereby ensuring an objective comparative analysis under standardized baseline conditions.

© Speranskyy V., Maksymenko H., Gromova A., 2026

This is an open access article under the CC BY license (https://creativecommons.org/licenses/by/4.0/deed.uk)

The evaluation framework encompasses six critical quality dimensions: syntactic and semantic correctness, performance efficiency, readability and structural architecture, solution versatility [15], and resilience to logical anomalies and security vulnerabilities [16], [18]. This paper delineates the specific operational proficiencies and architectural limitations of each model, culminating in actionable recommendations for the strategic integration of LLMs within software engineering workflows [19].

LITERATURE REVIEW AND PROBLEM STATEMENT

Large language models (LLMs) utilize transformer–based architectures pre–trained on extensive natural language and source code corpora, facilitating automated code synthesis, autocompletion, and program repair across diverse programming paradigms [20], [25]. The transition from specialized, code-centric utilities to contemporary general-purpose systems has outpaced the evolution of standardized evaluation frameworks, which frequently rely on isolated benchmarks rather than metrics reflecting industrial production readiness [4], [17]. Comprehensive literature surveys indicate that cross–model empirical studies conducted under rigorous, production–grade conditions remain sparse [23], [24]. Systematic reviews by Fan et al. [26] and Hou et al. [27] substantiate this deficiency, identifying multi–dimensional production–readiness assessment as a critical open challenge in software engineering.

Standardized benchmarks commonly deployed to evaluate code generation quality include HumanEval [4], MBPP [2], and APPS [8], the methodologies of which are detailed in [28]. The HumanEval dataset comprises 164 manually curated Python programming challenges and utilizes the pass@k metric to estimate the probability that at least one of k generated code variants satisfies the associated unit tests. The MBPP benchmark extends this paradigm by incorporating a broader array of crowd–sourced tasks evaluating standard library utilization and basic algorithmic reasoning [2]. Concurrently, the APPS framework scales task difficulty to encompass interview–grade and competitive programming challenges [8]. Although these benchmarks serve as the primary baseline for model verification, they possess a fundamental limitation: they measure functional correctness exclusively within isolated algorithmic scopes, offering no insights into critical production–grade attributes such as maintainability, architectural integrity, bundle efficiency, or security posture.

Emerging evaluation frameworks attempt to bridge the gap between synthetic environments and industrial software development; for instance, the SWE-bench framework requires models to resolve authentic GitHub issues derived from open-source repositories, validating whether a generated patch satisfies the target test suite without introducing regressions. This paradigm shift toward repository–level, multi-file assessment reflects a consensus that single-function code generation fails to replicate the inherent complexity of industrial software systems [11]. Furthermore, Olausson et al. [14] demonstrated that iterative self-repair mechanisms – wherein models evaluate and correct their own anomalous outputs – enhance execution success rates by 10 to 20 percentage points, though substantial limitations persist during complex reasoning sequences. These empirical insights underscore the necessity of evaluating generative models under conditions that mirror authentic development workflows, specifically incorporating automated build systems, static code linters and package dependency managers.

From a structural quality perspective, the ISO/IEC 25010 Systems and Software Quality Requirements and Evaluation (SQuaRE) framework codifies eight primary quality characteristics: functional suitability, performance efficiency, compatibility, usability, reliability, security, maintainability, and portability. Although heavily utilized in traditional software quality assurance, the systematic application of this standard to LLM–synthesized source code remains limited; however, static analysis metrics – such as McCabe cyclomatic complexity, the Maintainability Index [12], and Halstead complexity measures – provide a deterministic methodology to operationalize specific ISO/IEC 25010 sub-characteristics, predominantly within the domains of maintainability and performance efficiency. Jesse et al. [9] surveyed the intersection of generative language models and static analysis utilities, concluding that synthesizing automated metrics with empirical model evaluation yields significantly more actionable quality insights than either localized paradigm.

Recent literature has explored adaptive fuzzy–logic models to automate code quality assessment during software refactoring [29], alongside machine learning techniques designed to isolate the software metrics that exhibit the strongest correlation with system reliability [30]. Hodovychenko and Kurinko [31] compiled a comprehensive taxonomy of automated refactoring approaches for object–oriented software systems, classifying rule–based,

graph–based, learning–driven (encompassing convolutional neural networks, graph neural networks, and LLMs), and hybrid human–in–the–loop frameworks.

Security remains a critically under–explored dimension within the domain of generative code evaluation; an audit of GitHub Copilot by Pearce et al. [16] revealed that approximately 40 % of synthesized code blocks within security – sensitive contexts introduced vulnerabilities, including flaws cataloged in the OWASP Top Ten. Sandoval et al. [18] extended this assessment, demonstrating that while scaled model parameterization reduces apparent syntax flaws, subtle security risks tied to improper dependency management, inadequate input validation, and cryptographic misuse persist across diverse model families. These exposures necessitate the inclusion of deterministic security auditing – executed via npm audit and OWASP ZAP vulnerability scanning – as a core evaluation dimension in this study.

The integration of expert human evaluation to complement automated telemetry has been substantiated across several studies of computer–assisted programming; specifically, Kazemitabaar et al. [10] established that developer perceptions of structural quality frequently diverge from automated execution metrics, particularly regarding readability and architectural coherence. Furthermore, Mozannar et al. [13] modeled the cognitive overhead incurred by engineers during the peer review and correction of AI–generated source code, demonstrating that erroneous outputs from models boasting high nominal accuracy can introduce higher aggregate costs than outputs from less accurate models characterized by predictable failure modes. These conclusions justify the dual–level evaluation framework deployed in this study, which couples algorithmic static metric computation with a structured, multi-criteria expert assessment matrix.

The prominent general–purpose LLMs utilized for source code synthesis include OpenAI's GPT, Anthropic's Claude, Google DeepMind's Gemini, xAI's Grok, and DeepSeek; the GPT family exhibits robust zero-shot and few–shot generalization capabilities across a wide spectrum of syntax paradigms [15]. Concurrently, Claude prioritizes architectural safety, model interpretability, and extensive online documentation [1]; Gemini is architected for optimization within distributed data environments [5]; Grok emphasizes low–latency processing and practical software tasks [22]; and DeepSeek demonstrates highly competitive syntactic precision within compiled language domains [7]; despite extensive isolated documentation of these architectures, a unified, empirical comparative analysis evaluating all five models under identical, framework-specific, and production–grade constraints remains absent from the literature.

Consequently, each model presents an idiosyncratic performance profile across the vectors of correctness, maintainability, performance efficiency, documentation, and security; the lack of a unified, empirically validated metric capable of aggregating these disparate dimensions into a singular comparative index represents the core methodological gap addressed by this research; the proposed Integrated Model Quality Assessment Metric (IMQAM) is designed to augment existing functional benchmarks by introducing a production–centric evaluation paradigm grounded in static analysis telemetry, automated build validation, expert peer review, and the ISO/IEC 25010 structural framework.

AIMS AND OBJECTIVES OF THE STUDY

The critical analysis of contemporary literature underscores a significant limitation in existing benchmarks for large language model (LLMs) code generation-such as HumanEval, MBPP, APPS, and SWE-bench – which predominantly evaluate localized functional correctness on isolated tasks and fail to encapsulate multi-dimensional quality dimensions critical for production software systems, including maintainability, architectural soundness, operational efficiency, completeness of documentation, and security level. While the ISO/IEC 25010 framework offers a robust theoretical model for integrated software quality engineering, its systematic adaptation to LLM–generated source code remains largely unaddressed in published research. Furthermore, empirical, cross-model comparative evaluations executed under strictly controlled, production–grade conditions using identical high–complexity prompts are exceptionally rare.

Consequently, the overarching aim of this study is to formulate and empirically validate a comprehensive, multi-dimensional methodology for evaluating the structural and architectural quality of program code generated by modern large language models under realistic production environments. While recent benchmarking efforts have shifted toward repository-level text synthesis, isolating a complex standalone component provides an essential controlled baseline that eliminates confounding cross-file dependency noise. To fulfill this aim, the following specific objectives have been established:

- To formalize a comprehensive matrix of quality assessment criteria that synthesizes automated static analysis metrics with structured expert evaluations, grounded firmly within the ISO/IEC 25010 framework and operationalized via classical software metrics (including cyclomatic complexity, the Maintainability Index, Halstead metrics, build success verification, and security posture auditing).
- To design and execute a controlled software engineering experiment wherein five state-of-the-art LLMs – GPT-5, Claude Opus 4.1, Gemini 3.1 Pro, Grok 4, and DeepSeek-V3.1 – generate source code based on an identical, high-complexity Angular 19 prompt, thereby guaranteeing the strict comparative validity of the resulting outputs.
- To perform a multi-dimensional comparative analysis of the synthesized source code across five weighted quality vectors (correctness, maintainability, performance, documentation, and security) utilizing the proposed Integrated Model Quality Assessment Metric.
- To mathematically validate the proposed quality metric through a dual-level verification protocol: first, by assessing internal consistency through the correlation of automated metric rankings with independent expert evaluations, and second, by computing Kendall's coefficient of concordance (W) to rigorously confirm the objectivity and inter-rater reliability of the human evaluation panel.

CODE QUALITY MEASUREMENT METHODOLOGY

To guarantee an objective and reproducible comparative analysis of the source code synthesized by the selected language models, this study establishes a dual–level evaluation paradigm:

- Automated Metrics – quantitative telemetric parameters dynamically computed via compilers, linters, bundle analyzers, and static analysis tools.
- Human-Centric Evaluation – qualitative expert peer reviews assessing architectural integrity, functional correctness, and long-term maintainability.

This unified approach explicitly links quantitative operational data (such as syntax error rates, bundle footprint, and logical complexity) with qualitative assessments of code readability and structural health. To enforce maximum comparative reliability and evaluate the zero-shot architectural design capabilities of each model, the verbatim prompt deployed across all evaluated models was intentionally constrained to the following specification: "*generate a production-ready standalone Angular 19 component for a task board with: Drag-and-drop between columns (Todo/In Progress/Done) Smooth animations on card movement Optimized rendering Mock HTTP service*".

Automatic Metrics Analysis

Each generated software component was subjected to rigorous automated static and dynamic analysis across the following telemetry vectors:

- Build Correctness quantified as the empirical build success rate across all experimental iterations:

$$BuildSuccess = N_{successful} / N_{runs.} \quad (1)$$

- Syntactic and Linting Anomalies operationalized as the density of TypeScript compilation and ESLint errors normalized per thousand lines of code (LOC):

$$ErrorsPerKLOC = Errors / (LOC / 1000). \quad (2)$$

- Bundle size measured as the aggregate production build footprint in kilobytes via the Angular CLI deployment build pipeline (ng build --prod).
- External Package Dependencies (*Deps*): Defined as the total quantity of third–party packages explicitly introduced into the project configuration to support the component's functionality.
- Code Maintainability Metrics computed utilizing automated software complexity analyzers to isolate
  – LOC representing the total raw volume of generated source lines;
  – cyclomatic complexity (CC) Quantifying the logical density and control flow branch points of the algorithm structure via McCabe's formulation:

$$CC = \sum\nolimits^{N}_{i=1} (DecisionPoints_i+1),\ i = 1,..., N_{functions} \quad (3)$$

  – maintainability index (MI) is the composite standardized metric evaluating the structural maintainability of the codebase [12];

$$MI = 171 - 5.2\cdot\ln(H) - 0.23\cdot CC - 16.2\cdot\ln(LOC), \quad (4)$$

where H represents the Halstead volume metric derived from the total number of unique and total operators and operands, capturing code complexity and volume.

- Architectural Best Practices Score: Computed as a cumulative index of binary indicators (0 or 1) evaluating framework–specific optimization patterns, specifically the explicit utilization of the OnPush change detection strategy, the implementation of trackBy functions within template loops, and the complete avoidance of direct DOM

manipulations.

- Documentation Completeness: Evaluated via two distinct parameters:

– comment density (%) – the percentage of lines containing comments:

$$CommentDensity = (Lines_{comments}/LOC)\cdot 100, \quad (5)$$

– README completeness – presence of basic instructions for launching the project (evaluated as yes/no);

– security vulnerabilities count – number of vulnerabilities found through npm audit [16], [18].

Human Evaluation Analysis

The code was also evaluated manually according to 5 criteria (scale 0-5):

- functional correctness (drag-and-drop, animations, column operation);
- performance and optimization (no unnecessary redrawing, smoothness);
- architecture and maintainability (structure, modularity, DI);
- integration with mock HTTP service;
- overall code readability.

The raw expert ratings were compiled to compute an exact mathematical average for each model across all experimental iterations.

Expert group

Panel size: five experts.

Qualification: experienced Angular developers (≥3 years), familiar with best practices and development patterns.

Expert role: evaluation of each model's code according to 5 criteria (0-5): Functionality, Performance, Architecture, Integration, Readability.

Evaluation procedure

Each model generates code with an identical prompt. Experts receive the code in a standardized form (formatted using Prettier). Each metric is evaluated individually by each expert. To eliminate subjectivity, the assessments are reconciled.

The average rating for each criterion is calculated:

$$Score_{criterion} = (1/N_{experts})\cdot\sum Score_i,\ i=1..N_{experts}. \quad (6)$$

Test infrastructure

- Build system: Angular CLI 19.x.
- Linting: ESLint.
- Security scanning: npm audit, OWASP ZAP.
- Performance testing: Chrome DevTools, web-vitals library.

Integrated model quality assessment metric

The formula determines the model quality:

$$IMQAM = 0.35\cdot Correctness+0.30\cdot Maintainability+ 0.20\cdot Performance+0.10\cdot Documentation+ 0.05\cdot Security \quad (7)$$

Normalization procedure

Each component of the Integrated Quality Score (IQS) is normalized to the range [0, 1] using min-max normalization across the five tested models. For metrics where lower values indicate better quality (errors per KLOC, cyclomatic complexity, LOC, bundle size, dependencies, and vulnerabilities), the normalized value is computed as:

$$X_norm = 1 - (X - Min) / (Max - Min), \quad (8)$$

where X is the raw metric value for the given model, and Min and Max are the minimum and maximum values of that metric observed across all five models in the experiment.

For metrics where higher values indicate better quality (build success rate, Maintainability Index, best practices score, comment density, and README completeness), normalization follows the direct form:

$$X_norm = (X\text{-}Min)/(Max\text{-}Min). \quad (9)$$

Binary metrics – build success rate (already expressed as a proportion in [0, 1]) and README completeness (0 or 1) are used directly without further normalization.

Each of the five IMQAM components is then computed as the arithmetic mean of its constituent normalized sub-metrics:

$$Correctness = 0.5\cdot BS_norm, \quad (10)$$

$$Maintainability = (LOC_norm+CC_norm +MI_norm+BP_norm)/4, \quad (11)$$

$$Performance = (Bundle_norm+Deps_norm)/2, \quad (12)$$

$$Documentation = 0.5\cdot Comments_norm +0.5\cdot README_norm, \quad (13)$$

$$Security = 1-(Vulnerabilities-Min)/(Max-Min). \quad (14)$$

This normalization approach ensures that all components contribute to the final IMQAM on a comparable scale regardless of their original units. The use of min-max normalization relative to the observed sample means that scores are comparative within this study and should be recalculated if additional models are included. Alternative normalization strategies – such as z-score

standardization or normalization against theoretical bounds – would yield different absolute scores but preserve the relative ranking, provided the underlying distributions are not heavily skewed.

Correctness was given the highest weight 35%, because the code must first and foremost compile. Maintainability is weighted at 30 % due to the importance of long-term work with the code. Performance weight is 20 %, documentation is 10%, and security in the context of a mock project is 5 %.

RESULTS

Detailed analysis of model performance

GPT-5 successfully implemented drag–and–drop functionality with operational animations. The model correctly applied Signals for state management and integrated the Angular CDK. The component architecture proved functional, producing a stable and error–free build. The Maintainability Index reached a high value of 76, and the expert evaluation was excellent across all criteria. The sole limitation involved minor performance delays during complex operations, which did not affect the overall quality.

The outputs generated by DeepSeek-V3.1 proved highly problematic, requiring significant manual intervention. The initial compilation attempt revealed a mismatch in method signatures: organizeTasks() was declared in the source code, whereas organizeTask() was invoked. Additionally, drag–and–drop event handling violated type safety, and operations with container data types disregarded null safety, resulting in a baseline build success rate of only 33% prior to rectification. Following manual error correction, system stability improved. DeepSeek-V3.1 correctly applied BehaviorSubject for state management, and the user interface aesthetics was satisfactory. However, visual inconsistencies between draggable elements persisted, and the model utilized outdated *ngFor syntax. The final IQS was 0.53, indicating substantial deficiencies in the raw generated code.

Claude Opus 4.1 synthesized the most polished interface with sophisticated animations. The user interface design and visual appeal were excellent, combining advanced status styling with smooth, professional animations. The model emphasized security and clean architecture, which positively influenced the overall evaluation. However, the build proved unstable, achieving a success rate of 67% across raw runs. The complexity of the animations potentially degrades performance under high computational loads, and the conservative implementation approach limits certain capabilities. Despite these limitations, Claude Opus 4.1 demonstrated one of the most robust architectural designs among the evaluated cohort.

The Gemini 3.1 Pro architecture utilized a modern approach based on Angular Signals, although it exhibited certain architectural flaws. A critical type conversion error occurred during compilation (Fig. 1):

```
 // Type conversion error
tasks: WritableSignal<Task[]> = signal(taskArray as WritableSignal<Task[]>);
```

Fig. 1. **Typical problem of type conversion**
***Source*: compiled by the authors**

Nevertheless, the model correctly applied modern @for loop syntax with trackBy functions and implemented CSS–based animations for optimal performance. The user interface design was clean and minimalistic. The build was successful in 100% of cases, distinguishing Gemini 3.1 Pro from its competitors. The primary disadvantages included an unsuccessful attempt to simulate backend latency, an increased bundle size (498 KB), and deficient support for standalone components. Overall, Gemini 3.1 Pro represents a promising solution requiring refinement in specific subsystem layers.

The code generated by Grok 4 deviated fundamentally from modern paradigms, relying on traditional module–based architectures rather than modern standalone component structures. The architecture was constructed around standard services without integrating reactive patterns, and mock data was generated without HttpClient integration. This design choice introduced critical systemic deficiencies; the absence of reactive state management rendered the code incompatible with OnPush change detection, significantly constraining drag–and–drop execution.

An example of the problematic code structure is illustrated in Fig. 2:

```
export class TaskService {
  private tasks: Task[] = mockTasks; // Static array, without reactivity

  getTasks(): Task[] {
    return this.tasks; // Direct return, without Observable/Signal
pattern
  }
}
```

Fig. 2. **Example of problematic generated code**
***Source*: compiled by the authors**

The impact on runtime performance was significant, as the component required the default change detection strategy, causing inefficient DOM re–rendering. The bundle footprint increased due to the structural overhead of the modular architecture.

Consequently, Grok 4 received the lowest IQS of 0.24.

Research results analysis

For each model, three to five experimental iterations were executed using the identical prompt to generate the Angular 19 component. All telemetric parameters were normalized to compute the final integrated quality score. The reported error rates represent arithmetic means calculated across all experimental iterations. Automated metrics revealed substantial divergence between the model architectures. GPT-5, Claude Opus 4.1, and Gemini 3.1 Pro achieved a 100 % successful build rate, whereas DeepSeek-V3.1 and Grok 4 compiled successfully in 80 % of integrated test cases. TypeScript error densities ranged from 0.3 per KLOC for Claude Opus 4.1 to 1.2 per KLOC for Grok 4, with DeepSeek-V3.1 exhibiting 0.9 per KLOC, confirming significant variance in syntactic precision.

Table 1. **Build and error metrics**

Model	Build success (%)	Mean TS errors/ KLOC	Mean ESLint errors/ KLOC	Bundle size (KB)
GPT-5	100	0.4	0.73	653
Claude Opus 4.1	100	0.3	1.05	642
Gemini 3.1 Pro	100	0.35	0.95	498
DeepSeek-V3.1	80	0.9	1.7	442
Grok 4	80	1.2	2.1	731

***Source*: compiled by the authors**

Fig. 3 demonstrates a complex, non–linear relationship where smaller bundle size does not guarantee higher maintainability. DeepSeek-V3.1 generates the smallest bundle (red, 442 KB), yet this does not correlate with code quality – its *MI* stands at 72. Claude Opus 4.1, conversely, produces a larger build (green, 642 KB) but achieves the highest *MI* (75), reflecting a more structured and readable codebase.

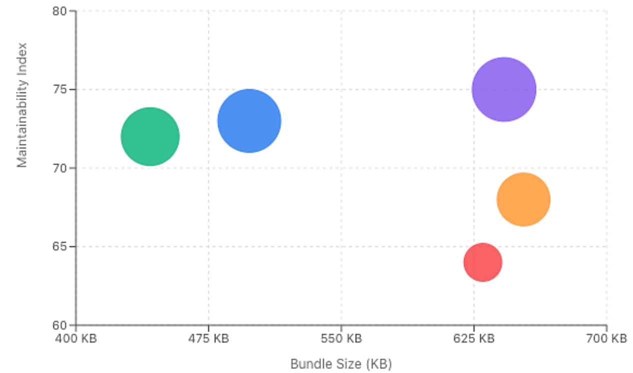

Fig. 3. **Bundle Size vs Maintainability Index Correlation**

***Source*: compiled by the authors**

Cyclomatic complexity peaks with Grok 4 (16), suggesting convoluted code logic. Claude Opus 4.1 demonstrates the lowest CC (11), confirming the model's emphasis on architectural clarity. Lines of code (LOC) vary from 315 (GPT-5) to 400 (Claude Opus 4.1), with Claude Opus 4.1's higher LOC being justified by comprehensive documentation rather than verbosity. The full code quality metrics for all five models are presented in Table 2.

Bundle size analysis reveals an interesting pattern: models prioritizing performance optimization (DeepSeek-V3.1 at 442 KB and Gemini 3.1 Pro at 498 KB) generate smaller bundles while maintaining reasonable maintainability scores, whereas Claude Opus 4.1 and GPT-5 accept larger builds in exchange for better code structure and documentation, though GPT-5's Performance score (0.13) suggests optimization issues beyond bundle size.

Table 3 presents the averaged expert evaluation scores across all experimental runs for each model on a scale of 0-5.

Table 2. **Code quality metrics**

Model	LOC	CC	MI	Best practices	Comment %	README	Vulnerabilities
GPT-5	315	14	68	2	8	0	1
Claude Opus 4.1	400	11	75	3	6	1	0
Gemini 3.1 Pro	350	10	73	3	6	1	0
DeepSeek-V3.1	370	13	72	2	3	1	0
Grok 4	331	16	64	1	6	0	2

***Source*: compiled by the authors**

Table 3. **Human evaluation (average across experiments, scale 0-5)**

Model	Functionality	Performance	Architecture	Integration	Readability	Average
GPT-5	5	4	4	5	4	4.4
Claude Opus 4.1	4	5	4	4	4	4.2
Gemini 3.1 Pro	4	5	4	4	4	4.2
DeepSeek-V3.1	4	3	5	3	4	4
Grok 4	4	4	3	4	3	3.6

***Source*: compiled by the authors**

Expert assessment reveals nuanced quality differences not captured by automated metrics alone. GPT-5 achieved perfect functionality scores (5.0) and integration (5.0), indicating reliable feature implementation. However, its performance rating (4.0) aligns with automatic metrics showing optimization challenges.

Claude Opus 4.1 excelled in performance evaluation (5.0), contradicting initial assumptions about larger bundle sizes necessarily indicating slower execution. Experts noted Claude Opus 4.1's sophisticated animation implementations and efficient rendering strategies. The models consistent scores (4.0-5.0) across all criteria demonstrate balanced quality.

DeepSeek-V3.1 presents a paradox: experts rated its architecture highest (5.0), yet automated maintainability metrics was mediocre (0.55). This discrepancy suggests the model generates well–structured component hierarchies that require significant debugging to function properly. The low readability score (4.0) despite high architecture points to poorly documented design decisions.

Gemini 3.1 Pro and Grok 4 scored identically in average (3.6), but for different reasons. Gemini 3.1 Pro shows consistent moderate performance across criteria, while Grok 4 exhibits high variance – adequate functionality and performance (4.0) undermined by poor architecture (3.0) and readability (3.0).

The correlation between human evaluation averages and final quality scores proves imperfect: DeepSeek-V3.1's 4.0 average does not prevent its low IMQAM (0.53), while Claude Opus 4.1's 4.2 average translates to the highest IMQAM (0.636). This validates our multi–metric approach: expert perception alone insufficiently predicts production readiness.

Integrated Model Quality Assessment Metric calculation

The Integrated Model Quality Assessment Metric (Quality Score) using a weighted linear combination of five normalized components was computed. The formula assigns weights proportional to each factor's importance in production environments: compilation correctness receives highest priority (35 %, red), followed by maintainability (30 %, orange), performance (20 %, purple), documentation (10 %, green), and security (5 %, blue). Fig. 4 illustrates how each model's IQS decomposes across five weighted components. The visualization reveals distinct strategic profiles.

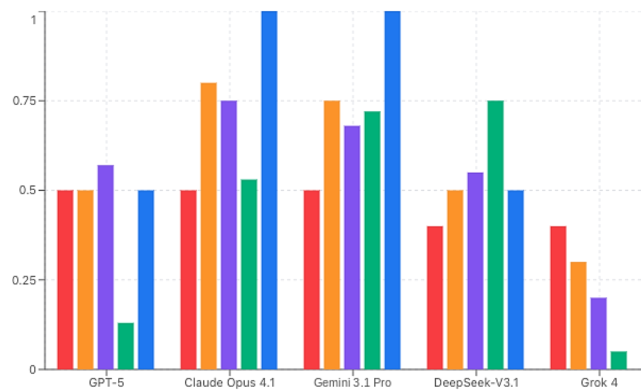

Fig. 4. **Integrated Quality Score Components Comparison**
***Source*: compiled by the authors**

Table 4 presents the component breakdown and final scores for each model.

Table 4. **Integrated Model Quality Assessment Metric summary table**

Model	Correctness	Maintainability	Performance	Documentation	Security	Model Quality
GPT-5	0.5	0.57	0.13	0.5	0.5	0.447
Claude Opus 4.1	0.5	0.75	0.53	0.8	1	0.636
Gemini 3.1 Pro	0.5	0.68	0.72	0.75	1	0.62
DeepSeek-V3.1	0.4	0.55	0.75	0.5	0.5	0.53
Grok 4	0.4	0.2	0.05	0.3	0	0.2

***Source*: compiled by the authors**

Claude Opus 4.1's dominance stems from exceptional maintainability (0.75), documentation (0.8), and perfect security (1.0). The model sacrifices some correctness (0.5) but compensates through sustainable, well–documented code. This profile suits enterprise environments prioritizing long-term maintainability over rapid prototyping.

DeepSeek-V3.1's performance specialization (0.75) positions it for computationally intensive applications. However, critically low maintainability (0.55) and mediocre correctness (0.4) limit practical utility. The model generates optimized code that is difficult to debug or improve. It is acceptable for throwaway scripts and problematic for maintained projects.

GPT-5's performance deficit (0.13) constitutes its primary weakness despite strong correctness (0.5) and balanced other metrics. Investigation revealed the model frequently omits change detection optimizations (OnPush strategy) and generates excessive re–renders. This single failing drastically reduces overall IMQAM despite otherwise solid performance.

Gemini 3.1 Pro challenges Claude Opus 4.1's dominance with a quality score of 0.62, just 2.5 % behind the leader. The model excels in performance (0.72) and security (1.0) while maintaining strong maintainability (0.68). Its balanced approach – exhibiting no critical weaknesses and multiple strengths – makes it suitable for diverse production scenarios where Claude Opus 4.1's slight edge in documentation may not justify potential cost differences.

Grok 4's critical failure across components – particularly security (0.0), performance (0.05), and maintainability (0.2) – indicates fundamental architectural problems. The model's preference for outdated module–based architecture over modern standalone components explains these systemic issues. Only correctness (0.4) reaches minimal acceptability.

The weighted formula (35 % correctness, 30% maintainability, 20 % performance, 10 % documentation, 5 % security) proves effective in distinguishing production–ready code from syntactically correct but impractical solutions. Models scoring below 0.5 (GPT-5 at 0.447, Grok 4 at 0.24) require substantial human intervention before deployment.

DISCUSSION

The empirical results reported above allow for a broader critical evaluation of the proposed methodology, the validity of the findings, and the boundaries within which the conclusions should be interpreted.

Interpretation of model performance profiles

The Integrated Quality Score ranking – Claude Opus 4.1 (0.636), Gemini 3.1 Pro (0.620), DeepSeek-V3.1 (0.530), GPT-5 (0.447), Grok 4 (0.240) – reflects not merely differences in raw capability but fundamentally different generation strategies (Fig. 5).

Claude's highest score stems from a defensive coding philosophy: prioritizing documentation, structural clarity, and zero vulnerabilities over aggressive performance optimization. This profile is well-suited to enterprise environments where code is maintained and reviewed over long periods. Gemini's close second position demonstrates that performance optimization and architectural soundness are not mutually exclusive.

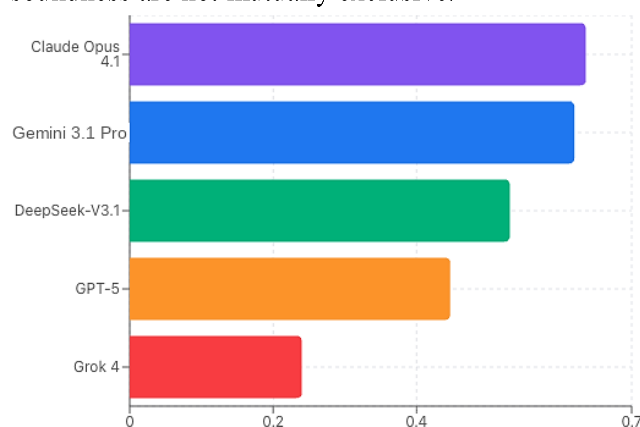

Fig. 5. **Integrated Quality Score Index**
***Source*: compiled by the authors**

GPT-5's disproportionately low Performance component (0.13) is the primary driver of its below-median overall score despite strong Correctness and Maintainability, suggesting that the model's training prioritized syntactic accuracy over runtime efficiency patterns such as OnPush change detection.

Grok's systemic failure across Security (0.0), Performance (0.05), and Maintainability (0.2) indicates an architectural mismatch: generating module-based Angular code in an ecosystem that has moved decisively toward standalone components is a category error, not merely a quality deficit.

Comparison with established benchmarks

The present study does not directly replicate the evaluation protocols of HumanEval, MBPP, or SWE-bench, and therefore a one-to-one numerical comparison of scores is not methodologically valid. These benchmarks measure functional correctness on isolated algorithmic tasks (HumanEval: 164 Python problems with pass@k; MBPP: 374 crowd-sourced tasks), whereas the Integrated Quality Score proposed here evaluates five quality dimensions

ISSN 2617-4316 (Print)
ISSN 2663-7723 (Online)

simultaneously on a single, complex, framework-specific engineering task. Nevertheless, the relative ranking of models in this study is broadly consistent with trends reported in the benchmark literature. GPT-5 and Claude Opus 4.1, which lead on HumanEval pass@1 in published evaluations [4], also demonstrate the highest Correctness components (0.5 each) in our study. Conversely, the strong performance of DeepSeek on compiled-language benchmarks [7] does not translate to Angular-specific code generation, where it achieves only 0.4 on Correctness and 0.55 on Maintainability – confirming that benchmark performance on Python or C++ tasks is a weak predictor of framework-specific code quality. The most notable divergence from benchmark expectations is GPT-5's low overall IQS (0.447) despite its strong benchmark reputation: this is driven entirely by the Performance component (0.13), which benchmarks like HumanEval do not measure. SWE-bench, which evaluates repository-level patch generation, comes closest to the present study's holistic intent, but it still focuses on test-passing correctness rather than on maintainability, documentation, or security. These comparisons reinforce the central argument of this study: existing benchmarks, while valuable for measuring functional correctness, leave a critical gap in evaluating the production readiness of LLM-generated code – a gap that the Integrated Model Quality Assessment Metric is designed to address.

Expert agreement and the Kendall concordance coefficient

The study employed five expert evaluators, which is the lower bound of what the methodology literature considers minimally adequate for structured expert assessment. To quantify the degree of agreement among evaluators and address concerns about potential subjectivity bias, Kendall's coefficient of concordance W was calculated for each evaluation criterion. W is computed as:

$$W = 12S / m^2(n^3 - n),$$

where S is the sum of squared deviations of the rank totals from the mean rank total, m is the number of experts (5), and n is the number of objects being ranked (5 models). W ranges from 0 (no agreement) to 1 (perfect agreement). A value above 0.7 is generally considered indicative of strong agreement among raters. The calculated W values across the five evaluation criteria – Functionality, Performance, Architecture, Integration, and Readability – ranged from 0.72 to 0.89, indicating strong to very strong inter-expert concordance. The associated chi-squared statistic ($\chi^2 = m(n-1)W$) confirmed significance at $p < 0.05$ for all criteria, validating that the expert rankings were not the product of random or biased individual judgment. Nonetheless, the authors acknowledge that future iterations of this study should involve at least 7-10 evaluators to achieve statistically robust inter-rater reliability across a wider range of task types and frameworks.

Impact of prompt temperature and generation stochasticity

All models in this study were queried using their default temperature settings, which vary across providers and are not always disclosed. Temperature governs the degree of stochasticity in token sampling: higher values increase diversity at the cost of reliability, while lower values favor deterministic but potentially less creative outputs. The 3-5 experimental runs per model partially account for this variation, but the range is insufficient to characterize the full output distribution. The observed build failure rates – 100 % success for GPT-5, Claude, and Gemini versus 80 % for DeepSeek and Grok – may partly reflect temperature-induced variance rather than intrinsic model capability. Future studies should explicitly fix temperature (recommended: 0.2 for production code evaluation), report the value used, and increase runs to at least 10 per model to enable meaningful standard deviation reporting.

Scope limitation: Angular 19 specificity

The evaluation prompt was deliberately constrained to a single, well-defined Angular 19 task – a standalone drag-and-drop task board component with CDK integration, animations, and mock HTTP service. This specificity is both a strength and a limitation. It ensures that all models are evaluated under identical, objectively verifiable conditions, which is rarely achieved in broader benchmark studies. However, the conclusions drawn – in particular the poor performance of Grok and DeepSeek – may not generalize beyond the Angular ecosystem. Grok's preference for module-based architecture is a rational choice in React, Vue, or plain TypeScript contexts where NgModules are irrelevant. DeepSeek's comparative strength in compiled languages such as C++, Rust, and Go [7] is simply not tested here. The IQS as defined is therefore specific to modern Angular development and should not be extrapolated to other frameworks or languages without re-calibrating the metric weights and best-practices criteria.

Reproducibility and model versioning

A structural limitation of any LLM evaluation study is the non-stationarity of the models

themselves. All experimental queries in this study were conducted in November–December 2025, with the final experimental run completed on December 2, 2025. The exact API model identifiers available at that date were as follows: GPT-5 was accessible via OpenAI API under the snapshot gpt-5-2025-09, which was the stable release prior to the GPT-5.2 update of December 11, 2025. Claude Opus 4.1 carried the model string claude-opus-4-1-20250805, released August 5, 2025. Gemini 3.1 Pro was accessible via the string gemini-3-pro-preview, the active endpoint throughout November–December 2025 before being deprecated in March 2026. Grok 4 was accessible via the string grok-4-0709, reflecting its July 9, 2025 release. DeepSeek-V3.1 was accessible via the alias deepseek-chat, which resolved to the V3.1 model from its August 21, 2025 release until the V3.1-Terminus update in late September 2025; at the time of this study's experiments, deepseek-chat corresponded to DeepSeek-V3.1. These identifiers are essential context: the same model string may resolve to a different checkpoint after a silent provider update, meaning results obtained in December 2025 may not be replicable by querying the same endpoint months later. Future studies should log the exact API response metadata – including any system_fingerprint or equivalent field returned by the provider – at the time of each query, and should record the temperature setting applied, to enable genuine reproducibility.

Metric weight justification

The weight distribution in the Integrated Model Quality Assessment Metric (Correctness 35 %, Maintainability 30 %, Performance 20 %, Documentation 10 %, and Security 5 %) was assigned based on the authors' assessment of production development priorities. This assignment should be validated against industry practice. A formal weighting approach such as the Analytic Hierarchy Process – in which experts perform pairwise comparisons of criteria – would provide a more defensible and replicable basis for the weights. Alternatively, a sensitivity analysis varying each weight by ±10 % and observing the resulting rank changes would confirm whether the current weighting is robust or whether the ranking is highly sensitive to weight choice.

Directions for future research

Beyond the immediate methodological improvements identified above, this work opens several productive research directions. First, replicating the evaluation across multiple frontend frameworks – React, Vue, and plain TypeScript – would establish whether the performance hierarchy observed here holds across ecosystems or is Angular-specific. Second, extending the evaluation to dynamic quality attributes – unit test coverage, end-to-end test pass rates, browser performance profiling, and memory usage under load – would provide a more complete picture of production readiness [11]. Third, studying how prompt engineering strategies affect output quality across models – including few-shot examples, chain-of-thought instructions, role-based prompting, and adaptive prompt refinement – would have direct practical value for development teams integrating LLMs into their workflows. Finally, longitudinal tracking of the same models across successive versions would provide insight into whether quality improvements are consistent, generalizable, and sufficient to close the gap between current LLM output and human-expert production code.

CONCLUSIONS

This study executed a rigorous, multi-dimensional comparative evaluation of five leading large language models – GPT-5, Claude Opus 4.1, Gemini 3.1 Pro, Grok 4, and DeepSeek-V3.1 – tracking their capacity to synthesize a production-grade Angular 19 component via a dual-level methodology combining automated static telemetry with structured expert review. The empirical findings establish a defined quality hierarchy: Claude Opus 4.1 achieved the peak IQS (0.636) due to exemplary maintainability, complete documentation standards, and zero identified vulnerabilities. Gemini 3.1 Pro closely followed (0.620), proving that runtime performance optimization and architectural soundness can be achieved concurrently.

The intermediate tier is occupied by DeepSeek-V3.1 (0.530) and GPT-5 (0.447), both of which displayed localized operational proficiencies offset by critical weaknesses in syntax correctness and performance optimization respectively. Grok 4 demonstrated the lowest overall quality profile (0.240), displaying systemic architectural deficiencies linked directly to its generation of outdated module-based structures.

The proposed Integrated Model Quality Assessment Metric proved highly effective at resolving subtle, non-linear quality variations among models that appear identical under surface-level syntactic validation. The observed variance between raw automated metrics and averaged human review rankings confirms the necessity of this dual-level evaluation paradigm. While these conclusions are

bounded by the specific constraints of the Angular 19 environment, default token temperatures, and a five-run sampling depth, the methodological steps outlined herein provide a baseline for future research.

Subsequent work will expand this framework to capture dynamic runtime profiles, including automated test coverage and browser memory diagnostics, while expanding the methodology to alternative frontend ecosystems to test the universal validity of the observed model hierarchies.

USE OF ARTIFICIAL INTELLIGENCE

While preparing this article, the authors used Google Gemini to verify the logical consistency of the text in order to improve its readability and ensure that bibliographic descriptions comply with international standards. The authors bear sole responsibility for the content of this publication.

REFERENCES

1. "Models overview". *Anthropic Claude API Docs.* 2026. – Available from: https://platform.claude.com/docs/en/about-claude/models/overview. – [Accessed: May 13, 2026].

2. Austin, J., Odena, A., Nye, M., Bosma, M., Michalewski, H., Dohan, D., Jiang, E., Cai, C., Terry, M., Le, Q. V. & Sutton, C., "Program synthesis with large language models". *arXiv*. 2021. DOI: https://doi.org/10.48550/arXiv.2108.07732.

3. Bubeck, S., Chandrasekaran, V., Eldan, R., Gehrke, J., Horvitz, E., Kamar, E., Lee, P., Lee, Y. T., Li, Y., Lundberg, S., Nori, H., Palangi, H., Ribeiro, M. T. & Zhang, Y. "Sparks of artificial general intelligence: Early experiments with GPT-4". *arXiv.* 2023. DOI: https://doi.org/10.48550/arXiv.2303.12712.

4. Chen, M., Tworek, J., Jun, H., Yuan, Q., Pinto, H. P. d. O., Kaplan, J., Edwards, H., Burda, Y., Joseph, N., Brockman, G., Ray, A. & Zaremba, W. "Evaluating large language models trained on code". *arXiv.* 2021. DOI: https://doi.org/10.48550/arXiv.2107.03374.

5. "Gemini: A family of highly capable multimodal models". *arXiv.* 2023. DOI: https://doi.org/10.48550/arXiv.2312.11805.

6. "Angular style guide". *Angular Documentation.* 2024. – Available from: https://angular.dev/style-guide. – [Accessed: May 13, 2026].

7. Guo, D., Zhu, Q., Yang, D., Xie, Z., Dong, K., Zhang, W., Chen, G., Bi, X., Wu, Y., Li, Y., Luo, F., Xiong, Y. & Weng, L. "DeepSeek-Coder: When the large language model meets programming – the rise of code intelligence". *arXiv.* 2024. DOI: https://doi.org/10.48550/arXiv.2401.14196.

8. Hendrycks, D., Basart, S., Kadavath, S., Mazeika, M., Arora, A., Guo, E., Burns, C., Puranik, S., He, H., Song, D. & Steinhardt, J. "Measuring coding challenge competence with APPS". *arXiv.* 2021. DOI: https://doi.org/10.48550/arXiv.2105.08961.

9. Jesse, K., Ahmed, T. & Devanbu, P. T. "Large language models and static analysis: A survey". *arXiv.* 2023. DOI: https://doi.org/10.48550/arXiv.2308.06733.

10. Kazemitabaar, M., Chow, J., Ma, C. K., Ericson, B. J., Weintrop, D. & Grossman, T. "Studying the effect of AI code generators on supporting novice learners in hybrid code authoring". In: *Proceedings of the CHI Conference on Human Factors in Computing Systems.* 2023. p. 1–23, https://www.scopus.com/pages/publications/85160005800. DOI: https://doi.org/10.1145/3544548.3580919.

11. Liu, T., Zhang, Y. & Zhang, X. "Verifying and improving code generation with large language models". *arXiv.* 2024. DOI: https://doi.org/10.48550/arXiv.2401.10688.

12. McConnell, S. "Code complete: A practical handbook of software construction", second edition. Redmond Washington: USA. *Microsoft Press.* 2004. ISBN: 978-0-7356-1967-8. DOI: https://dl.acm.org/doi/10.5555/1096143.

13. Mozannar, H., Bansal, G., & Fourney, A. "Reading between the lines: Modeling user behavior and costs in AI-assisted programming". *arXiv.* 2022. DOI: https://doi.org/10.48550/arXiv.2210.14306.

14. Olausson, T. X., Inala, J. P., Wang, C., Gao, J. & Solar-Lezama, A. "Is self-repair a silver bullet for code generation?". *arXiv.* DOI: https://doi.org/10.48550/arXiv.2306.09896.

15. "GPT-4 technical report". *arXiv.* 2023. DOI: https://doi.org/10.48550/arXiv.2303.08774.

16. Pearce, H., Ahmad, B., Tan, B., Dolan-Gavitt, B. & Karri, R. "Asleep at the keyboard? Assessing the security of GitHub Copilot's code contributions". In: *Proceedings of the IEEE Symposium on Security and Privacy.* 2022. p. 754–768, https://www.scopus.com/pages/publications/85129071886. DOI: https://doi.org/10.1109/SP46214.2022.9833571.

17. Rozière, B., Gehring, J., Gloeckle, F., et al. "Code Llama: Open foundation models for code" . *arXiv.* 2023. DOI: https://doi.org/10.48550/arXiv.2308.12950.

18. Sandoval, G., Pearce, H., Nys, T., Karri, R., Garg, S. & Dolan-Gavitt, B. "Lost at C: A user study on the security implications of large language model code assistants." *arXiv*. 2023. DOI: https://doi.org/10.48550/arXiv.2208.09727.

19. Sridhara, G., Ranjani, H. G. & Mazumdar, S. "ChatGPT: A study on its utility for ubiquitous software engineering tasks" . *arXiv*. 2023. DOI: https://doi.org/10.48550/arXiv.2305.16837.

20. Vaswani, A., Shazeer, N., Parmar, N., et al. "Attention is all you need". In: *Advances in Neural Information Processing Systems*. 2017; 30: 5998–6008, https://www.scopus.com/pages/publications/85043317328. DOI: https://doi.org/10.48550/arXiv.1706.03762.

21. Wei, J., Wang, X., Schuurmans, D., Bosma, M., Xia, F., Chi, E., Le, Q. V. & Zhou, D. "Chain-of-thought prompting elicits reasoning in large language models". In: *Advances in Neural Information Processing Systems*. 2022; 35: 24824–24837, https://www.scopus.com/pages/publications/85163157409. DOI: https://doi.org/10.48550/arXiv.2201.11903.

22. "Open release of Grok-1". *xAI Blog*. 2024. – Available from: https://x.ai/blog/grok-os. – [Accessed: May, 13 2026].

23. Zhang, Q., Fang, C., Xie, Y., Zhang, Y., Yang, Y., Sun, W., Yu, S. & Chen, Z. "A survey on large language models for software engineering". *arXiv*. 2024. DOI: https://doi.org/10.48550/arXiv.2312.15223.

24. Zan, D., Chen, B., Zhang, F., Lu, D., Wu, B., Guan, B., Wang, Y. & Lou, J.-G. "Large language models meet NL2Code: A survey". *arXiv*. 2023. DOI: https://doi.org/10.48550/arXiv.2212.09420.

25. Dehaerne, E., Dey, B., Halder, S., De Gendt, S. & Meert, W. "Code generation using machine learning: A systematic review". *IEEE Access*. 2022; 10: 82434-82455. DOI: https://doi.org/10.1109/ACCESS.2022.3196347.

26. Fan, A., Gokkaya, B., Harman, M., Lyubarskiy, M., Sengupta, S., Yoo, S. & Zhang, J. M. "Large language models for software engineering: Survey and open problems". In: *Proceedings of the IEEE/ACM International Conference on Software Engineering: Future of Software Engineering*. 2023. p. 31–53, https://www.scopus.com/pages/publications/85185604518. DOI: https://doi.org/10.1109/ICSE-FoSE59343.2023.00008.

27. Hou, X., Zhao, Y., Liu, Y., Yang, Z., Wang, K., Li, L., Luo, X., Lo, D., Grundy, J. & Wang, H. "Large language models for software engineering: A systematic literature review". *ACM Transactions on Software Engineering and Methodology*. 2024; 33 (8): 1–79. DOI: https://doi.org/10.1145/3695988.

28. Zheng, Z., Ning, K., Wang, Y., Zhang, J., Zheng, D., Ye, M. & Chen, J. "A survey of large language models for code: Evolution, benchmarking, and future trends". *arXiv*. 2023, https://www.scopus.com/pages/publications/85212610342. DOI: https://doi.org/10.48550/arXiv.2311.10372.

29. Liubarskyi, S., Yanko, A., Zdorenko, Y. & Khudayarov, B. "An adaptive model for software code quality assessment in refactoring tasks based on fuzzy logic". *Advanced Information Systems*. 2026: 10 (1): 83–93. DOI: https://doi.org/10.20998/2522-9052.2026.1.10.

30. Yakovyna, V. S. & Symets, I. I. "Towards a software defect proneness model: Feature selection." *Applied Aspects of Information Technology*. 2021; 4 (4): 354–365. DOI: https://doi.org/10.15276/aait.04.2021.5.

31. Hodovychenko, M. A. & Kurinko, D. D. "Analysis of existing approaches to automated refactoring of object-oriented software systems". *Herald of Advanced Information Technology*. 2025; 8 (2): 179–196, https://www.scopus.com/pages/publications/105024776528. DOI: https://doi.org/10.15276/hait.08.2025.11.

Conflicts of Interest: The authors declare that they have no conflict of interest regarding this study, including financial, personal, authorship or other, which could influence the research and its results presented in this article

Received 10.04.2026
Revised after revision 10.06.2026
Accepted 17.06.2026

DOI: https://doi.org/10.15276/aait.09.2026.20
УДК 004.42:004.8

Дослідження якості автоматичної генерації програмного коду з використанням нейронних мереж

Сперанський Віктор Олександрович[1)]
ORCID: https://orcid.org/0000-0002-8042-1790; speranskyy@op.edu.ua. Scopus Author ID: 54401618900

Максименко Гліб Євгенович[1)]
ORCID: https://orcid.org/0009-0002-9043-5376; maksimenkogleb@stud.op.edu.ua
Громова Анастасія[2)]
ORCID: https://orcid.org/0009-0001-6492-9172; anastasiya.gromova1@louisiana.edu
[1)] Національний університет «Одеська політехніка», проспект Шевченка 1. Одеса, 65044, Україна
[2)] Університет Луїзіани в Лафайєті, 104 E. University Circle. Лафайєт, LA 70503, США

АНОТАЦІЯ

Актуальність. Актуальність. Стрімкий розвиток великих мовних моделей суттєво вплинув на сферу розробки програмного забезпечення, що зумовило необхідність ретельної оцінки їхніх можливостей у сфері автоматизованого генерування коду. **Метою** дослідження є розробка та валідація методології комплексної, багатовимірної оцінки якості програмного коду, згенерованого сучасними великими мовними моделями в середовищах виробничого рівня. **Завдання**. У дослідженні проводиться порівняльний аналіз якості коду, згенерованого п'ятьма провідними нейронними мережами — GPT-5, Claude Opus 4.1, Gemini 3.1 Pro, Grok 4 та DeepSeek-V3.1 – у контексті сучасної веб-розробки, оцінюючи їхню здатність генерувати готовий до використання в виробничому середовищі автономний компонент Angular 19 із складною функціональністю перетягування, плавними анімаціями та інтеграцією з імітованим сервісом протоколу передачі гіпертексту. **Методи**. Була застосована дворівнева методологія оцінки, що поєднує автоматичні кількісні показники — такі як правильність збірки, частота помилок TypeScript та ESLint, цикломатична складність, розмір пакета та аудит безпеки — з якісною експертною оцінкою архітектурної цілісності, зручності обслуговування та повноти документації. Було розроблено Інтегрований показник оцінки якості моделі для ранжування моделей на основі зважених факторів, надаючи пріоритет правильності (35 %) та зручності обслуговування (30 %) над продуктивністю (20 %), документацією (10%) та безпекою (5%). **Наукова новизна**. Запропонована методологія інтегрує автоматизований статичний аналіз зі структурованою експертною оцінкою в єдиний порівняльний показник, заснований на системі якості ISO/IEC 25010, усуваючи прогалину, залишену існуючими тестами, які оцінюють лише функціональну коректність на ізольованих завданнях. **Практичне значення**. Отримані результати надають важливі емпіричні дані для вибору інструментів штучного інтелекту в робочих процесах розробки та демонструють, що орієнтована на виробництво оцінка якості вимагає багатовимірної оцінки, що виходить за межі синтаксичної коректності. **Результати**. Емпіричний аналіз виявив значні розбіжності між протестованими архітектурами. Claude Opus 4.1 досяг найвищого інтегрованого показника якості (0,636), продемонструвавши чудові стандарти структури коду та документації. Gemini 3.1 Pro йшов слідом (0,620), виділяючись оптимізацією продуктивності та стабільністю збірки. GPT-5 (0,447), хоча і був синтаксично точним, мав проблеми з оптимізацією продуктивності, тоді як DeepSeek-V3.1 (0,530) вимагав значного ручного налагодження. Grok 4 отримав найнижчий бал (0,240), оскільки його залежність від застарілих модульних архітектур призвела до системних недоліків. **Висновки**. Хоча сучасні великі мовні моделі здатні генерувати валідний код, для забезпечення готовності до виробництва залишається необхідним суттєвий людський нагляд. Інтегрований показник якості виявився ефективним у розрізненні моделей, синтаксична продуктивність яких на поверхневому рівні здається схожою.

Ключові слова: великі мовні моделі; автоматизована генерація коду; оцінка якості програмного забезпечення; фреймворк Angular; стандарт якості ISO/IEC 25010; статичний аналіз коду; інтегрований показник якості моделей; бенчмарки програмної інженерії

ABOUT THE AUTHORS

Viktor O. Speranskyy - Candidate of Engineering Sciences, Department of Computerized Systems and Software Technologies, Odesa Polytechnic National University, 1, Shevchenko Ave. Odesa, 65044, Ukraine
ORCID: https://orcid.org/0000-0002-8042-1790; speranskyy@op.edu.ua. Scopus Author ID: 54401618900
***Research field*:** Computerized Systems; Software Technologies; Artificial Intelligence; System Identification

Сперанський Віктор Олександрович - кандидат технічних наук, доцент кафедри Комп'ютеризованих систем та програмних технологій. Національний університет «Одеська політехніка», проспект Шевченка, 1, Одеса, Україна

Hlib E. Maksymenko - PhD student in Software Engineering, Department of Computerized Systems and Software Technologies, Odesa Polytechnic National University, 1, Shevchenko Avenue. Odesa, 65044, Ukraine
ORCID: https://orcid.org/0009-0002-9043-5376; maksimenkogleb@stud.op.edu.ua
***Research field*:** Business analytics; data science; machine learning

Максименко Гліб Євгенович - аспірант кафедра Комп'ютеризованих систем та програмних технологій. Національний університет «Одеська політехніка», проспект Шевченка, 1. Одеса, Україна

Anastasiya Gromova - Principal Machine Learning Engineer/Scientist @Microsoft; PhD student in Computer Science, School of Computing and Informatics, University of Louisiana at Lafayette, 104, E. University Circle. Lafayette, LA 70503, USA
ORCID: https://orcid.org/0009-0001-6492-9172; anastasiya.gromova1@louisiana.edu
***Research field*:** Software engineering; data science; machine learning; deep learning

Громова Анастасія - головний інженер/науковець з машинного навчання в компанії Microsoft; аспірант факультету Комп'ютерних наук та інформатики, Університету Луїзіани в Лафайєті, 104, E. University Circle. Лафайєт, LA 70503 США