Benchmarking AI-Generated Code: Evaluating Correctness, Maintainability, and Performance

Sonali Shrivastava[1], Dr. Geetanjali Jindal[2]
[1]*Technical Trainer, Knight Fintech Research Pvt Ltd., Mumbai, India*
[2]*Associate Professor, School of Computer Science and Applications, Nirwan University, Jaipur, India*

***Abstract*—AI code generators are becoming more common in software development, but the tests that rate them usually focus only on whether the code works correctly. This article looks at how the correctness; ease of maintenance and speed of AI-generated code are measured. It also asks if these factors are connected and why some studies comparing AI code with human-written code have results. A detailed summary of 16 sources published from 2023 to 2026 was made. These sources included five surveys, four studies that tested models, six small studies and theses and one benchmark paper. Some of the numbers from the studies were recalculated. The summary found that (a) passing rates on tasks are high but don't say much about other qualities, (b) there is no standard way to measure maintainability and the tools used to check it don't agree much (c) AI code that works correctly can be much slower than code written by experts (d) whether AI code is better or worse, than human code depends on the human example used, the programming language, the tools and how results are adjusted and (e) fixing AI code in steps can add new problems. Based on these findings the article suggests a way to test AI code. This includes a metric that checks quality, a set of things to report and a way to test the method. The results are for the sources used and the new way has not been tested in real situations yet.**

***Index Terms*—AI-generated code, large language models, code generation benchmarks, software quality, maintainability, static analysis**

I. INTRODUCTION

AI coding assistants have moved from being an experiment to a part of daily work. A Stack Overflow survey mentioned in the corpus found that over 75% of developers were either using AI tools or planning to use them (Dossas, 2025). Another report says that AI assistants are now writing a portion of the code that developers produce (Sabra et al. 2025). When it comes to productivity the results are not clear. Patel et al. (2026) Based on data found that tasks were completed faster but the stability of deliveries at the organizational level was lower. No matter what the overall effect on productivity is, the code that AI produces must still be correct, easy to understand, efficient and safe to deploy.

For a time, research on code generation has focused on a narrow question: does the generated function pass the tests? Benchmarks, like HumanEval, MBPP and APPS measure this using the pass@k metric (Truong et al., 2026). In real-world development the questions are broader. Can another developer maintain the code? Does it use resources in a way? Does it contain bugs that might cause problems later? Truong et al. (2026) Point out that most research still focuses on the first question. The studies examined here keep showing that passing tests and writing code are not the same thing (Asfandyar et al. 2025; Molison et al. 2025; Sabra et al. 2025).

A. Problem Statement and Research Gap

Three gaps drive this article. First, maintainability and performance are not tested enough compared to correctness. There is no benchmark for maintainability (Truong et al. 2026). Second, studies that compare code written by AI and code written by people often disagree. It's hard to say if these differences come from changes between models or from different ways of doing the studies. Third, most quality data come from analyzers. These tools rarely check how well they agree with each other as Dossas noted in 2025. Also, there is no used measure that connects correctness with both maintainability and efficiency. This is unlike pass@k, which links correctness, with security as Truong et al. Pointed out in 2026.

B. Aim and Research Questions

The goal is to bring what we know about testing AI-generated code in three main areas: correctness, maintainability, and performance. Security is treated as a cutting concern that matters across all the others. From this we want to create a benchmarking plan that fills in the missing pieces. Five research questions lead the way:

1. RQ1. How do people actually test correctness, maintainability and performance of code made by AI?
2. RQ2. What do real studies tell us when comparing AI-generated code to code written by humans on each of these three areas?
3. RQ3. If code is correct does that mean it's also maintainable or fast? Can one score predict the others?
4. RQ4. What are the problems in how studies are done—like shaky measurement tools make it hard to trust or compare the results?
5. RQ5. Based on all this what should a benchmark look like and what should future research focus on?

C. Contributions

This article offers four contributions:

a) A full review pulling together evidence on correctness, maintainability, and performance from 16 different sources.
b) A look at why studies disagree on whether AI code is better or worse than human code with a focus on how the baseline code (human-written) is chosen and how different tools measure quality.
c) A check of the collected studies, including recalculating the number of bugs found in published fuzzing tests to see if numbers were reported correctly.
d) A new benchmark protocol that starts with correctness then checks the qualities only if correctness is met. It includes a pass/fail rule based on overall quality, a standard for reporting results and a design for testing if the benchmark works.

II. BACKGROUND: QUALITY DIMENSIONS AND METRICS

A. A Quality Taxonomy

Truong and colleagues in 2026 took ISO/IEC 25010. Divided it into four parts for code made by large language models: functional correctness, security, performance efficiency and maintainability. They believe that old measures, such, as complexity, size and readability only describe the code. These old measures do not give a picture of what large language models produce. In 2025 Sabra and colleagues used SonarQube to split code into three categories: reliability, which covers bugs; security, which covers vulnerabilities; and maintainability, which covers code smells.

B. Correctness Metrics

Functional correctness is mostly checked using unit-test execution. The usual measure is called pass@k. This is the chance that least one of k tried solutions passes all tests. It is calculated based on n tries for each problem (Truong et al. 2026). Some research also looks at validity. That is the percentage of outputs that can be compiled or run (Pandya & Soni 2026). They also consider graded results. These include partially correct incorrect compile error and runtime error. These results give information than just pass or fail (Azbarka et al. 2025). People who do surveys point out four problems with pass@k. It is a yes or no measure. Does not consider partial correctness. It depends on n and k. Favors models with high variation when k is big. Weak test suites make results look better than they are. Also, randomness in hardware, kernels and libraries makes it hard to repeat results when using the same settings (Truong et al. 2026). Pandya and Soni (2026) say that text-similarity measures, like BLEU are also used. They do not reflect actual function behavior well.

C. Maintainability Metrics

No accepted benchmark for maintainability exists, as Truong et al. 2026 Point out. Researchers often rely on metrics such as complexity, cognitive complexity and Halstead measures. Other used indicators, for maintainability include the maintainability Index, lines of code, code duplication and code-smell density (Asfandyar et al. 2025; Gontiya 2025; Molison et al. 2025; Sabra et al. 2025). A few studies evaluate SonarQube remediation effort measured in minutes (Molison et al. 2025). Others apply rule-based tools, like PMD and check style to detect code smells (Dossas, 2025). Also, there are checklists that assess line length, function length, nesting levels and naming conventions (Azbarka et al., 2025). Truong et al. (2026) Found two studies focused on maintainability

but both were not fully automated or applied in a systematic way.

D. Performance Metrics

EffiBench takes the execution time (NET) and memory (NMU, NTMU). Compares them to human canonical solutions. When the values are higher than 1 it means the code that was generated is slower or uses memory (Huang et al., 2024). ENAMELs eff@k extends pass@k to include efficiency. It also considers timeouts as censored observations (Truong et al., 2026). Other studies look at repeated profiling runs (Asfandyar et al., 2025). Some studies do Big-O analysis without running the code (Azbarka et al., 2025). Others use performance-smell rules. An example of this is object creation, inside loops (Dossas, 2025).

E. Security Metrics

Security is assessed by mapping findings to Common Weakness Enumeration (CWE) categories with static analyzers such as CodeQL, Bandit, SonarQube, and Clang, sometimes supplemented by dynamic testing (Chong et al., 2024; Kantek, 2023; Truong et al., 2026). CodeGuard+ introduces secure-pass@k, which couples' correctness and security in a single metric (Truong et al., 2026).

III. METHOD

A. Review Design

The study is a synthesis. The process had five steps. These steps are (a) collecting the corpus (b) taking out the data (c) checking the quality (d) putting things together based on quality and (e) creating a benchmark protocol from the gaps and conflicts that were found. A quantitative meta-analysis was not done. The reason is that the sources are different, in tasks, models, languages, metrics and baselines.

B. Corpus Assembly

There are 16 sources in total that were chosen so the analysis stays within this group and does not say it covers all the literature. The sources were kept if they looked at or tested the quality of code, in least one of the three areas and were published between 2023 and 2026 as journal articles, preprints, theses or conference papers. These sources are split into four categories: surveys and systematic reviews (Anand et al. 2024; Garousi et al. 2026; Lyu et al. 2025; Patel et al., 2026; Truong et al. 2026); Large or multi-model empirical studies (Asfandyar et al. 2025; Chong et al. 2024; Molison et al. 2025; Sabra et al. 2025); Small comparative studies and theses (Azbarka et al. 2025; Dossas, 2025; Gontiya, 2025; Kantek, 2023; Pandya & Soni 2026; van Kleij, 2026);. A special benchmark paper (Huang et al. 2024).

C. Data Extraction

For each source the following were recorded: models or tools that were tested programming languages used the set of tasks and how big they were the human baseline, the test and analysis instruments along with their versions if mentioned the metrics used how many times each task was generated the statistical tests applied, the main numerical results and the limitations that the authors admitted.

D. Quality Appraisal

We examined each source to see if it was peer-reviewed and from which venue it came. We also checked each source for the tool baseline that was used the sample size and how times each task was generated. Then we looked at each source to see how the statistics were handled whether the measurement tools were valid and if the numbers reported were consistent, with each other. Any findings that a source includes from publications like industry statistics mentioned in a summary paper are marked as secondary.

E. Synthesis and Recalculation

The findings were grouped thematically by dimension. Then put together across dimensions to answer RQ3. The numerical results are shown as they were originally published. When a source had tabled data that made it possible to check a number that number was recalculated. Any evidence that a source uses from publications is marked as secondary.

IV. RESULTS

Results are organized by the three focal dimensions, followed by security, the measurement-instrument problem, the interdependence of the dimensions (RQ3), improvement strategies, and human factors.

A. Functional Correctness

Benchmarks. Correctness is the dimension. Function-level (HumanEval, MBPP) program-level (APPS, CodeContests) class-level (ClassEval) and repository-level (DevEval, EvoCodeBench) benchmarks all use hidden unit tests (Truong et al. 2026). Repository-level evaluation shows the limits of dependency-based metrics: models may create functions that do not exist of using the ones that do exist and recall@k cannot spot wrong integration. Patel et al. (2026) Report, from sources that models with HumanEval scores differ sharply on the evolved EvoEval benchmark, which suggests contamination; this should be checked against the original study.

Results on small tasks. On tasks that don't need much outside help current models are mostly good. GPT-4o solved 87.6% of 202 LeetCode problems (Chong et al. 2024). OpenAIs o4-mini solved 96% of LeetCode Java tasks 95% of medium tasks and 93% of hard tasks (about 95% overall; Dossas, 2025). The pass@1 score on Java assignments went from 60.43% (OpenCoder-8B) to 77.04% (Claude Sonnet 4; Sabra et al. 2025). Across tools ChatGPT got 8 out of 10 first-phase problems right while Copilot got 2 Claude got 1 and Gemini got 0. On 30 problems ChatGPT had 17 fully correct solutions, compared to 12, from Copilot. The difference wasn't statistically significant (Mann-Whitney p =.215; Azbarka et al. 2025). A previous study showed that Codex compiled 91%, AlphaCode 86% and CodeT5 74% with pass@5 of 76% 69% and 58% (Pandya & Soni 2026). Performance dropped for problems that had functions or long contexts.

Where correctness breaks down. Chong and colleagues (2024) report a failure: the SHA-1 implementation of GPT-4o compiled but generated incorrect digests for all test vectors. In an experiment success at filling a buffer size dropped from 99.3 percent for a constant to 1.5 percent when the size required multiplying two floats. In some studies correctness decreases as the task gets harder (Asfandyar and others 2025; Soni, 2026) but not in every case. Dossas (2025) found that acceptance rates remained nearly flat across difficulty levels for a recent reasoning model. However, code-smell counts increased with difficulty. The pattern indicates that as models get better difficulty shows up first in quality than, in pass rates.

Correctness does not imply quality. Sabra et al. (2025) Report no correlation between a model's pass@1 and the number of SonarQube issues in the solutions that passed. Models still averaged 1.45 to 2.11 issues per passing task and the model with the pass rate had the fewest issues per passing task. Molison et al. (2025) Show the converse: their fine-tuned GPT-4o produced the code by static metrics (issue rates around 0.07% to 0.09% on introductory and competition tasks) yet its Pass@1 on a sample of 15 solutions was.47 against.87 for few-shot and 1.0, for human code. Static quality scores can therefore improve while function deteriorates so they should be reported conditional on correctness as Sabra et al. (2025) do.

Summary. Correctness on small tasks is high and has largely saturated as a discriminating signal. Residual variance lies in task difficulty, specification complexity, and silent error modes, and unit-test pass rates alone are insufficient evidence of production readiness.

B. Maintainability

Evaluation infrastructure. Maintainability is the benchmarked dimension (Truong et al. 2026). The corpus relies on SonarQube default rule sets (Kantek, 2023; Molison et al. 2025; Sabra et al. 2025) the SonarQube, PMD and Checkstyle trio (Dossas, 2025) complexity and Halstead metrics (Gontiya, 2025) and hand-built readability checklists (Azbarka et al. 2025).

Evidence that AI code is better. Molison et al. (2025) Analyzed APPS tasks with SonarQubes default Python profile. Found fewer bugs and code smells in GPT-4o output for most task types and lower estimated remediation effort (3.75 minutes for zero-shot introductory code vs. 7.20 For human code).

Evidence that AI code is worse. Asfandyar et al. (2025) Report a cyclomatic complexity of 8.2 vs. 5.4 for human code (about 34% higher) duplication of 14.1% vs. 6.8%. A Maintainability Index of 68.5 vs. 82.1 and 5.22 vs. 2.12 Code smells per 100 lines. Chong et al. (2024) Found that GPT-4os C code was shorter and had absolute complexity but higher complexity per line (0.33 vs. 0.28 On LeetCode; 0.21 vs. 0.17 On data structures) and attributed the lower absolute figure to bare-bones code that omits defensive constructs. Dossas (2025) found that o4-minis Java code although accepted by LeetCode in 95% of cases was not free of smells: the analysers flagged line-length violations, missing braces, magic numbers, cognitive-complexity findings, unused assignments and object creation inside loops and the

number of smells rose with problem difficulty. Therefore, fixing issues may sometimes cost more, than writing the code manually.

Evidence that results depend on model and task. In a two-way ANOVA involving four tools and six problems, the problem type accounted for 90% of the variation in code length. Cyclomatic complexity showed no difference between the models ($p = .864$ $\eta^2 = .007$). However, the maintainability index did show a difference with code Llama being the most maintainable and copilot being the least (Gontiya, 2025). Kantek (2023) ranked copilot first among four tools across 15 quality and security metrics scoring 107 points compared to 96 for the second-place tool.

Enterprise code. Van Kleij (2026) evaluated refactoring of a VB.NET code base across 530 annotated tasks. A deterministic rule-based engine achieved a 95% success rate on house-rule violations but did not improve structural metrics. Gemini was the system to resolve any cyclomatic complexity issues handling 15% of them. A LoRA tuned StarCoder2-3B model did not perform significantly better than its prompt-only baseline. This suggests that improving maintainability in settings cannot be achieved through model adaptation alone.

Longitudinal evidence. The strongest claims about long-term maintainability—such as declining refactoring rates, increasing code duplication and higher churn—come from repository analyses and industry reports. These are found in this corpus through Patel et al. (2026). These findings are secondary, in nature and should be interpreted by referring directly to the original sources.

Summary. The apparent contradiction (AI code better in Molison et al., 2025; worse in Asfandyar et al., 2025, and Chong et al., 2024) is largely explained by differences in baseline, analyzer, language, and normalization, examined in the methodological appraisal that follows. What is consistent is a profile: AI code tends to be compact and locally tidy but omits validation and error handling and accumulates smells as tasks grow harder, and its quality varies with model and task more than with any universal AI effect.

C. Performance Efficiency

Benchmarks. EffiBench has 1,000 LeetCode problems that were chosen because they are sensitive to efficiency. Each of these problems comes with a solution that has the runtime on the LeetCode leaderboard (Huang et al., 2024). When looking at 42 language models (35 open-source and 7 closed-source) the code that was generated was usually not as efficient as the solution that was provided. GPT-4 code took on average 3.12 times longer to run than the solution that was provided. In some cases, it took 13.89 times longer. Used 43.92 times more memory. The efficiency of the code was not connected to whether the code was correct. For example, GPT-4-turbo-preview passed tests than GPT-4 but the code it generated was less efficient (Huang et al., 2024). ENAMEL, which was reviewed by Truong et al. (2026) Improves on pass@k by introducing eff@k. It handles cases where the code runs out of time by treating them as censored data. It also provides reference solutions because the original solutions can be inefficient (for example using O(n) when O (log n) is possible). EffiBench also limits the time, for runs, which can make efficiency seem better than it's. On the improvement side a method called execution-feedback prompting (EFFI-LEARNER) is said to have reduced the time it takes for StarCoder2-15B to run on EffiBench by 87.1%. The memory it uses by 90.8% (Truong et al., 2026). This indicates significant scope for improvement when efficiency is a primary consideration.

Other empirical evidence. Asfandyar et al. (2025) found that AI-generated code had overheads of 15.2% in time and 18.5% in memory on basic tasks. These numbers increased to 40.1% in time, 34.8% in memory on advanced tasks. Overall, the average time overhead was 27.3% and memory overhead was 25.1%. The study also showed that in 28% of tasks 65% of intermediate tasks and 72% of advanced tasks, the AI chose suboptimal algorithms. Across all levels 52% of the time, the algorithm choice was not the best. For example, in sorting tasks the AI used sorting when faster O (n log n) options were available. The link between task complexity and time overhead was strong with a Spearman correlation coefficient of 78.

Azbarka et al. (2025) Looked at time and space complexity by analysing code without running it. They found no difference between ChatGPT and Copilot. The p-value for time complexity was 575 and for space complexity it was 405. This suggests that based on code analysis the two models performed similarly.

Dossas (2025) examined the Java code produced by o4-mini and found performance-related issues. These included creating objects inside loops and using loops

with break or continue statements. These patterns are known as performance smells. The findings show that static analysers can detect efficiency problems that execution-based benchmarks might miss. This highlights a gap, in how performance's evaluated for AI-generated code.

Measurement validity. Efficiency measurements depend on hardware, compiler behavior, runtime configuration, and input distributions, which complicates comparison across studies (Truong et al., 2026). Reference baselines, timeout handling, and language-runtime effects matter as much as the generated code itself.

Summary. Performance deficits are real, widen with task difficulty (Asfandyar et al., 2025; Huang et al., 2024), and are largely independent of whether the code passes tests. (Huang et al., 2024). Truong et al. (2026) identified only two performance benchmarks among roughly 30 evaluation frameworks, and none addresses system-level resource behavior.

D. Interdependence of the Dimensions

RQ3 asks whether correctness predicts quality on the dimensions. Five lines of evidence from the corpus suggest that it does not or only weakly. First, in a Java study involving five language models pass@1 showed no correlation with SonarQube issue counts among solutions that passed (Sabra et al. 2025). Second, a tuned model generated code that scored the cleanest according to static metrics yet had the lowest sampled Pass@1 (Molison et al. 2025). Third, efficiency did not align with correctness: GPT-4-turbo-preview had a pass@1 than GPT-4 but its code was less efficient (Huang et al., 2024). Fourth, o4-mini's acceptance rate stayed nearly flat across difficulty levels while smell counts increased (Dossas, 2025). Fifth, in the comparison that included recruited developers, degradation in maintainability, security and performance each correlated with task difficulty (Spearman ρ =.65 to.78) though this study did not control for correctness (Asfandyar et al. 2025). There is also a trade-off within dimensions: defensive programming features like input validation increase complexity per line and code length but help prevent crashes (Chong et al. 2024). So, a lower complexity score might mean protection rather than better design. All of this suggests that the different quality dimensions are only partly related and that differences in quality appear before differences, in pass rate as models get better.

E. Improvement Strategies

Prompting. Prompt design, task specification and developer expertise are the factors often mentioned as having a strong impact on quality (Garousi et al. 2026). Truong et al. (2026) Look at chain-of-thought and execution-feedback prompting as ways to make models perform better. Pandya and Soni (2026) use the prompts across different models to focus only on how architecture affects outcomes. This method helps make comparisons easier. It doesn't tell us how much better or worse results could be with different prompting methods.

Feedback loops are a concern. In one study, when GPT-4o was asked to fix problems in C code the number of malloc overflows dropped from 40 to 24 and null dereferences went down from 3 to 2. The number of array-index errors went up from 15 to 18. Worse four new malloc overflows appeared in 30 files that were previously clean (Chong et al. 2024). Another report shows that after five rounds of refinement critical vulnerabilities rose by 37.6% (Patel et al. 2026). Anand et al. (2024) Point out that feedback loops are a trend, in both code repair and code generation research. That makes their reliability a big unanswered question. All the sources suggest that when using iterative repair, it's important to run regression checks to avoid consequences.

Fine-tuning and adaptation. Tuning research focuses on making sure programs work correctly by using supervised fine-tuning and execution-feedback reinforcement learning. This is because correctness provides a signal that can be automated and is not very changeable. Security, efficiency and especially maintainability are not studied much because their signals are not easy to measure are based on opinions or depend on the environment (Truong et al. 2026). Small studies show the trade-offs: tuning removed serious issues but made Pass@1 worse when using 240 training examples (Molison et al. 2025) and it did not improve much over prompting in a companys VB.NET setting (van Kleij, 2026).

Hybrids. Lyu et al. (2025) Say that automated program repair can provide reliable code along with proof that it is reliable. Van Kleij (2026) says that hybrid pipelines should use rule-based steps to enforce style and let LLM transformations be used when needed,

controlled by static analysis and human checks. Dossas (2025) supports combining analysers in a way. Truong et al. (2026) warn that using agents to manage tasks cannot fix the problem of measuring aspects because, without clear signals it is not clear if more steps make the code better or just change it.

F. Human Factors and Process

Garousi and colleagues in 2026 say that the quality of code in AI-assisted development is a result of how prompts are designed, how tasks are described, how skilled the developers are and how people work with AI. They also say that the best ways to do this include using prompts testing many times checking work by humans and combining human and AI work. Lyu and others in 2025 see trust as the issue when putting AI into use. They believe that programmers will need to change their jobs. Dossas in 2025 suggests using AI for tasks that are clearly defined and checking the quality all the time. Patel and others in 2026 suggest a seven-step process that includes stopping work if the risk is high and making sure all tests pass and getting a quality score of at least 80. These rules are not tested in real life. Some reports show that people who use AI tools trust them much. They write code that's not as safe but think it is safe. This happens in studies by Chong in 2024 Patel in 2026 and Sabra, in 2025. Current tests look at the code, not the process of writing it. How much work reviewers do, how well they find mistakes and how good their choices are are rarely looked at.

V. METHODOLOGICAL APPRAISAL OF THE EVIDENCE

A. The Baseline Problem

The direction of the AI-versus-human result changes with the human baseline (Table 2).

Competition solutions are written to pass tests quickly, not to be maintained (Molison et al., 2025), whereas OpenBSD is hardened production code (Chong et al., 2024). A single AI-versus-human effect is not identifiable independently of baseline provenance, and future studies should report and, where possible, stratify it.

B. Construct Validity

Static proxies do not reflect maintainability. Metrics like complexity, duplication and code smell can give a sense of understandability. They miss key aspects like system architecture, the effort needed to make changes and how the code evolves over time (Truong et al. 2026).

Analyzers often disagree. The choice of tool and how rules are set up can shift the number of reported issues by much as ten times (Dossas, 2025) which means studies relying on just one tool are not reliable.

Normalization is important. Absolute complexity tends to be lower in AI code but complexity per line ends up being higher (Chong et al., 2024). Measures like issues per file, issues per thousand lines and issues per passing task (Sabra et al. 2025) each highlight things—no single metric tells the full story.

Terse code is not necessarily better code. Chong et al. (2024) Say that lower absolute complexity comes from missing guard clauses. Therefore, a lower score can mean protections.

Quality must be judged based on function. If a model fails or produces code it can still score well Molison et al. (2025). Sabra et al. (2025) show that this can happen.

Vendor involvement is an issue. Sabra et al. (2025) Wrote the industry study and the study was made by the maker of the analyzer it recommends. This does not automatically bias the results. The fact should be disclosed.

Task representativeness and contamination. LeetCode and APPS tasks dominate; they are public and can be memorized. Chong et al. (2024) Saw that GPT-4o reproduced human GitHub implementations in 2 of 18 data-structure tasks.

C. Statistical Practice

Pseudo-replication. Asfandyar et al. (2025) compare 642 AI samples (three per model and task) with 107 human samples and report a chi-square test with 12 degrees of freedom for what is described as a two-group comparison; repeated generations for the same task are not independent, so analyses should aggregate by task or use mixed models. Reported Cramér's V values for individual security categories are small (about .09 to .12).

Small samples. Examples include 30 problems with single generations and the free tier of one tool (Azbarka et al., 2025), six problems (Gontiya, 2025), 15 sampled solutions per condition for Pass@1 (Molison et al., 2025), and 100 tasks with one model (Dossas, 2025).

Non-determinism. Sampling temperature, model versions, and hardware are often under-reported, and deterministic settings do not guarantee identical outputs (Truong et al., 2026).
Effect sizes. Cliff's delta is used well by Asfandyar et al. (2025) and Molison et al. (2025); most other studies report significance without effect sizes.

D. Evidence Quality and Consistency Flags
The observations in Table 3 were made while reading the sources and should be verified before any figure is reused.

E. Temporal Validity
All evaluated models predate the current generation, benchmark saturation on HumanEval- class tasks reduce their discriminating power, and the large longitudinal claims in Patel et al. (2026) come from industry reports. Findings should be treated as time-stamped and replicated on current models.

VI. DISCUSSION

A. Principal Findings
Regarding RQ1, correctness is benchmarked with mature unit-test suites and the pass@k family, whereas maintainability relies on heterogeneous static proxies and efficiency on a small number of dedicated benchmarks (Huang et al., 2024; Truong et al., 2026). Regarding RQ2, AI-generated code tends to be compact and locally tidy but omits validation and error handling, accumulates smells as tasks grow harder, and can run several times slower than expert solutions (Asfandyar et al., 2025; Chong et al., 2024; Dossas, 2025; Huang et al., 2024). Regarding RQ3, correctness is a weak predictor of other qualities. Regarding RQ4, the evidence is weakened by baseline heterogeneity, analyzer disagreement, small samples, and internal inconsistencies in several sources.

B. Reconciling Conflicting Evidence
The conflict between studies that say AI code is better (Molison et al., 2025) and studies that say AI code is worse (Asfandyar et al., 2025; Chong et al., 2024) can be explained without thinking that models act differently. The baselines come from sources (contest solutions, hardened production code, recruited developers). The analyzers use rules and cover different areas. The number of issues is adjusted based on each file per thousand lines or per passing task. Only some studies take quality into account when looking at functional success (Table 2). Because results from an analyzer can vary by a large amount (Dossas, 2025) comparing absolute issue counts across studies is not reliable. The real takeaway is that a benchmark needs to manage these factors of just showing one AI, versus human result.

C. Proposed Protocol: Correctness-conditioned Multidimensional Benchmarking
The evidence shows three design principles. Condition: quality metrics should be calculated for solutions that pass the tests because otherwise code that fails or is very simple can get high scores (Molison et al. 2025; Sabra et al. 2025). Triangulate: each dimension should be measured using more than one method and the agreement should be reported (Dossas, 2025). Stratify: results should be separated by task difficulty, language and human-baseline level (Asfandyar et al. 2025; Chong et al., 2024; Molison et al. 2025). The proposed protocol, the Correctness-conditioned multidimensional benchmark protocol applies these principles in five layers
The quality-conditioned pass metric. Let n samples be generated per task, let c be the number that pass all tests, and let $m \leq c$ be the number that additionally satisfy the quality gates for maintainability, efficiency, and security. The standard metric (pass@k) and the proposed metric (Q@k) are defined as

$$\text{pass@k} = E[\ 1 - C(n - c, k) / C(n, k)\ ]$$

$$\text{Q@k} = E[\ 1 - C(n - m, k) / C(n, k)\ ]$$

where C(a, b) is the binomial coefficient and the expectation is taken over tasks.
Because $m \leq c$, $Q@k \leq pass@k$, and the quality gap $\Delta k = pass@k - Q@k$ measures how much apparent correctness is not production-ready. Q@k generalizes the logic of secure-pass@k (Truong et al., 2026) from security to the other dimensions. The gates are parameters rather than fixed constants: the recommended practice is to define them relative to the matched human-baseline tier, for example complexity per KLOC and severity-weighted smell density no worse than the tier median on at least two analyzers, a normalized execution time (NET) below a stated multiple of the expert reference, and no high-severity CWE finding from static and dynamic analysis. Reporting results for several gate settings avoids

dependence on arbitrary thresholds. Table 4 gives the accompanying minimum reporting protocol.

D. Proposed Validation Study

CC-MBP generates testable hypotheses (Table 6) that define the empirical phase of the research.

The proposed design relies on existing models using at least ten samples for each task under the specified sampling conditions. Tasks are divided into three categories: function-level problems, change-based maintenance tasks such as modifying, extending and refactoring existing code and repository-level tasks. The languages involved include Python, Java, C and a managed enterprise language like C#. C# is represented in one study within the corpus (Kantek, 2023). For each language at least two analyzers would be used. Efficiency would be measured using a testing environment with repeated runs and an expert reference baseline. Human baselines would come from at least three different provenance tiers to ensure diversity. Analyzers would consider generations as nested within tasks, report effect sizes with confidence intervals and account for multiple comparisons. All code prompts, generated outputs and analyzer configurations would be made available.

E. Implications

For researchers the findings suggest that they should focus on reporting quality that depends on correctness, mention the analyzers and rulesets used and explain where the baseline came from. For practitioners a high pass rate does not mean that the code is ready for use; teams need to check AI-generated code using analysis, dynamic testing and reviews and they should also check the output from automated repair loops again (Chong et al., 2024; Lyu et al., 2025; Patel et al., 2026). For educators the idea that fixing issues, in code written by machines can take time than writing it by hand. (Dossas, 2025) shows that it is important to teach students to check and make sure the code is correct.

F. Limitations

Four limitations affect the study. First the collection was provided of being carefully gathered so some important work might be missing; an improvement would involve doing a clear search across IEEE Xplore, ACM Digital Library, Scopus and arXiv with a phrase that includes terms for AI-generated code, benchmarking or evaluation and the quality aspects along with careful checking. Second some of the sources are preprints, theses or journals that're not well known and some have problems (Table 3). Third all the models that were tested are older than the ones so the numbers are only valid, for the time they were collected. Fourth CC-MBP and the Q@k metric are ideas and have limitations. How sensitive they are and how well they can tell differences are still not known.

VII. CONCLUSION AND FUTURE WORK

Across 16 sources one conclusion is clear. Functional correctness, the measured dimension is a weak guide to the maintainability, efficiency and security of AI-generated code (Dossas, 2025; Huang et al., 2024; Sabra et al. 2025; Truong et al. 2026). The rest of the inconsistency in the literature comes from methods. Baselines, languages, analyzers and normalization decide whether AI-generated code looks better or worse than human code (Asfandyar et al. 2025; Chong et al., 2024; Molison et al. 2025) and the analyzers themselves agree weakly (Dossas, 2025). Therefore, this article proposes a protocol that conditions quality on functional correctness triangulates instruments, stratifies baselines and reports a joint Q@k metric. Future work includes (a) a search to extend the corpus, (b) the validation study in Table 6 (c) a maintainability benchmark based on change tasks (d) a feedback-loop evaluation that reports per-iteration regressions, (e) repository-level and agentic evaluation, with reviewer-in-the-loop measures and (f) coverage of managed enterprise languages and legacy codebases.

Tables

Table 1. Profile of the Reviewed Corpus

Source	Type	Focus
Anand et al. (2024)	Survey	27 papers on program repair and code generation
Asfandyar et al. (2025)	Empirical (journal)	GPT-4 and Claude 3 vs. 20 developers; 107 tasks
Azbarka et al. (2025)	Empirical (student journal)	ChatGPT, Copilot, Claude, Gemini; 40 LeetCode problems
Chong et al. (2024)	Empirical (preprint)	GPT-4o vs. human C code; 202 LeetCode problems and 18 data-structure files

Dossas (2025)	Master's thesis	Java analyzers PMD, SonarQube, Checkstyle compared; o4-mini code on 100 LeetCode tasks
Garousi et al. (2026)	Systematic review	24 primary studies on factors affecting quality
Gontiya (2025)	Master's thesis	4 models, 6 LeetCode problems; ANOVA on quality metrics
Huang et al. (2024)	Benchmark	EffiBench: 1,000 efficiency-critical problems, 42 LLMs
Kantek (2023)	Master's thesis	Copilot, Tabnine, ChatGPT, CodeGeeX; 860 programs in 6 languages
Lyu et al. (2025)	Survey and position paper	Trust in generated code; program repair
Molison et al. (2025)	Empirical (preprint)	GPT-4o zero-shot, few-shot, and fine-tuned vs. APPS human code
Pandya & Soni (2026)	Comparative study (journal)	Codex, CodeT5, AlphaCode on HumanEval, APPS, CodeXGLUE
Patel et al. (2026)	Literature synthesis	24 sources; governance framework
Sabra et al. (2025)	Empirical (industry preprint)	5 LLMs; 4,442 Java tasks
Truong et al. (2026)	Survey	Quality taxonomy; about 30 evaluation frameworks
van Kleij (2026)	Master's thesis	LLM refactoring of VB.NET code; 530 tasks

Table 2. Human Baselines and Direction of Results

Study	Human baseline	Language and instrument	Direction of result
Chong et al. (2024)	Open-source repositories (including OpenBSD) and LeetCode solutions merged into a repository	C; Clang, AFL, SCC	AI worse on security and complexity per line
Molison et al. (2025)	APPS solutions mined from competition websites	Python; SonarQube default profile	AI better on bugs, smells, effort; tuned model lower Pass@1
Asfandyar et al. (2025)	20 recruited developers	Python; SonarQube, Code Climate, Bandit	AI worse on all three dimensions
Dossas (2025)	None for AI code (human open-source Java used only to compare analyzers)	Java; PMD, SonarQube, Checkstyle	Smells present and rising with difficulty
Gontiya (2025)	None (model vs. model)	Python; complexity and MI metrics	Model-dependent

Table 3. Consistency Flags in Reviewed Sources

Source	Observation	Implication
Chong et al. (2024)	Introduction cites 21 unsolved LeetCode tasks (20 medium or hard), whereas 12.4% of 202 is 25, as listed in the paper's Table 2; fuzzing totals are not like-for-like	Use table-level data, not headline percentages
Molison et al. (2025)	Few-shot Pass@1 is .86 in the introduction and .87 in Table V; Pass@1 rests on 15 solutions per approach	Treat performance comparison as indicative
Asfandyar et al. (2025)	Developers have at least 10 years of experience in the abstract but 6.2 years on average in the method; text says all AI samples had a vulnerability while tables give 22.1%; some table values are garbled	Verify before citing numbers
Azbarka et al. (2025)	Efficiency judged by hand without execution; first-phase readability (91%–96%) differs from second-phase means (82%–84%)	Subjective, small-sample evidence
Pandya & Soni (2026)	Six-page paper; compares Codex, CodeT5, and AlphaCode without stating how the systems were accessed or run, sample sizes, or variance; complexity and security reported as qualitative labels	Results not reproducible; low weight
Patel et al. (2026)	In-text numbers do not match the reference list (e.g., the survey cited as reference 1 resolves to an unrelated image-mining paper); many statistics come from industry blogs	Use as a pointer to primary sources only
Dossas (2025)	Default analyzer configurations; single model (o4-mini); tool comparison on human projects	Indicative; replicate with configured rulesets

Table 4. Proposed Minimum Reporting Protocol for Code Generation Benchmarks

Element	Minimum requirement
Model and sampling	Exact model version and date; temperature, top-p, number of samples n; seeds where available; how the model was accessed
Tasks	Provenance, difficulty distribution, contamination check
Correctness	pass@1 with confidence intervals; graded outcomes (correct, partial, incorrect, compile error, runtime error)
Quality metrics	Reported conditional on passing solutions; per task, per thousand lines, and severity-weighted
Performance	Hardware, repetitions, timeout handling, expert reference solution
Static tools	Tools, versions, and rule profiles; at least two analyzers where feasible; inter-tool agreement; manually validated sample
Baseline	Origin and selection of human code
Statistics	Unit of analysis, treatment of repeated generations, effect sizes
Artifacts	Code, prompts, and generated outputs released

Table 5. Layers of the Proposed Correctness-Conditioned Multidimensional Benchmark Protocol (CC-MBP)

Layer	Purpose	Metrics and instruments	Motivating evidence
1. Correctness gate	Establish functional success	pass@k with confidence intervals; graded outcomes; stronger test suites	Azbarka et al. (2025); Truong et al. (2026)
2. Conditional quality profile	Measure maintainability, efficiency, and security on passing samples only	Complexity per KLOC; severity-weighted smell density; duplication; NET and NMU against an expert reference; CWE-mapped findings	Huang et al. (2024); Sabra et al. (2025)
3. Instrument triangulation	Control analyzer dependence	At least two analyzers per language with documented rulesets; inter-tool agreement; dynamic analysis where feasible	Chong et al. (2024); Dossas (2025)
4. Baseline and difficulty stratification	Make AI-versus-human claims identifiable	Human baseline tiers (contest, open-source, hardened production); difficulty strata; language strata	Asfandyar et al. (2025); Molison et al. (2025)
5. Joint metric and reporting	Summarize and enable replication	Q@k and quality gap; minimum reporting protocol (Table 4)	Truong et al. (2026)

Note. CC-MBP is a proposal derived from the synthesis and has not been empirically validated.

Table 6. Hypotheses for the Empirical Validation of CC-MBP

Hypothesis	Statement	Motivating evidence	Planned test
H1	$\Delta 1 > 0$ for every model, and $\Delta 1$ increases with task difficulty	Dossas (2025); Huang et al. (2024); Sabra et al. (2025)	Mixed-effects model with difficulty as a fixed effect and task and model as random effects
H2	Agreement between analyzers on AI-generated code is low	Dossas (2025)	Spearman or Kendall correlation of per-solution issue counts; Cohen's kappa for severity classes
H3	Model rankings by pass@1 differ from rankings by Q@1	Molison et al. (2025); Sabra et al. (2025)	Kendall's tau with bootstrap confidence intervals
H4	The AI-versus-human quality difference varies by human-baseline tier	Asfandyar et al. (2025); Chong et al. (2024); Molison et al. (2025)	Model-by-tier interaction with effect sizes
H5	Iterative LLM repair does not monotonically reduce gate violations	Chong et al. (2024); Patel et al. (2026)	Repeated-measures analysis across repair iterations

REFERENCES

[1] Anand, N. Yadav, A. Gupta, and S. Bajaj, "A comprehensive survey of AI-driven advancements and techniques in automated program repair and code generation," *arXiv preprint* arXiv:2411.07586, 2024. [Online]. Available: https://arxiv.org/abs/2411.07586

[2] Asfandyar, M. W. Akhtar, and H. S. Khalil, "Evaluating the impact of AI code generated on software quality: An observed study of maintainability, security, and performance," *Spectrum of Engineering Sciences*, vol. 3, no. 12, pp. 1379–1389, 2025, doi: 10.5281/zenodo.18172279.

[3] S. Azbarka, A. Mughrabi, M. Sabbagh, and I. Roth, "Evaluation of the code quality generated by generative AI," *Information Systems Student Research Journal*, vol. 2, Art. no. 1, pp. 1–17, 2025.

[4] J. Chong, Z. Yao, and I. Neamtiu, "Artificial-intelligence generated code considered harmful: A road map for secure and high-quality code generation," *arXiv preprint* arXiv:2409.19182, 2024. [Online]. Available: https://arxiv.org/abs/2409.19182

[5] S. Dossas, "Software quality evaluation: A comparative study of code quality analysis tools and an assessment of AI-generated code," M.S. thesis, Univ. of Macedonia, 2025.

[6] V. Garousi, Z. Aliyeva, and E. Tüzün, "Factors influencing the quality of AI-generated code: A synthesis of empirical evidence," *arXiv preprint* arXiv:2603.25146, 2026. [Online]. Available: https://arxiv.org/abs/2603.25146

[7] R. Gontiya, "Analyzing code generation by AI models: An ANOVA-based study of quality, consistency, and composite ranking," M.S. thesis, Univ. of Alabama in Huntsville, 2025. [Online]. Available: https://louis.uah.edu/uah-theses/770

[8] Huang, Y. Qing, H. Cui, W. Shang, and J. M. Zhang, "EffiBench: Benchmarking the efficiency of automatically generated code," in *Advances in Neural Information Processing Systems (NeurIPS), Datasets and Benchmarks Track*, 2024.

[9] P. Kantek, "AI-driven software development: Source code quality," M.S. thesis, Faculty of Informatics, Brno, 2023.

[10] M. R. Lyu, B. Ray, A. Roychoudhury, S. H. Tan, and P. Thongtanunam, "Automatic programming: Large language models and beyond," *ACM Transactions on Software Engineering and Methodology*, vol. 34, no. 5, Art. no. 140, 2025, doi: 10.1145/3708519.

[11] S. Molison, M. Moraes, G. Melo, F. Santos, and W. K. G. Assunção, "Is LLM-generated code more maintainable and reliable than human-written code?" *arXiv preprint* arXiv:2508.00700, 2025. [Online]. Available: https://arxiv.org/abs/2508.00700

[12] H. G. Pandya and N. Soni, "Exploring and evaluating AI algorithm for automated code generation," *International Journal of Engineering Research & Technology*, vol. 15, no. 4, Art. no. IJERTV15IS040913, 2026.

[13] J. Patel, A. Patel, H. Prajapati, K. Rathod, and M. Suthar, "Evaluating the reliability, security, and quality of AI-generated code in modern software engineering," *International Journal of Novel Trends and Innovation*, vol. 4, no. 5, pp. a482–a496, 2026.

[14] Sabra, O. Schmitt, and J. Tyler, "Assessing the quality and security of AI-generated code: A quantitative analysis," *arXiv preprint* arXiv:2508.14727, 2025. [Online]. Available: https://arxiv.org/abs/2508.14727

[15] J. Truong, V. Nguyen, and T. T. Nguyen, "An overview of evaluation and enhancement methods for code generation by large language models," *Information and Software Technology*, vol. 197, Art. no. 108185, 2026, doi: 10.1016/j.infsof.2026.108185.

[16] L. van Kleij, "AI-driven code quality assessment and enhancement," M.S. thesis, Utrecht Univ., 2026.