

Characterizing the Quality Profile of AI-Generated C++ in Production

Michael Tran, Fred Lewis, Kun Yang, Saksham Thakur, Aditya Kini, Aditya Patil, Milad Hashemi, Parthasarathy Ranganathan¹

¹Google, Mountain View, USA

The widespread integration of AI coding assistants offers undeniable boosts to engineering velocity. Yet, recent studies point to a growing trade-off, revealing persistent challenges with code quality and maintainability. Industry leaders, including frontier AI labs, echo these concerns. As large language models are increasingly relied upon to author production code, understanding their impact on shipped software quality has become a critical priority. However, assessing these effects in industrial workflows remains difficult due to challenges with observability. We study the impact of AI-generated code on production quality within a large enterprise that operates multiple global products relied upon by billions of users daily. Driven by this immense scale and user trust, the organization deeply values code quality and has built thorough observability for every line of code deployed into production, enabling us to overcome industry-wide measurement barriers to assess these effects.

This study presents a large-scale empirical analysis of AI-generated C++ code from April 2025 to April 2026, tracking 3.52 million code changes across this enterprise’s massive, brownfield codebase. The core purpose is to understand the quality, performance, and maintenance characteristics of AI-generated code compared to human-written code in a production environment of this scale. We find that AI-generated C++ code has a distinct quality profile, showing a higher rate of interface and coupling burdens, copy and allocation overheads, and a stylistic reliance on explicit loops over optimized standard APIs. These issues translate into tangible downstream costs, including increased review effort and a 5-8% increase in compute resource consumption. However, we also demonstrate that providing models with targeted, taxonomy-informed feedback can mitigate these effects, leading to an 11.1% reduction in targeted static analysis warnings and improved computational efficiency.

Keywords: AI-generated code, software quality taxonomy, authoring-time provenance, static analysis, code review, empirical software engineering

1. Introduction

Generative AI has moved from a laboratory capability to a routine part of software development, with code completion, text-to-code generation, and agentic editing now embedded in programming workflows. Field and enterprise studies report speedups from these tools [27, 29], and case-study evidence reports similar productivity gains in everyday development [11]. Diary and interaction studies also report trust, steering, and verification costs when developers use generated code in real tasks [6, 8, 9, 24, 30]. In our setting, AI-generated changes comprised nearly 70% of submitted code with known authoring provenance by the close of our study window (April 1, 2025–April 1, 2026). This includes code generated through a diverse set of interaction modes, ranging from low-latency code completion models to complex multi-turn agent pipelines. Generated code can exhibit correctness, security, and maintainability weaknesses [28, 36, 37, 44]. Other studies report performance, efficiency, and API-modernization weaknesses [5, 12, 21, 32, 43]. Completion rate or task speed leaves out production effects, including whether generated code survives review, creates

maintenance burden, or contributes to compute cost at deployment scale [17, 33, 34].

Evaluating AI-generated code in production requires a fundamentally different approach to measuring code quality. First, evaluation systems must associate generated code edits across a complex tooling ecosystem, spanning code reviews, tests, static analyzers, and deployment pipelines [33, 34]. Furthermore, these measurements must track the code across a lifetime of edits, capturing how the generated lines are accepted, heavily revised, moved, or removed before final submission [13, 17, 42]. Because a warning may disappear during review or remain in the submitted revision and a function-level cost signal may appear only after deployment, measuring quality in this setting requires fine-grained authorship and aligned line-, change-, and function-level outcomes. In large production environments, C++ remains central in performance-sensitive systems. In these contexts, choices around memory movement, container usage, API modernity, and low-level abstractions affect both maintainability and runtime efficiency. C++ is also a useful stress case because production code combines mature static analysis with local performance-relevant idioms, such as copying versus moving, declaration placement, container construction, and API selection. Indeed, LLM-automated optimization frameworks like ECO [22] are already being deployed specifically to refactor these exact idioms at warehouse scale. Prior C++ and non-functional studies motivate the same concern because correctness alone does not characterize engineering burden in performance-sensitive systems [12, 32]. We need to know whether AI-generated C++ has a recognizable issue profile, whether that profile appears after review and deployment, and whether it is large enough to justify intervention.

Most prior work does not observe the production lifecycle because existing studies evaluate generated code in controlled prompting settings, benchmark tasks, repository snapshots, or online evaluations on real-world completions that stop before code reaches review [15, 18, 19, 23, 46]. Non-functional studies usually analyze standalone tasks whose authorship is known by construction [5, 12, 21, 32, 43]. In production repositories, authorship is often mixed within the same change, while review, static-analysis, and compute effects are observed at line, change, and function levels, complicating measurement of where AI-generated code appears, what issues characterize it, and which categories are most relevant downstream. We observe AI generation during authoring and follow the generated code through static findings, code review, and function-level compute outcomes. Interaction- and review-centric studies provide complementary evidence about steering costs, sense-making, AI-assisted review behavior, and human-in-the-loop agent workflows, but they do not analyze mixed-authorship production code alongside downstream review and runtime outcomes [6, 10, 24, 40].

This gap in production-lifecycle analysis is particularly critical given the growing consensus across academia and industry regarding the trade-offs of AI-assisted development. While general surveys chart massive growth in code generation technologies [16], recent data clearly highlights accompanying shifts in code quality and maintainability [38]. Static analyses further highlight variations in security and reliability across different platforms [1, 25]. In industrial software engineering, teams are increasingly navigating a balancing act: capitalizing on high initial development velocity while managing the subsequent risk of technical debt and maintenance overhead [3, 14]. Even as frontier models continue to evolve toward parity with human developers[4] managing the structural footprint of AI-assisted code remains a central challenge. Yet, despite this broad consensus on the quality-velocity trade-off, the structural and performance characteristics of AI-generated code have not yet been evaluated at scale within a large-scale brownfield industrial codebase.

We analyze AI-generated C++ in production, spanning change structure, static issue categories, source-level efficiency measures, and downstream outcomes in a large technology company with a monorepo-based development environment, centralized review, and post-submit operational monitoring. Using authoring-time provenance, we observe AI generation across submitted changes, final submitted versions, and deployed functions. Our analysis covers 3.52 million submitted changes and

a focused C++ slice of 10.46 million lines of code with informative authoring provenance for the static-analysis study. We use these data to measure adoption at production scale, identify concentrated upstream code properties, estimate their associations with review, reliability, and compute outcomes after adjustment, and test targeted feedback on functions.

We organize the empirical analysis around four research questions. **(RQ1)** How is AI generation distributed across submitted changes in the development workflow? **(RQ2)** What upstream code properties characterize AI-generated and human-written C++ changes, including change structure, static issue categories, and source-level efficiency measures? **(RQ3)** How do the upstream properties identified in RQ2 relate to later review, reliability, and compute outcomes? **(RQ4)** Can taxonomy-informed feedback reduce the highest-priority issue categories on AI-generated C++ functions? This paper makes four contributions.

- We present a longitudinal study of AI-generated C++ in production code changes using authoring-time provenance, characterizing how AI generation is distributed across submitted changes during the study window.
- We compare AI-generated and human-written C++ changes on upstream code properties, including change structure, a taxonomy-grounded static issue profile, and source-level efficiency measures.
- We compare downstream outcomes between AI-generated and human-written C++ code and estimate how earlier structural, static, and efficiency-related measures are associated with those outcomes.
- We evaluate whether taxonomy-informed feedback reduces targeted static findings on AI-generated C++ functions.

2. Related Work

Empirical studies of LLM-generated code usually measure code before it is used in development workflows. Early Copilot and ChatGPT studies reported uneven correctness and code quality across tasks, models, and prompts [19, 46]. Studies of real completion behavior show that benchmark completions do not fully capture what developers accept in practice [15]. Fine-grained error taxonomies further show that generated code can fail through syntactic, semantic, and repair-related patterns [44]. Work on non-functional properties reports security weaknesses, runtime inefficiency, and C++ performance costs [12, 28]. These studies establish that pass rates miss important quality variation, but most evaluate standalone tasks, repository snapshots, or generated solutions with authorship fixed by study design. In production repositories, generated code can be edited by developers, mixed with human-authored code, revised during review, and observed again only through submitted code or post-submit outcomes. That lifecycle changes both the measurement unit and the interpretation of quality. A finding observed on an initial submitted line may disappear before landing, while a source-level performance pattern may become visible only after deployment. Our study addresses this gap by observing AI generation during authoring and following submitted C++ changes through static findings, review, reliability, compute outcomes, and targeted feedback, where review can alter issue burden and downstream signals can appear at a later unit than the line where the generated code was written.

Related work on developer productivity and AI-assisted workflow motivates the importance of analysis in a production setting. Controlled and enterprise studies report faster task completion, while interaction studies describe steering, inspection, and trust costs [6, 29]. Industrial work on LLM-based review and human-in-the-loop agent deployments shows that AI systems can affect review time, failure trajectories, and developer oversight [10, 40]. These process studies focus on task time,

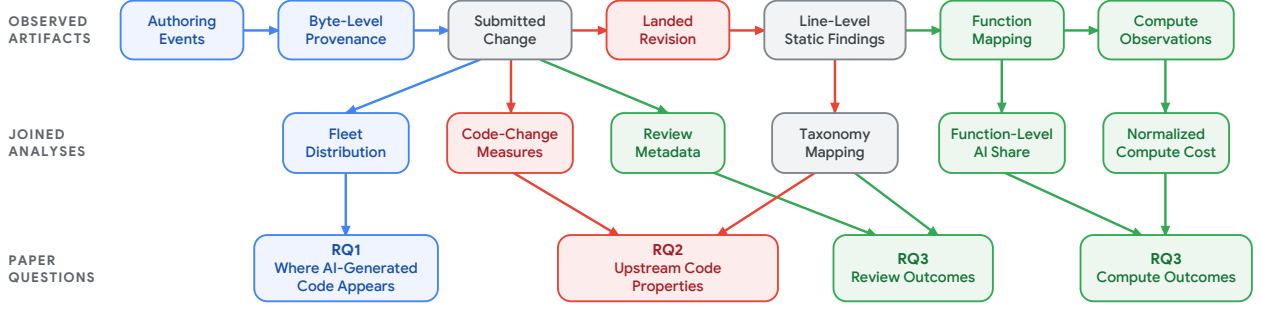


Figure 1 | Study overview. Provenance is observed during authoring, aggregated to submitted changes, projected to landed revisions, and joined to static findings together with review and operational outcome signals. Arrows indicate aggregation, projection, and cross-surface joins. RQ1 uses submitted changes fleet-wide, RQ2 characterizes upstream C++ code properties, and RQ3 links those upstream signals to review and compute outcomes.

interaction behavior, review-tool use, repository activity, or benchmark-solving traces, so they do not characterize the source-level issue profile of submitted production code. Submitted code also raises an attribution problem because final code may contain model output that developers edited, interleaved, or partly rewrote. Detectors of AI-generated source code generalize poorly across models and settings [39], and memorization studies show why origin can be hard to infer from final code alone [2]. Our setting avoids post hoc attribution by using authoring-time provenance as a measured property of the development process. The study also builds on industrial systems for static analysis, code review, and actionable feedback. Tricorder and Modern Code Review show how analysis and review operate under large-codebase workflow constraints [33, 34]. Productive Coverage and recent LLM-based review systems show that quality signals become more useful when they are prioritized by practical relevance and presented as developer-facing feedback [17, 26]. Existing work motivates our evidence standard, but it does not characterize provenance-observed AI-generated C++ after industrial review, relate taxonomy-level static findings to downstream review, reliability, and compute outcomes, or test whether high-priority categories respond to targeted feedback using authoring-time provenance instead of post-submit reconstruction.

3. Study Design

3.1. Dataset and Study Scope

The study is conducted in a large technology company with a multi-language monorepo, centralized review, and standard build and static-analysis pipelines. The analysis follows submitted code across 5 levels of observation, from organization-wide provenance to C++ change structure, line-level static findings, review and reliability outcomes, and function-level compute observations. A intervention then tests targeted feedback on categories selected from the upstream analyses. We exclude generated outputs, vendor code, and code areas that bypass the standard pipeline, consistent with prior industrial studies of large-scale review and static-analysis ecosystems [17, 33, 34]. The organization-wide analysis covers submitted changes from April 1, 2025 to April 1, 2026, using all languages represented in the provenance data during that window. The final analysis includes 3.52 million submitted changes overall, and the focal C++ sample covers 10.46 million lines of code with informative authoring provenance, excludes most tests from static-finding comparisons, and contains 350k reviewable changes covered by static analysis. Function-level compute analyses require source-range-to-function mapping and normalized post-submit compute measurements from production monitoring, producing

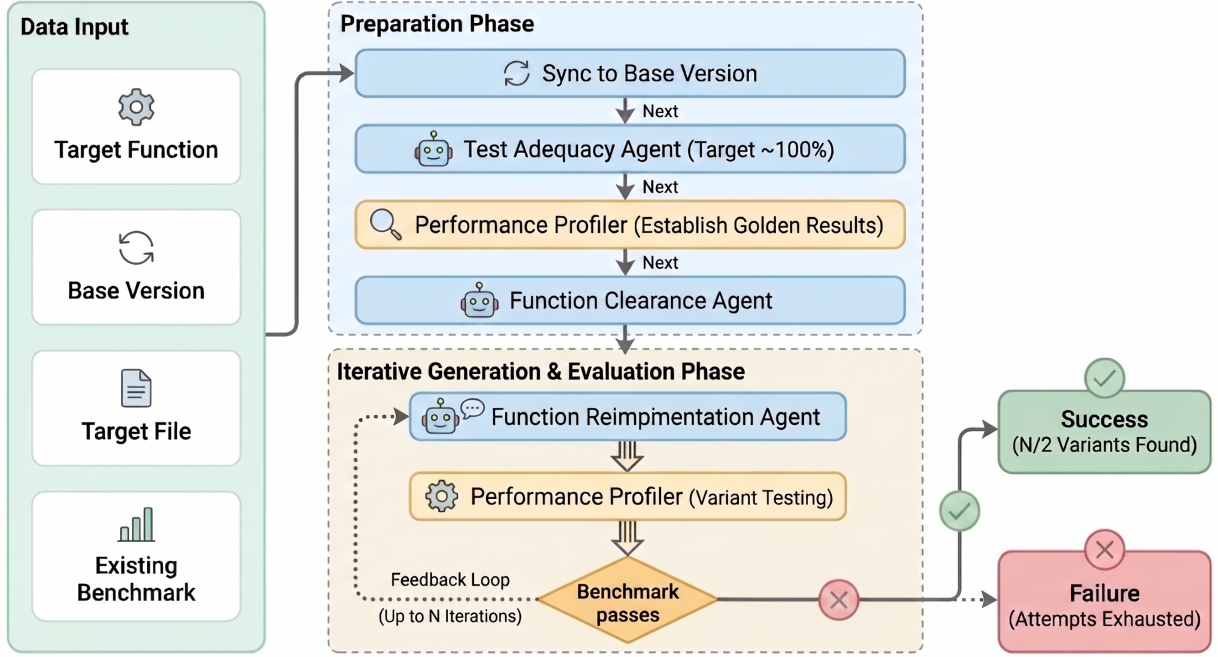


Figure 2 | Evaluation dataset generation process for the intervention analysis.

70k observations. Comparative compute analysis observes cohorts of approximately 25k AI-heavy and 12k human-typed functions, and further stratifies by edit volume, resulting in approximately 6k matched functions in each cohort.

Submitted changes are the unit for organization-wide distribution, upstream code-property, review, and reliability analyses, while functions are the unit for compute analyses. We aggregate byte-level provenance within each change to obtain AI-generation share across months, languages, interaction modes, and anonymized organizational slices. The intervention uses a synthesized benchmark of 50 C++ functions with at least one target-category finding and micro-benchmarks for validating computational behavior. We remove each original function body while preserving inputs, outputs, and intended logic, then ask the coding agent to reimplement the function under the prompt conditions in Section 4.4, as shown in Figure 2.

3.2. Provenance, Revision Mapping, and Analysis Units

Provenance and projection. We observe authorship provenance on individual changed bytes during authoring, and then project these byte-level annotations onto the lines, functions, and static findings of the final submitted snapshot used by later analyses. We aggregate consistently logged tool features into AI-generation and human-authoring categories, pooling inline completion, conversational generation, agentic editing, and transformation-based editing into 1 AI-generation signal.

Cohorts and analysis units. For a line ℓ , $AIShare(\ell)$ is the fraction of informative bytes attributed to AI-generation features. Because static analysis tooling often attaches a finding to a narrow subset of the symptomatic code, a static finding inherits $AIShare$ from the full matched line or line span, using a length-weighted mean when the finding spans multiple lines. Function-level compute analyses aggregate AI and human informative byte counts over the enclosing function and project static findings to the same function to derive category covariates.

Table 1 | Top-level definitions for the static taxonomy used in the C++ analyses.

Quality Attribute	Definition
<i>Efficiency and Resource Use</i>	Avoidable runtime or resource overhead in otherwise valid code, including wasted work, needless copying, inefficient access patterns, or expensive low-level choices.
<i>Correctness and Safety</i>	Semantic hazards, invalid state, unsafe API use, or undefined-behavior risk.
<i>Maintainability and Readability</i>	Issues that increase inspection, review, or maintenance effort by making code harder to read, reason about, or evolve.
<i>Modernity and API Evolution</i>	Outdated language usage, obsolete APIs, or missed adoption of safer and clearer modern C++ idioms.
<i>Policy, Portability, and Environment Fit</i>	Poor fit with platform, policy, or deployment constraints that can already be detected statically.

3.3. Static Taxonomy

The taxonomy classifies line-level static findings in C++ final submitted snapshots. Each mapped finding is anchored to a line or line range and propagated to the enclosing function, file, and code change. Review comments, review latency, revert behavior, sanitizer outputs, and post-submit compute cost are downstream outcomes and are not part of the taxonomy. The 3-level taxonomy groups findings into quality attributes, issue categories, and issue types.

We construct the taxonomy by collapsing raw checks and aliases across tools, then assigning each finding to a primary issue category based on the developer-facing problem it signals. The initial codebook is seeded from high-frequency checks and refined through stratified samples, following taxonomy-development guidance and empirical taxonomy practice for generated-code failures [18, 41, 44]. 2 annotators independently code a validation sample spanning frequent checks, high-AI-share checks, and every quality attribute. Before downstream use, domain experts review the final raw-check-to-category mapping and confirm that high-volume checks retain their developer-facing meaning. The validation records raw-check coverage, validation sample size, inter-rater agreement, and unmapped-bucket size.

3.4. Metrics and Models

Upstream code measures. We use added LOC, deleted LOC, add/delete ratio, files touched, analyzed C++ LOC, and new-code status as change-level descriptors and controls. The static issue profile is measured with weighted findings per analyzed KLOC, category composition, and AI/human rate ratios. Mixed-authorship findings contribute to AI and human rates in proportion to observed provenance shares. Issue-type ratios are reported only in follow-up analyses, and issue types with combined support below 1,000 are not included.

Source-level efficiency measures. We measure loop constructs, standard-library/API usage, move-related warnings, container-insertion warnings, map-access warnings, repeated-work warnings, and low-level implementation-overhead warnings on the same C++ submitted-code sample. Each measure is summarized by observed signal, AI/human contrast, denominator, and its relationship to compute modeling. We include a measure in the compute models only when it meets the predefined support threshold, remains directionally stable under normalized denominators, and has a concrete relationship to runtime cost.

Review, reliability, and compute outcomes. Review and reliability outcomes are measured at

the code-change level using categories computed on the initial submitted snapshot. The primary review outcomes are total comments, blocking comments, and time to merge, with reviewer iterations and submit attempts reported as secondary process outcomes and build failure, sanitizer findings, and revert rate reported as reliability outcomes. These review measures follow industrial studies that use comments, latency, and blocking interactions as workflow outcomes [13, 34, 42]. Rather than relying on parametric regression models, we evaluate these outcomes through stratified cohort comparisons. We report effect sizes as relative rates or percent changes, accounting for confounding factors by stratifying across month, change size, coarse organizational slice, anonymized author or team controls, and new-code status where available. Function-level compute comparisons utilize longitudinal resource tracking across two distinct function cohorts classified by AI-generation share, measuring utilization growth indexed to a baseline at the start of our study window. Our primary specification stratifies these cohorts by their respective edit volumes. To isolate relative resource consumption, the final outcomes are normalized against the growth of total application compute, denominating CPU cost and heap-resident memory strictly as a percent-of-application share.

Execution profile shifts. To quantify how upstream structural code properties alter runtime behavior over time, we conducted a longitudinal CPU shift analysis comparing production profiling data from the first and last months of the study window. Using the established cohorts of AI-heavy and human-written functions, we partitioned total compute cycles, spanning the focal functions and their direct callees, into distinct categories. Execution time was classified into imperative signals, which include direct on-CPU processing, object constructors, vector operations, and map lookups; and declarative signals, capturing cycles spent calling shared, optimized library implementations (e.g., `abs1` and `gt1`). By calculating the change in the percentage of total CPU cycles allocated to these categories, we determined the net shift in the runtime execution profile of each cohort.

Intervention outcomes. The intervention evaluates taxonomy-informed feedback on the 50-function benchmark using 3 prompt stages and 3 independent runs per stage, producing 450 generated implementations before validation exclusions and following prior work on LLM-based review support [26, 40]. The primary outcome is targeted static finding count, and the secondary outcome is R_{eff} , a benchmark-based score computed from CPU instruction count and memory usage relative to the original implementation. The score maps the weighted instruction/memory comparison to 3 values, with 1 indicating improvement, 0.5 parity, and 0 regression. Instruction count receives 2× the memory weight because execution latency is the main cost channel in the benchmark.

4. Results

4.1. RQ1: Distribution and Trends

Figure 3 shows the organization-wide distribution of AI-generated code during the study window. Provenance instrumentation becomes available from March 2025 onward, but the reported trend and cohort summaries in this section use only submitted changes from April 1, 2025 to April 1, 2026. Each submitted change in that window inherits an AI-generation share aggregated from its annotated bytes, and the figure summarizes those submitted-change shares across languages, months, anonymized organizational slices, and interaction modes. AI-generated code represents a rapidly growing share of submitted code with informative authoring provenance during this period, with its share across major languages rising from 28.99% in April 2025 to 68.62% in March 2026. At the submitted-change level for C++, the share of majority AI-generated changes grew from 27.65% to 59.69% over the same period, while majority human-written changes fell from 72.10% to 40.04%. These descriptive results establish where AI-generated code appears and why the C++ sample is large enough and important enough to analyze in depth.

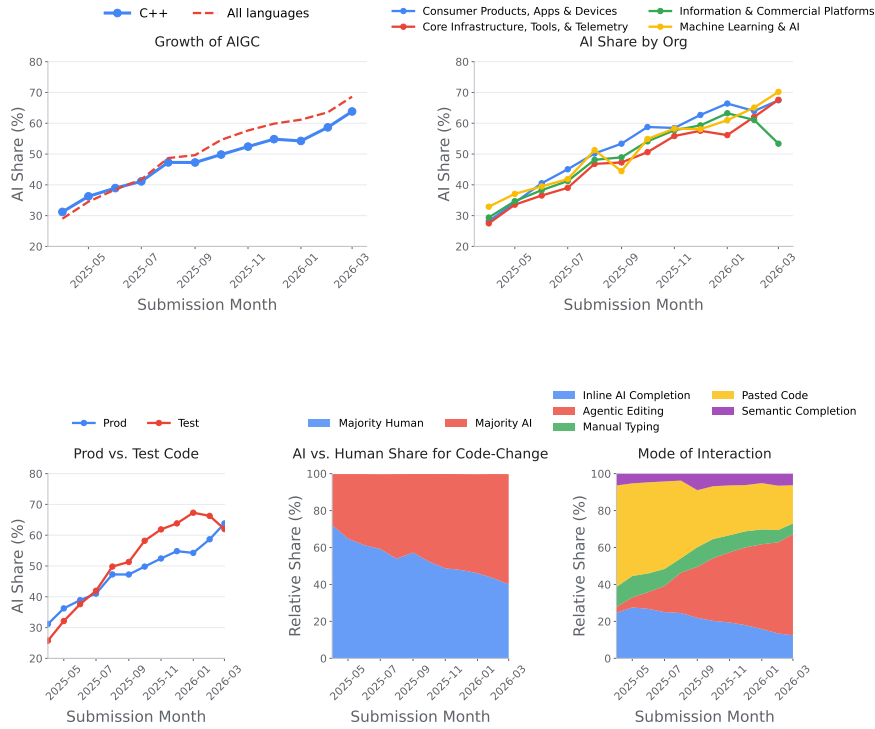


Figure 3 | Distribution of AI-generated code during the study window.

We also observe that the distribution changes over time, but not at the same rate in every organizational slice. Some coarse organizational slices show steady growth, while others plateau after initial adoption. For example, the overall C++ monthly share rises from 28.56% at the start of the window to 62.80% at the end. By the end of the window, adoption spreads steadily across the long tail of product domains, though absolute penetration varies. AI-generated code spreads beyond the earliest high-use slices, with notable concentration visible in domains like Machine Learning & AI (reaching 70.19%) and Consumer Products, Apps & Devices (reaching 67.45%). Later analyses use distributional summaries and coarse slice- and domain-level controls instead of treating the organization as homogeneous.

The interaction-mode mix also provides important context for the later results. During the study window, bytes attributed to AI generation are distributed across two primary interaction modes: Inline tooling (which includes inline completions, smart pasting, and next-edit predictions) and agentic workflows (which encompass conversational generation, transformation-based editing, automated test generation, and AI-driven refactoring). These shares are descriptive and clarify what the pooled AI-generation signal represents in this deployment setting. The pooled AI-generation signal analyzed in this study is fundamentally heterogeneous. Over the 1-year study window, the production environment deployed a shifting mix of multiple model generations, varying significantly in parameter scale, training objectives, and context-window capabilities. This includes early-window legacy architectures optimized strictly for inline latency alongside later-window conversational and agentic models. Accordingly, the results characterize the operational reality of a rolling production ecosystem rather than the standalone capability ceiling of any singular, modern state-of-the-art model.

RQ1 summary. We find that AI generation is already a large source of submitted code in this workflow, but its distribution is uneven across languages, organizational slices, and interaction modes, and developers substantially filter generated text before it reaches submitted-code analysis.

Table 2 | Contrasts between AI-generated and human-written C++ changes. Entries report median [IQR] and Cliff’s δ .

Metric	AI (Med [IQR])	Human (Med [IQR])	δ
<i>Code-change Level</i>			
Lines Changed	89 [27, 230]	33 [8, 122]	0.284
Files Touched	3 [2, 5]	2 [1, 3]	0.233
New Code Ratio	0.83 [0.51, 0.97]	0.60 [0.14, 0.93]	0.232
Desc. Length	522 [375, 742]	491 [363, 699]	0.056
<i>Function Level</i>			
LOC	11 [5, 26]	15 [7, 31]	-0.118
Complexity	3 [1, 6]	3 [1, 6]	-0.029

4.2. RQ2: Upstream Code Properties

RQ2 uses three upstream views of the same submitted-code population, covering change structure, static issue categories, and source-level efficiency measures before review, reliability, or compute outcomes are observed. Table 2 compares AI-generated and human-written C++ code changes on code-change and function-level structure, and Table 3 reports the category-level static issue profile on the same submitted-code sample. AI-generated C++ changes are larger and newer, but the static issue-rate gap is concentrated in a small number of categories instead of spread evenly across the taxonomy.

Table 2 shows why change structure needs to be modeled before interpreting issue rates. Relative to human-written changes, AI-generated changes touch more lines and more files, and have a higher new-code ratio, while description length changes much less. These measures keep size, novelty, and surface area from being mistaken for issue-category effects, allowing the later comparison to focus on which static categories and local implementation patterns characterize AI-generated C++.

At the quality-attribute level, Efficiency, Resource Use and Maintainability and Readability have above-parity AI/human rate ratios of 1.23 and 1.08, respectively. Modernity and API Evolution is close to parity at 1.04, while Correctness and Safety and Policy, Portability, and Environment Fit are lower in AI-generated code, with ratios of 0.94 and 0.92. The attribute-level view provides the context for the category results in Table 3. Even at this coarser level, the profile is not uniform: some quality attributes carry most of the excess burden, while others contribute little or move in the opposite direction.

Concentration. Among categories with support of at least 1,000, the largest positive absolute rate gaps come from Interface and Coupling Burden and Copy and Allocation Overhead. These two categories represent 82.21% of the total positive absolute rate gap. We find a narrow practical target in interface/coupling checks and copy/allocation checks, while many mapped categories stay near parity or move lower in AI-generated code. Later analyses track category rates beyond an overall finding rate because the positive gap is concentrated in these developer-facing mechanisms.

Composition shows which issue classes reviewers encounter most often. Interface and Coupling Burden represents 43.96% of the weighted findings in AI-generated code versus 40.36% in human-written code, and API Misuse and Invalid Calls represents 29.49% versus 30.05%. The same two categories make up 73.45% of weighted findings in AI-generated code and 70.41% in human-written code. We report rate contrast and composition because a category can be overrepresented in AI-generated code while forming a small share of the total burden, and a common category can dominate weighted findings in both cohorts without being the most disproportionate.

Table 3 | Static issue profile of AI-generated and human-written C++, grouped by quality attribute. Columns report AI/Human rate ratio and within-attribute share of weighted findings.

Issue Category	Rate Ratio (AI/Human)	Share (%)
<i>Efficiency and Resource Use</i>	1.23	100.0%
Copy and Allocation Overhead	<u>1.39</u>	83.4%
Data Structure and Access Inefficiency	0.80	8.9%
Low-Level Implementation Overhead	0.56	6.7%
Redundant Work	0.79	0.6%
I/O and Formatting Inefficiency	<u>3.16</u>	0.4%
<i>Modernity and API Evolution</i>	1.04	100.0%
Legacy Language Idioms	0.82	43.1%
Deprecated or Obsolete API Usage	<u>1.41</u>	41.2%
Missed Type-Safety or Ownership Evolution	0.96	15.7%
<i>Maintainability and Readability</i>	1.08	100.0%
Interface and Coupling Burden	<u>1.15</u>	75.9%
Dead, Redundant, or Unused Code	0.97	13.4%
Clarity and Explicitness	0.86	5.9%
Naming and Local Documentation Issues	0.80	4.4%
Control-Flow and Structural Complexity	0.92	0.4%
<i>Correctness and Safety</i>	0.94	100.0%
API Misuse and Invalid Calls	0.93	91.3%
Numeric and Conversion Hazards	<u>1.25</u>	3.9%
Initialization and State Validity	0.86	2.4%
Lifetime and Ownership Hazards	0.71	1.7%
Concurrency and Synchronization Hazards	<u>1.01</u>	0.7%
<i>Policy, Portability, and Environment Fit</i>	0.92	100.0%
Policy or Compliance Violations	0.91	87.2%
Portability Risks	<u>1.01</u>	12.6%
Operational Environment Mismatch	0.33	0.2%

Support changes how we read these ratios. A high ratio with sparse support marks a narrow contrast, while a high ratio paired with a large absolute rate gap is a stronger candidate for interpretation because it is both disproportionate and substantial in the total gap. Within highlighted categories, issue types under clang-tidy[20] definition such as `misc-include-cleaner` and `misc-definitions-in-headers` make up 98.95% of Interface and Coupling Burden, while `runtime-missing-move` makes up 51.60% of Copy and Allocation Overhead. These issue types translate the category profile into concrete mechanisms that reviewers and tool builders can target.

Source-level efficiency measures. RQ2 extends the static taxonomy with source-level efficiency measures observed before downstream outcomes. Table 4 reports what was measured and whether

Table 4 | Upstream source-level efficiency measures in the C++ submitted-code sample. Ratios are normalized AI/human rates.

Measure	Observed signal	Ratio
Loop usage	AST loop counts	~2.0×
Standard library use	<code>std::*</code> usages	~0.4×
Move/copy behavior	<code>missing-move</code> warnings	1.39×
Container insertion	<code>use-emplace</code> warnings	~2.0×
Map access	<code>inefficient-map</code> warnings	~2.0×
Repeated work	findings per analyzed KLOC	<1.0×

the normalized AI/human rate ratio is above or below parity. These quantities are upstream code properties measured on submitted or final code snapshots and do not use computational cost, heap, or any other production outcome. Measures that fail the support or normalization-stability threshold remain descriptive. Measures with sufficient support and a plausible runtime-cost interpretation are used in Section 3 as candidate explanatory variables. We find that AI-generated code is nearly twice as likely to author loops, uses standard-library/API calls 30–40% less often, has almost twice as many map-access warnings, and shows a higher rate of `push_back`-related warnings. The pattern points to more explicit local control flow and fewer calls into shared library or API implementations. The downstream analysis uses AST (`loops/libs`) for loop and standard-library/API usage, `missing-move` for move-semantics-related warnings, `use-emplace` for container-insertion warnings, and `inefficient-map` for map-access warnings.

RQ2 summary. AI-generated C++ changes are larger and more new-code-heavy, and their excess static burden is concentrated in two developer-facing categories. The source-level efficiency measures add a resource-use pattern in which AI-generated code relies more on explicit local work and less on standard-library/API delegation.

4.3. RQ3: Review, Reliability, and Compute Outcomes

To understand the downstream impact of AI-generated code, we tracked changes through the review process and into production, comparing reliability and efficiency metrics across our authorship cohorts.

Reliability: Build Failures and Revert Rates. Our longitudinal cohort analysis reveals a distinct split between pre-deployment friction and post-deployment stability for AI-generated code. When examining build failure rates over the 12-month study window, the AI-generated cohort exhibited consistently higher initial instability. As shown in Figure 4, the ratio of AI-generated to human-written build failures remained above parity (with a median ratio of roughly 1.3 $\times$).

However, AI-generated code demonstrated a consistently *lower* revert rate than human-written code. Across the study period, the relative ratio for revert rates remained largely below parity (median $\sim 0.9\times$). This indicates that while AI-generated code requires more automated pipeline cycles to stabilize, once it passes initial review and automated checks, it is less likely to require an immediate rollback than human-authored code.

Compute and Memory Outcomes. When tracking code into production, we observe a growing divergence in resource consumption over time (Figure 4, bottom panels). Over the study window, the median normalized compute cost for AI-heavy functions grew at a steeper trajectory than for mostly human-written functions. By early 2026, AI-generated compute growth reached approximately 1.31 $\times$ baseline, compared to roughly 1.25 $\times$ for human code (an approximate 5% relative increase). This gap is even more pronounced in memory consumption, where AI-heavy functions reached nearly 1.36 $\times$ baseline growth versus 1.25 $\times$ for human functions (an approximate 8% relative increase). This confirms that the upstream efficiency issues identified in RQ2 translate into measurable compute and memory overheads at scale.

Table 5 reports the longitudinal CPU shift from the beginning to end of the study window, demonstrating that the source-level efficiency properties identified in Table 4 translate into measurable execution overheads in production. While Table 4 established that AI-generated code statically biases toward explicit local loops ($\sim 2.0\times$) and away from standard API calls ($\sim 0.4\times$), Table 5 reveals the dynamic consequences: AI-heavy functions exhibit a 3.46% relative shift toward direct on-CPU execution and a 1.08% relative decrease in delegation to declarative libraries. Furthermore, the specific static efficiency antipatterns, such as elevated warnings for missing move semantics and suboptimal container usage, manifest in localized runtime cost driving relative compute footprint

Table 5 | 1-Year CPU Shift Analysis: Imperative vs. Declarative Constructs (AI vs. Human)

Category	AI Shift (%)	Human Shift (%)	Relative Shift
Imperative Signals			
On-CPU	+0.88%	-2.57%	+3.46%
Constructors	+1.31%	-0.05%	+1.36%
Vectors	+0.07%	-0.03%	+0.10%
Map Lookups	+0.07%	-0.01%	+0.07%
Declarative Signals			
Libraries (absl/gtl)	-2.34%	-1.27%	-1.08%

increases in constructors (+1.36%), vectors (+0.10%), and map lookups (+0.07%).

Descriptive downstream contrasts. Figure 4 summarizes the unadjusted downstream contrasts used to motivate the staged models. For review friction, AI-generated changes require significantly more human effort: they receive 1.92× as many Blocking Threads (reviewer feedback that must be resolved before submission) and 1.39× as many total Comments. We also observe elevated friction across secondary review metrics, including Reviewer Iterations (1.24×), Time to Merge (1.19×), and Submit Attempts (1.07×). For reliability and process checks, both Sanitizer findings and Build Failure Rates show higher incidences in AI-generated code (medians ~1.3×), while the Revert Rate remains below parity (median ~0.9×).

Summary of RQ3. The reliability data paints a clear picture: the burden of AI-generated code is not found in acute breakages or reverts. Its comparable build failure rate and superior revert rate suggest that existing automated checks and human reviewers successfully catch fatal errors before, or immediately during, deployment. The true cost of AI generation, therefore, shifts from acute reliability failures to chronic maintenance and operational burdens, manifesting as the interface coupling and compute overheads identified in our static and source-level analysis (RQ2).

4.4. RQ4: Intervention

RQ4 uses the RQ2 profile as an intervention target. We evaluate taxonomy-informed feedback on the 50-function benchmark described in Section 3.4, where each selected function contains at least one target-category finding from the upstream efficiency profile. Each function is regenerated under 3 prompt stages, as illustrated in Figure 5. Stage 1 uses the base reimplementation prompt, Stage 2 adds general feedback, and Stage 3 adds category-specific feedback for the targeted efficiency findings. The benchmark keeps the same function context, reimplementation budget, and validation harness across stages, so changes in findings and R_{eff} reflect the feedback condition under a fixed task setup.

Quantitative impact of taxonomy-informed refinement. To evaluate whether the concentrated upstream profile can guide feedback, we measured the normalized computational efficiency score (R_{eff}) and the volume of static efficiency findings across the 3 prompt stages. Table 6 reports the results. The baseline in the table stands for the original code snippet before the synthesis. Under the standard drafting instructions (Stage 1), the LLM matched the human baseline finding rate (1.26 versus the 1.28 baseline) with a modest R_{eff} score of 0.294.

Stage 3, which adds category-specific feedback, increased the average R_{eff} score to 0.385 while maintaining low generation variance (± 0.043) across three independent runs, a **31%** improvement over the prompt without feedback. Stage 3 also reduced the average volume of static efficiency

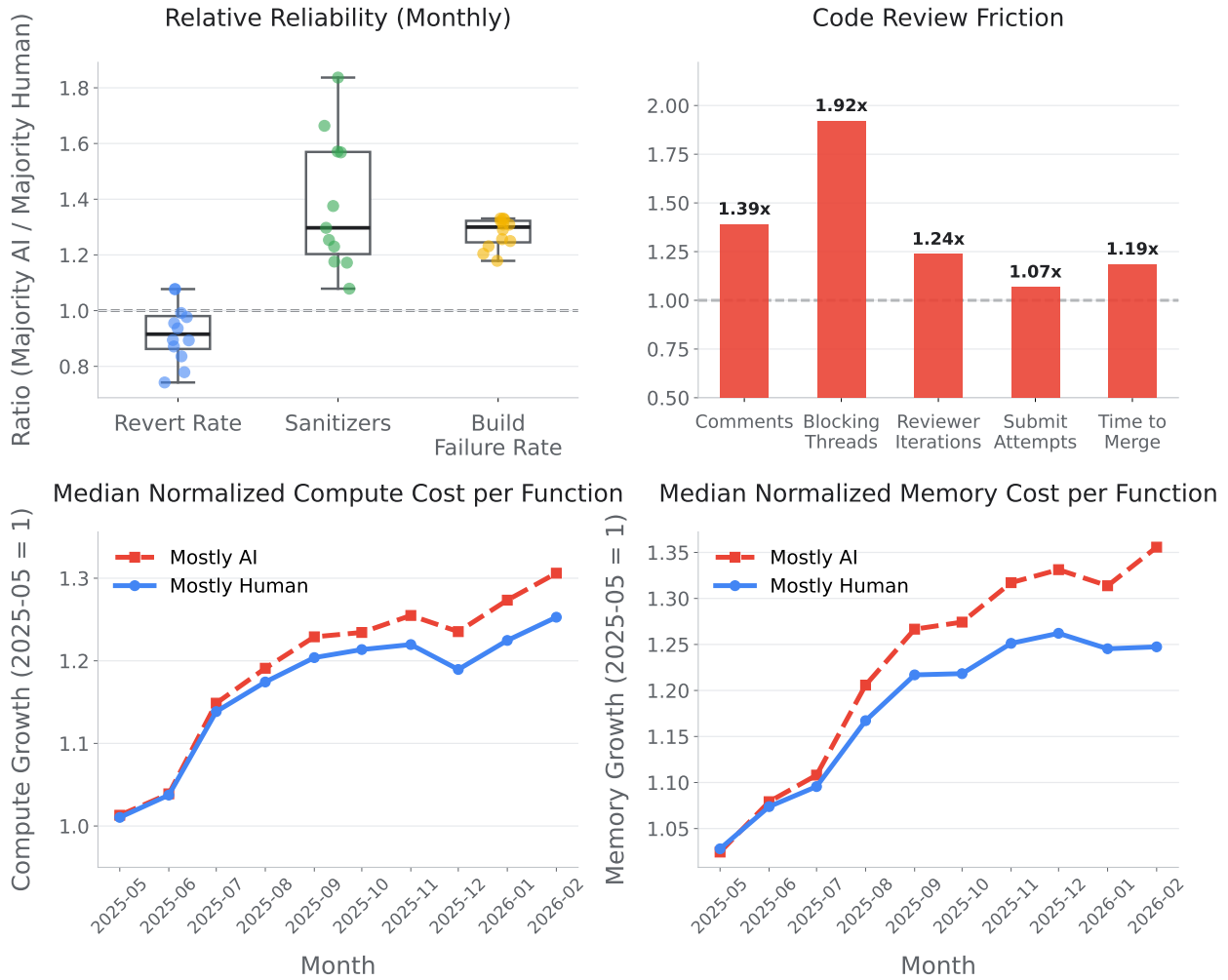


Figure 4 | Review, reliability, and compute outcomes for AI-generated and human-written C++ code. Panels show the descriptive downstream contrasts used to motivate the staged models: monthly ratios for revert rate, sanitizer findings, and build failure, review-friction ratios, and monthly normalized compute cost.

findings to 1.12, an **11.1% reduction** relative to Stage 1 and a **12.5% reduction** relative to the baseline finding rate. The same RQ2 categories used to characterize submitted code also identify feedback targets that respond in a controlled generation setting.

RQ4 summary. Taxonomy-informed feedback derived from the upstream profile reduces targeted static efficiency findings on functions while improving the benchmark efficiency score, providing an initial demonstration that the profile can guide category-specific mitigation.

5. Discussion

We interpret the empirical results as an initial baseline for code quality during the early stages of large-scale AI adoption. The scale result places the quality analysis in an active production workflow, while the C++ results show that the profile remains patterned after developer filtering and review-gated submission. Interface and Coupling Burden and Copy and Allocation Overhead represent much of the positive static gap, which provides reviewers and tool builders with a practical starting point for category-level attention.

Table 6 | Taxonomy-informed feedback results.

Stage	R_{eff} ($\uparrow$)	Findings ($\downarrow$)	Reduction ($\uparrow$)
Baseline	-	1.28	-
Stage 1	0.294 (± 0.041)	1.26	1.56%
Stage 2	0.342 (± 0.041)	1.22	4.69%
Stage 3	0.385 (± 0.043)	1.12	12.50%

Note: Results are averaged across three independent runs.

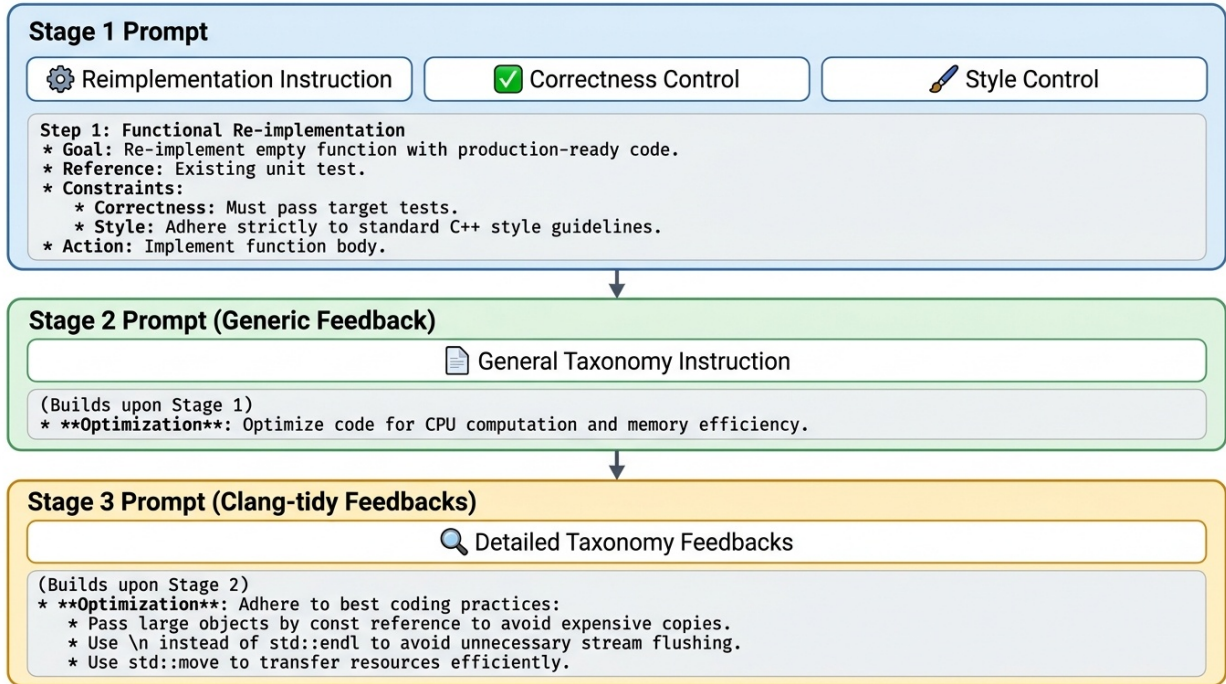


Figure 5 | Prompt stages for taxonomy-informed feedback.

The taxonomy helps interpret raw static checks as developer-facing mechanisms. In RQ2, `misc-include-cleaner` and `misc-definitions-in-headers` explain nearly all of Interface and Coupling Burden, while `runtime-missing-move` explains a large share of Copy and Allocation Overhead. Those mechanisms are concrete enough for review guidance, analyzer prioritization, and feedback design. They also reveal why target selection should consider both rate contrast and composition, since a category can be overrepresented without dominating review burden, and a common category can dominate findings in both AI-generated and human-written code.

The source-level efficiency measures add a second mechanism for interpreting the profile. The higher loop/API contrast, map-access warnings, container-insertion warnings, and missing-move findings point to code that performs more work locally within generated functions and delegates less work to shared optimized C++ implementations. That pattern helps explain why compute-facing analyses should look beyond low-level instruction throughput. IPC and MIPS show no notable degradation, while normalized compute and heap footprint suffer execution and allocation pressure after deployment.

The feedback study tests whether the profile can support intervention as well as measurement. Stage 3 names targeted efficiency categories selected from the upstream analysis, and the observed reductions show that the same category structure can guide feedback in a controlled generation setting. Practically, the result suggests a tooling path in which static analysis identifies recurring

category-level risks and feedback steers generated code away from the same local patterns.

Importantly, the intervention results from RQ4 demonstrate that the issues identified in RQ2 are not hard limits of the underlying models' capabilities. The fact that taxonomy-informed prompting successfully reduced targeted static analysis findings by 11.1% and boosted R_{eff} by 31% shows that these models are highly malleable. We argue that the observed weaknesses are largely artifacts of historical default system configurations and a lack of context-aware prompting in early implementations. For instance, the models' tendency toward an imperative loop bias vanishes when the system design provides adequate architectural guidance and targeted feedback.

For AI coding tools used in production, pass rate, completion acceptance, and benchmark runtime are insufficient on their own. Evaluation should measure whether generated code is edited away before submission, which static categories remain visible to reviewers, which source-level patterns appear in deployed functions, and whether the observed categories respond to feedback. That evidence standard is consistent with prior work on large-scale analysis ecosystems and practical prioritization of quality signals [13, 17, 33, 42].

6. Threats to Validity

Construct validity. The paper measures AI-generated code through authoring-time provenance observed during authoring and submission. Authoring-time provenance provides stronger measurement than post hoc source-origin attribution, but it does not eliminate all ambiguity [2, 35, 39, 45]. Overlapping AI and human-authoring features on the same bytes, revision mapping from changed bytes to the final submitted snapshot, and tool-specific static checks all introduce measurement choices. The static taxonomy is also intentionally local: it captures line-level issue categories, not every higher-level design or algorithmic property that might later affect compute cost. We address these risks through explicit overlap policies, overlap-excluded sensitivity analyses, frozen check-to-category mappings, support thresholds, and reporting of the unmapped bucket.

Internal validity. The review and compute analyses are observational. AI-generation share can correlate with task difficulty, repository context, author experience, or review norms. The paper uses sequentially adjusted models that first add code-change measures and then category variables, with controls for time, size, and organizational slice. These steps reduce confounding risk, but they do not convert the study into a causal estimate of AI on quality or cost. The intervention analysis offers a stronger mitigation test, yet it is still narrower than an organization-wide randomized rollout.

Model Mix and Prompt Evolution. Because individual model identifiers were masked or aggregated at the data-collection tier, this paper cannot disentangle whether the observed issues are evenly distributed across all models or heavily driven by earlier, less capable model versions. Furthermore, developer prompting proficiency naturally grew over the course of the 1-year timeline. As a result, older commits may also reflect lower-quality human steering.

Human-in-the-loop confounders and review dynamics: We also investigated whether standard review metrics—such as prolonged review time or higher iteration counts—correlated with the survival of inefficient AI-generated C++ code, hypothesizing that reviewer fatigue or over-trust might play a measurable role. However, our analysis yielded no clear correlation. The difficulty of isolating these specific architectural trade-offs during standard review suggests that conventional proxy metrics for human effort do not adequately capture the cognitive friction of evaluating AI-generated code. This null result highlights a critical reality in modern workflows: because human reviewers struggle to consistently intercept these localized inefficiencies regardless of review depth, upstream automated interventions—such as the taxonomy-informed prompt feedback evaluated in RQ4—are essential to

mitigating post-deployment compute costs.

External validity. The main static-issue, review, and compute analyses focus on C++ in one large industrial monorepo. That focus is deliberate because the language has strong performance constraints and mature static analysis, but it limits direct transfer to other languages and organizations. The deployed AI authoring tools, rollout timing, and relative mix of interaction modes are also organization-specific, so the observed organization-wide distribution and heterogeneity patterns may not transfer directly to environments with another tool ecology. The results are most directly transferable to organizations with similar review-gated monorepo workflows, authoring-time provenance, mature static analysis, and at least some function-level compute-cost observability [33, 34]. The organization-wide distribution analysis provides broader context across a multi-language production environment, but the category definitions and their associations with downstream outcomes should be interpreted as C++-specific until replicated elsewhere. The taxonomy is a production-grounded characterization of one high-value language, not a universal ontology for all AI-generated code.

Reproducibility and reporting validity. Some underlying data sources are enterprise-internal, which constrains full release. Company, repository, and tool names are anonymized for double-anonymous review, but the paper retains the contextual details needed to interpret the setting: monorepo-based development, centralized review, multi-language scope, provenance start date, and approximate C++ scale. We further mitigate reporting risk by defining the analytical units, code-change measures, overlap policies, mapping rules, denominators, and sensitivity analyses explicitly, while keeping the taxonomy independent of the internal tool names used to instantiate it. These reporting choices do not fully eliminate replication barriers, but they improve transparency and make partial replication feasible in other industrial settings.

Statistical conclusion validity. Some issue categories will have low support, and some downstream outcomes may be noisy or highly skewed. The paper addresses this through support thresholds for issue-type reporting, exposure-normalized finding-rate metrics, confidence intervals on category contrasts, staged models, and robustness checks with alternative denominators and overlap policies. Even with these safeguards, small categories, noisy outcome measures, and high-collinearity settings should be interpreted cautiously.

Data Availability

This study uses enterprise-internal provenance, review, repository, and production-monitoring data that cannot be released publicly. The anonymous submission includes a replication package for reviewers with the taxonomy definition, category codebook, analysis specifications, model formulas, and selected aggregated statistics needed to support the paper’s main claims. The package excludes raw data, person-level records, repository identifiers, service names, tool names, and other internal identifiers. The anonymous submission omits company, repository, and tool names while retaining the contextual details required to interpret the study design.

7. Conclusion and future works

This paper shows that AI-generated C++ in production has a structured quality profile that can be measured across authoring, submitted code, review, deployed execution, and feedback. Using authoring-time provenance across 3.52 million submitted changes, we find that AI-generated code is widespread but unevenly distributed, is filtered substantially before submission, and shows a C++ static issue-rate gap concentrated in Interface and Coupling Burden and Copy and Allocation Overhead. Source-level efficiency measures point to more explicit local work and lower standard-

library/API use, with move/copy and container patterns providing concrete targets for review and tooling.

Beyond direct operational costs, mitigating the compute overhead of AI-generated code is a growing sustainability imperative. As the industry scales AI-assisted authoring, adopting these taxonomy-driven feedback loops will be essential to curbing the broader carbon footprint of enterprise software ecosystems.

A critical driver of the observed compute overhead is the propensity of AI models to generate explicit, imperative logic rather than leveraging optimized, shared libraries. Our source-level analysis demonstrates that AI-generated C++ relies roughly twice as much on local loop constructs and utilizes standard library or API calls 30% to 40% less often than human-written code. This behavior is likely symptomatic of generalist LLMs lacking the highly specific context of internal enterprise monorepo structures. Future work should explore the integration of a knowledge base for the coding agents that dynamically inject relevant contexts into the prompt, directly addressing this imperative bias.

The intervention study shows that this profile can guide category-specific feedback on functions, reducing targeted static findings while improving the benchmark efficiency score. Production evaluation of AI-generated code should combine provenance, submitted-change structure, developer-facing issue categories, downstream outcomes, and targeted feedback studies. While our direct injection of taxonomy findings into the re-implementation prompt is effective (RQ4), we plan to extend this feedback in two directions. One is to use a recursive prompt optimization process with specific feedback loops, similar to [7] with modern prompt tuning like [31]. Besides, we plan to incorporate these empirical profiles into a Reinforcement Learning from Execution Feedback (RLEF) pipeline. By building larger synthetic benchmark sets with production-level compute measurements and static analysis findings counts as reward signals, we hope to turn passive, prompt-time interventions into continuous, proactive alignment of the AI system with enterprise performance standards.

This study deliberately focuses on C++ due to its rigorous performance constraints and mature static analysis tooling. However, a critical question for future research is whether these upstream code properties generalize to other strictly typed or performance-critical languages. Further investigation is required to determine if the same imperative biases and library-avoidance patterns manifest in languages like Rust, Go, or Java, or if different language ecosystems produce entirely distinct AI-generated issue taxonomies.

References

- [1] H. Al-Fawareh and H. M. Al-Shdaifat. Investigating ai-generated code on the impact on software efficiency code quality factor. In *Sustainable Data Management: Navigating Big Data, Communication Technology, and Business Digital Leadership. Volume 1*, pages 89–97. Springer, 2025.
- [2] A. Al-Kaswan, M. Izadi, and A. van Deursen. Traces of memorisation in large language models for code. In *Proceedings of the 46th IEEE/ACM International Conference on Software Engineering, ICSE '24*, pages 1–12. ACM, 2024. doi: 10.1145/3597503.3639133. URL <https://doi.org/10.1145/3597503.3639133>.
- [3] E. Anderson, G. Parker, and B. Tan. *The hidden costs of coding with generative AI*. MIT Sloan Management Review, 2025.

- [4] Anthropic. When ai builds itself, June 2026. URL <https://www.anthropic.com/institute/recursive-self-improvement>.
- [5] R. Apsan, V. Stoico, M. Albonico, R. Dhar, K. Vaidhyathan, and I. Malavolta. Generating energy-efficient code via large-language models – where are we now? *CoRR*, abs/2509.10099, 2025. doi: 10.48550/arXiv.2509.10099. URL <https://arxiv.org/abs/2509.10099>. Accepted at ICSE 2026 Research Track.
- [6] S. Barke, M. B. James, and N. Polikarpova. Grounded copilot: How programmers interact with code-generating models. *Proceedings of the ACM on Programming Languages*, 7(OOPSLA1): 85–111, 2023. doi: 10.1145/3586030.
- [7] S. Blyth, S. A. Licorish, C. Treude, and M. Wagner. Static analysis as a feedback loop: Enhancing llm-generated code beyond correctness. In *2025 IEEE International Conference on Source Code Analysis & Manipulation (SCAM)*, pages 100–109, 2025. doi: 10.1109/SCAM67354.2025.00017.
- [8] A. Brown, S. D’Angelo, A. Murillo, C. Jaspan, and C. Green. Identifying the factors that influence trust in AI code completion. In *Proceedings of the 1st ACM International Conference on AI-Powered Software, AIware ’24*, pages 1–9, 2024. doi: 10.1145/3664646.3664757. URL <https://doi.org/10.1145/3664646.3664757>.
- [9] J. L. Butler, J. Suh, S. Haniyur, and C. Hadley. Dear diary: A randomized controlled trial of generative AI coding tools in the workplace. In *Proceedings of the IEEE/ACM International Conference on Software Engineering: Software Engineering in Practice, ICSE-SEIP 2025*, pages 319–329, 2025. doi: 10.1109/ICSE-SEIP66354.2025.00034.
- [10] U. Cihan, V. Haratian, A. İçöz, M. K. Gül, Ö. Devran, E. F. Bayendur, B. M. Uçar, and E. Tüzün. Automated code review in practice. In *2025 IEEE/ACM 47th International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*, pages 425–436, 2025. doi: 10.1109/ICSE-SEIP66354.2025.00043. URL <https://doi.org/10.1109/ICSE-SEIP66354.2025.00043>.
- [11] M. Coutinho, L. Marques, A. Santos, M. Dahia, C. França, and R. de Souza Santos. The role of generative AI in software development productivity: A pilot case study. In *Proceedings of the 1st ACM International Conference on AI-Powered Software, AIware ’24*, pages 131–138, 2024. doi: 10.1145/3664646.3664773. URL <https://doi.org/10.1145/3664646.3664773>.
- [12] D. Erhabor, S. Udayashankar, M. Nagappan, and S. Al-Kiswany. Measuring the runtime performance of C++ code written by humans using GitHub Copilot. In *2025 IEEE/ACM 47th International Conference on Software Engineering (ICSE)*, pages 2062–2074, 2025. doi: 10.1109/ICSE55347.2025.00059. URL <https://doi.org/10.1109/ICSE55347.2025.00059>.
- [13] A. Frömmgen, J. Austin, P. Choy, N. Ghelani, L. Kharatyan, G. Surita, E. Khapko, P. Lamblin, P. Manzagol, M. Revaj, M. Tabachnyk, D. Tarlow, K. Vilella, D. Zheng, S. Chandra, and P. Maniatis. Resolving code review comments with machine learning. In *Proceedings of the 46th International Conference on Software Engineering: Software Engineering in Practice*, pages 204–215, 2024. doi: 10.1145/3639477.3639746. URL <https://doi.org/10.1145/3639477.3639746>.
- [14] H. He, C. Miller, S. Agarwal, C. Kästner, and B. Vasilescu. Speed at the cost of quality: How cursor ai increases short-term velocity and long-term complexity in open-source projects. *arXiv preprint arXiv:2511.04427*, 2025.

- [15] V. J. Hellendoorn, S. Proksch, H. C. Gall, and A. Bacchelli. When code completion fails: A case study on real-world completions. In *Proceedings of the 41st IEEE/ACM International Conference on Software Engineering*, ICSE '19, pages 960–970, 2019. doi: 10.1109/ICSE.2019.00101.
- [16] N. Huynh and B. Lin. Large language models for code generation: A comprehensive survey of challenges, techniques, evaluation, and applications. *arXiv preprint arXiv:2503.01245*, 2025.
- [17] M. Ivanković, G. Petrović, Y. Kulizhskaya, M. Lewko, L. Kalinovčić, R. Just, and G. Fraser. Productive coverage: Improving the actionability of code coverage. In *Proceedings of the International Conference on Software Engineering: Software Engineering in Practice*, ICSE-SEIP 2024, pages 58–68, 2024. doi: 10.1145/3639477.3639733.
- [18] M. Izadi, J. Katzy, T. van Dam, M. Otten, R. M. Popescu, and A. van Deursen. Language models for code completion: A practical evaluation. In *Proceedings of the 46th IEEE/ACM International Conference on Software Engineering*, ICSE '24, pages 1–13, 2024. doi: 10.1145/3597503.3639138. URL <https://doi.org/10.1145/3597503.3639138>.
- [19] M. T. Jamil, S. Abid, and S. Shamail. Can LLMs generate higher quality code than humans? an empirical study. In *Proceedings of the 22nd International Conference on Mining Software Repositories*, MSR 2025, pages 478–489, 2025. doi: 10.1109/MSR66628.2025.00081.
- [20] C. Lattner and V. Adve. Llvm: A compilation framework for lifelong program analysis & transformation. In *International symposium on code generation and optimization, 2004. CGO 2004.*, pages 75–86. IEEE, 2004.
- [21] S. Li, Y. Cheng, J. Chen, J. Xuan, S. He, and W. Shang. Assessing the performance of AI-generated code: A case study on GitHub copilot. In *2024 IEEE 35th International Symposium on Software Reliability Engineering (ISSRE)*, pages 216–227, 2024. doi: 10.1109/ISSRE62328.2024.00030. URL <https://doi.org/10.1109/ISSRE62328.2024.00030>.
- [22] H. Lin, M. Maas, M. Roquemoire, A. Hasanzadeh, F. Lewis, Y. Simonson, T.-W. Yang, A. Yazdanbakhsh, D. Altinbüken, F. Papa, M. N. Edmonds, A. Patil, D. Schwarz, S. Chandra, C. Kennelly, M. Hashemi, and P. Ranganathan. Eco: An llm-driven efficient code optimizer for warehouse scale computers, 2025. URL <https://arxiv.org/abs/2503.15669>.
- [23] Y. Liu, T. Le-Cong, R. Widyasari, C. Tantithamthavorn, L. Li, X. D. Le, and D. Lo. Refining ChatGPT-generated code: Characterizing and mitigating code quality issues. *ACM Transactions on Software Engineering and Methodology*, 33(5):1–26, 2024. doi: 10.1145/3643674. URL <https://doi.org/10.1145/3643674>.
- [24] H. Mozannar, G. Bansal, A. Fourney, and E. Horvitz. Reading between the lines: Modeling user behavior and costs in AI-assisted programming. In *Proceedings of the CHI Conference on Human Factors in Computing Systems*, pages 1–16, 2024. doi: 10.1145/3613904.3641936. URL <https://doi.org/10.1145/3613904.3641936>.
- [25] M. A. F. Muñoz, J. C. J. De La Torre, S. P. López, S. Herrera, and C. A. C. Uribe. Comparative study of ai code generation tools: quality assessment and performance analysis. *LatIA*, (2):21, 2024.
- [26] S. K. Ngassom, A. M. Dakhel, F. Tambon, and F. Khomh. Chain of targeted verification questions to improve the reliability of code generated by LLMs. In *Proceedings of the 1st ACM International Conference on AI-Powered Software*, AIware '24, pages 122–130, 2024. doi: 10.1145/3664646.3664772.

- [27] E. Paradis, K. Grey, Q. Madison, D. Nam, A. Macvean, V. Meimand, N. Zhang, B. Ferrari-Church, and S. Chandra. How much does AI impact development speed? an enterprise-based randomized controlled trial. In *2025 IEEE/ACM 47th International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*, pages 618–629, 2025. doi: 10.1109/ICSE-SEIP66354.2025.00060. URL <https://ieeexplore.ieee.org/document/11121676/>.
- [28] H. Pearce, B. Ahmad, B. Tan, B. Dolan-Gavitt, and R. Karri. Asleep at the keyboard? assessing the security of GitHub copilot’s code contributions. In *2022 IEEE Symposium on Security and Privacy (SP)*, pages 754–768, 2022. doi: 10.1109/SP46214.2022.9833571.
- [29] S. Peng, E. Kalliamvakou, P. Cihon, and M. Demirer. The impact of AI on developer productivity: Evidence from GitHub copilot. *CoRR*, abs/2302.06590, 2023. doi: 10.48550/arXiv.2302.06590.
- [30] G. V. Pereira, V. Jackson, R. Prikladnicki, A. van der Hoek, L. Fortes, C. Araújo, A. Coelho, L. Chelli, and D. Ramos. Exploring GenAI in software development: Insights from a case study in a large brazilian company. In *Proceedings of the IEEE/ACM International Conference on Software Engineering: Software Engineering in Practice, ICSE-SEIP 2025*, pages 330–341, 2025. doi: 10.1109/ICSE-SEIP66354.2025.00035.
- [31] R. Pryzant, D. Iter, J. Li, Y. Lee, C. Zhu, and M. Zeng. Automatic prompt optimization with “gradient descent” and beam search. In *Proceedings of the 2023 conference on empirical methods in natural language processing*, pages 7957–7968, 2023.
- [32] P. K. Rajput, Y. Song, A. A. Bonkougou, I. E. Olatunji, A. K. Kaboré, J. Klein, and T. F. Bissyandé. Correctness isn’t efficiency: Runtime memory divergence in LLM-generated code. *CoRR*, abs/2601.01215, 2026. doi: 10.48550/arXiv.2601.01215. URL <https://arxiv.org/abs/2601.01215>.
- [33] C. Sadowski, J. van Gogh, C. Jaspan, E. Söderberg, and C. Winter. Tricorder: Building a program analysis ecosystem. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, pages 598–608, 2015. doi: 10.1109/ICSE.2015.76.
- [34] C. Sadowski, E. Söderberg, L. Church, M. Sipko, and A. Bacchelli. Modern code review: A case study at google. In *Proceedings of the 40th International Conference on Software Engineering: Software Engineering in Practice, ICSE-SEIP 2018*, pages 181–190, 2018. doi: 10.1145/3183519.3183525.
- [35] F. Salerno, A. Al-Kaswan, and M. Izadi. How much do code language models remember? an investigation on data extraction attacks before and after fine-tuning. In *2025 IEEE/ACM 22nd International Conference on Mining Software Repositories (MSR)*, pages 465–477, 2025. doi: 10.1109/MSR66628.2025.00080. URL <https://ieeexplore.ieee.org/document/11025762/>.
- [36] G. Sandoval, H. Pearce, T. Nys, R. Karri, S. Garg, and B. Dolan-Gavitt. Lost at C: A user study on the security implications of large language model code assistants. In *32nd USENIX Security Symposium (USENIX Security 23)*, pages 2205–2222, 2023.
- [37] M. L. Siddiq, S. H. Majumder, M. R. Mim, S. Jajodia, and J. C. S. Santos. An empirical study of code smells in transformer-based code generation techniques. In *2022 IEEE 22nd International Working Conference on Source Code Analysis and Manipulation (SCAM)*, pages 71–82, 2022. doi: 10.1109/SCAM55253.2022.00014.
- [38] Sonar. State of code developer survey report. Technical report, SonarSource, 2026. URL <https://www.sonarsource.com/state-of-code-developer-survey-report.pdf>.

- [39] H. Suh, M. Tafreshipour, J. Li, A. Bhattiprolu, and I. Ahmed. An empirical study on automatically detecting AI-generated source code: How far are we? In *Proceedings of the 47th IEEE/ACM International Conference on Software Engineering*, ICSE '25, pages 859–871, 2025. doi: 10.1109/ICSE55347.2025.00064.
- [40] W. Takerngsaksiri, J. Pasuksmit, P. Thongtanunam, C. Tantithamthavorn, R. Zhang, F. Jiang, J. Li, E. Cook, K. Chen, and M. Wu. Human-in-the-loop software development agents. In *Proceedings of the IEEE/ACM International Conference on Software Engineering: Software Engineering in Practice*, ICSE-SEIP 2025, pages 342–352, 2025. doi: 10.1109/ICSE-SEIP66354.2025.00036.
- [41] M. Usman, R. Britto, J. Börstler, and E. Mendes. Taxonomies in software engineering: A systematic mapping study and a revised taxonomy development method. *Information and Software Technology*, 85:43–59, 2017. doi: 10.1016/j.infsof.2017.01.006.
- [42] M. Vijayvergiya, M. Salawa, I. Budiselic, D. Zheng, P. Lamblin, M. Ivankovic, J. Carin, M. Lewko, J. Andonov, G. Petrovic, D. Tarlow, P. Maniatis, and R. Just. Ai-assisted assessment of coding practices in modern code review. In *Proceedings of the 1st ACM International Conference on AI-Powered Software*, AIware '24, pages 85–93, 2024. doi: 10.1145/3664646.3665664. URL <https://doi.org/10.1145/3664646.3665664>.
- [43] C. Wang, K. Huang, J. Zhang, Y. Feng, L. Zhang, Y. Liu, and X. Peng. LLMs meet library evolution: Evaluating deprecated API usage in LLM-based code completion. In *Proceedings of the 47th IEEE/ACM International Conference on Software Engineering*, ICSE '25, pages 885–897, 2025. doi: 10.1109/ICSE55347.2025.00245.
- [44] Z. Wang, Z. Zhou, D. Song, Y. Huang, S. Chen, L. Ma, and T. Zhang. Towards understanding the characteristics of code generation errors made by large language models. In *Proceedings of the 47th IEEE/ACM International Conference on Software Engineering*, ICSE '25, pages 2587–2599, 2025. doi: 10.1109/ICSE55347.2025.00180.
- [45] W. Wu, H. Hu, Z. Fan, Y. Qiao, Y. Huang, Y. Li, Z. Zheng, and M. R. Lyu. An empirical study of code clones from commercial AI code generators. *Proceedings of the ACM on Software Engineering*, 2(FSE):2874–2896, 2025. doi: 10.1145/3729397. URL <https://doi.org/10.1145/3729397>.
- [46] B. Yetistiren, I. Ozsoy, and E. Tuzun. Assessing the quality of GitHub copilot’s code generation. In *Proceedings of the 18th International Conference on Predictive Models and Data Analytics in Software Engineering*, PROMISE '22, pages 62–71, 2022. doi: 10.1145/3558489.3559072.