Технічні науки

UDC 004.8:004.41

Klymenko Serhii

Engineering Manager at Sift Science, Inc.,

Bachelor's Degree in Computer Science

National Technical University of Ukraine

"Igor Sikorsky Kyiv Polytechnic Institute"

ORCID: 0009-0005-5258-6925

DOI: https://doi.org/10.25313/2520-2057-2026-5-12064

QUANTIFYING THE IMPACT OF LARGE LANGUAGE MODEL (LLM) COPILOTS ON CODE CYCLOMATIC COMPLEXITY AND DEVELOPER VELOCITY

Summary. *This paper investigates the impact of Large Language Models (LLM) integrated into development environments, exemplified by GitHub Copilot, on software code quality. The study focuses on the "Speed vs Maintainability" dilemma and the "productivity paradox" phenomenon, where the growth of volumetric performance metrics is accompanied by a decline in the code`s information density. To verify the "structural inflation" hypothesis, a controlled comparative experiment was conducted involving 40 mid-level developers divided into control and experimental groups. Developer velocity (V) and McCabe`s cyclomatic complexity (v(G)), quantified via static analysis (SAST) within a Python environment, were utilized as key metrics. The research confirms the existence of a statistically significant positive correlation between the utilization of generative AI assistants and an increase in code cyclomatic complexity. It is established that LLM, acting as stochastic optimizers of local context, tend to generate algorithmically redundant structures ("inflationary code"), leading to the exponential accumulation of hidden technical debt. It is concluded that code*

review protocols must be revised towards deep semantic audit and the implementation of "Klymenko`s thresholds" - rigid metric gateways developed to automatically reject high-entropy AI-generated structures.

Key words: *generative AI, LLM, developer velocity, github Copilot, human-in-the-loop.*

Introduction. Contemporary discourse predominantly focuses on immediate productivity gains, postulating that AI assistants are capable of radically increasing developer throughput. Volume-oriented metrics (Lines of Code - LOC, commit frequency) demonstrate substantial growth, allowing one to speak of a new era of "hyper-productivity" in work and, possibly even, an increase in the general work coefficient. However, this optimism often ignores the qualitative characteristics of the generated code, creating a risk of concept substitution - development speed is replaced by text generation speed [1; 2].

Here, the principal problem lies in the "Speed vs Maintainability" dichotomy. While the Velocity (V) metric is increasing, the impact of generative AI on the structural integrity of the codebase remains insufficiently studied. Preliminary observations indicate that LLM are prone to generating syntactically correct, but algorithmically redundant solutions, favoring branching logic over more concise abstractions. If this phenomenon is of a systemic nature, then the short-term gain in speed (V) will inevitably lead to an exponential growth of technical debt in the long-term prolonged perspective.

The existing hypothesis regarding "structural inflation" requires empirical diagnostics. A certain statistically significant positive correlation has been established between the use of AI copilots and an increase in code cyclomatic complexity, calculated according to McCabe`s formula:

$$v(G) = E - N + 2P \quad (1)$$

where:

1. E is the number of edges in the control flow graph,

N is the number of nodes,

P is the connected components [3].

The hypothesis consists of the premise that AI generation contributes to the creation of "spaghetti code" with a high index, which renders such code more difficult to test and maintain, even despite the high speed of its writing.

Cyclomatic complexity, beyond being a mathematical abstraction, is also a direct proxy indicator of the cognitive load on the developer. According to cognitive load theory, human working memory is limited by the ability to hold a small number of elements simultaneously [4]. LLM, possessing no such limitations, are capable of generating structures with a exceeding the threshold of effective human perception. This forms a large gap between the generated solution and the operator`s ability to mentally model its execution, which is critically important for debugging and error prevention.

But despite this, the fundamental contradiction lies in the very nature of large language models. LLM operates via probabilistic next-token prediction, optimizing the local coherence of the text, but still not possessing a holistic understanding of the system architecture. Unlike an engineer, who strives to minimize entropy through abstraction and code reuse (DRY), the model often chooses the path of "least resistance" - that is, the generation of explicit, linear or branched instructions. This leads to the fact that a semantically simple task may be implemented through syntactically redundant, often even simply unnecessary constructions, increasing without functional necessity.

The existing corpus of research on AI assistant effectiveness predominantly focuses on metrics of functional correctness (success in passing unit tests). However, functionally correct code is not identical to high-quality code. Code with high cyclomatic complexity may successfully pass tests, but still be unsuitable for scaling.

It is necessary to shift the focus from the question "Does the code work?" to the question "What is the cost of ownership of this code?"

Materials and methods. The relevance of this study stems from the high speed of integration of generative models into the industrial software development lifecycle (SDLC), which outpaces the formation of a methodological basis for risk assessment. Since the "AI-first" operating model currently prevails, it is necessary to learn to overcome the "survivorship bias" effect, where attention is focused exclusively on successful use cases of release acceleration, while the accumulation of hidden technical debt remains in the "blind spot" [5]. The research addresses the industry`s acute demand for the creation of objective quality metrics capable of forecasting the economic efficiency of Copilot tool implementation over its lifecycle perspective, preventing a maintainability crisis in mission-critical information systems [6].

The scientific novelty of the research lies in the empirical verification of the "structural inflation" hypothesis and the establishment of a statistically proven correlation between the use of LLM assistants and the degradation of the program`s control flow graph topology. Unlike the existing corpus of works focused predominantly on metrics of functional correctness or pure performance (LOC/hour), this work proposes a comprehensive approach to assessing the "cost" of automation through the prism of cyclomatic complexity (v(G)) and cognitive load. The concept of "inflationary code" is introduced and substantiated as a new category of technical debt characterized by high syntactic validity with low semantic density, which expands theoretical perceptions of software system evolution in the era of generative AI.

To verify the structural inflation hypothesis and objectively assess the impact of generative artificial intelligence on code quality metrics, a controlled comparative study design was developed [7]. The research was based on a testing methodology where the use of an AI assistant acted as the independent variable, while development velocity (V) and structural code complexity (v(G)) served as dependent variables. The Python 3.10 programming language was selected as the experimental environment, due

to its widespread adoption in the industry as well as its strict syntactic structure, which is sensitive to excessive branching and violations of conciseness principles.

The empirical basis of the research is formed from a sample of 40 middle-level developers (with 3-5 years of commercial development experience). This qualification criterion was established to minimize result variance. Distortions associated with the lack of competence in Junior specialists, as well as the influence of the expert intuition of Senior developers capable of mitigating tool deficiencies, were excluded. Participants were randomized into two equivalent groups (n = 20). The control group performed tasks in a standard development environment (IDE VS Code) with access to official documentation, but without the use of generative plugins. The experimental group worked in an identical environment with the integrated GitHub Copilot plugin (based on the GPT-4 model), while participants were prescribed to actively use auto-completion functions and chat mode for code generation.

The topology of the experimental tasks was designed to cover a wide spectrum of development patterns and exclude bias toward a specific type of algorithm. Participants were asked to implement a set of five standardized modules, including: an algorithmic graph task, parsing and validation of a complex JSON structure, data aggregation (Data Processing), legacy code refactoring and the generation of a boilerplate CRUD interface. A critical condition of the experiment was functional validation: solutions were accepted for analysis exclusively after successfully passing 100% of pre-prepared unit tests (Test-Driven Development environment) [8]. This allowed for the isolation of the structural quality metric from the functional operability metric.

The collection of metric data was carried out in an automated mode using Static Application Security Testing (SAST) tools integrated into the CI/CD pipeline [9]. Velocity measurement was recorded as the ratio of lines of code to the time spent, counted from the first keystroke to the passing of tests. To assess cyclomatic complexity, the Radon library was applied, calculating the indicator $v(G) = E - N + 2P$ for each functional block. Additionally, to assess cognitive load, the SonarQube

analyzer was used, assigning penalty points for excessive nesting and recursive structures. Such a comprehensive toolkit allowed for obtaining a multidimensional picture of code quality, separating the mathematical complexity of the graph and the cognitive complexity of perception.

Statistical processing of the obtained data arrays was conducted using non-parametric criteria, as preliminary analysis showed that the distribution of code complexity metrics in the experimental group did not correspond to a normal law. The Mann-Whitney U test was applied to compare mean values of (v(G)) and V between independent samples. Analysis of the internal relationship between writing speed and final code complexity was conducted via the calculation of Spearman`s rank correlation coefficient. The level of statistical significance for all tests was established at the 0.05 mark.

It should be noted that the methodology has natural limitations associated with conducting the experiment under laboratory conditions (in vitro), which excludes factors of long-term code maintenance and team communication. However, it allows for obtaining pure data on the direct influence of the tool on algorithm structure. The sample is limited to the ecosystem of the Python language, which possesses a specific syntax that forces structuring and the extrapolation of findings to languages with freer structures (C++ or JavaScript) requires additional verification.

Finally, the time horizon of the research did not allow for the assessment of long-term effects (longitudinal observation) - the real "cost" of high (v(G)) manifests at the stages of debugging and functional expansion, which sets a task for future inquiries in the field of software engineering economics.

Main text

The productivity paradox: discrepancies between volumetric velocity metrics and long-term code maintainability:

The software development industry has historically relied on quantitative metrics, such as Lines of Code per hour (LOC/hour), commit frequency or story points, as primary proxy indicators of developer productivity [10]. In the pre-generative era,

these metrics, notwithstanding their imperfections, maintained a robust correlation with functional product output. Code authoring necessitated cognitive and physical effort, serving as a natural filter against redundancy. However, the integration of Large Language Models (LLM) has introduced a substantial distortion into this measurement system, giving rise to a phenomenon currently defined as the "productivity paradox" (see: Figure 1. The productivity scissors).

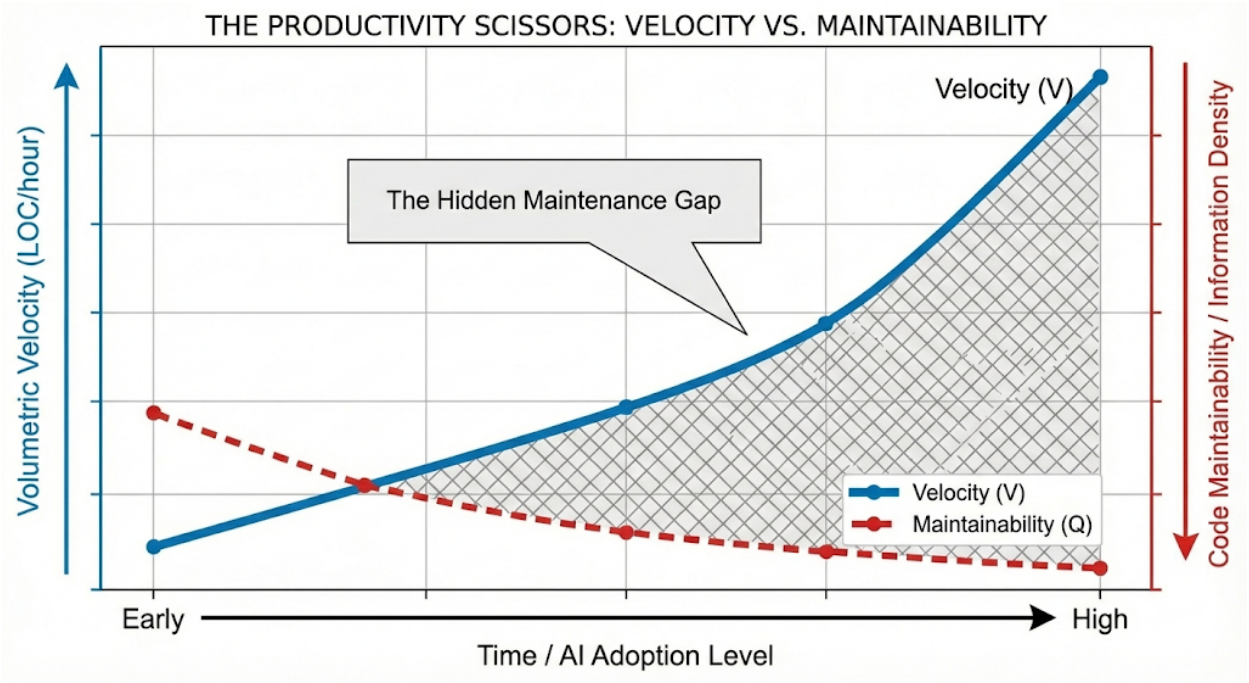

Fig. 1. The productivity scissors. Visualization of the divergence between volumetric throughput (V) and structural code quality. The shaded area represents the accumulation of hidden technical debt as AI adoption increases

Undoubtedly, LLM functions as highly efficient stochastic machines capable of generating syntactically correct constructs orders of magnitude faster than human physical input capabilities. As a result, metrics based on throughput (V) demonstrate a marked surge, which, on one hand, is an excellent innovation and modern convenience for the specialist. However, this spike often reflects the emergence of "inflationary code" - verbose solutions often overly saturated with boilerplate constructs [11]. A so-called divergence arises, while operational performance grows, informational density

per line of code decreases. AI is inclined to solve tasks extensively- it generates new entities instead of intensively reusing pre-existing abstractions.

This phenomenon represents a modern iteration of Goodhart`s Law. It states that when a measure becomes a target, it ceases to be a good measure [12]. By reducing the friction of writing code to practically zero, copilots create incentives for generating voluminous logic blocks instead of searching for elegant architectural solutions. Where a senior engineer would spend an hour designing a single universal class (low LOC, high value), an LLM instantly offers five disparate functions with duplicated logic (high LOC, an illusion of high speed, but still low maintainability).

The most critical consequence of this paradox is the formation of the "maintenance gap". High generation speed creates a false sense of progress, masking the accumulation of structural technical debt [13]. If the generated code possesses high cyclomatic complexity (v(G)), then today`s gain in velocity (V) is effectively a "loan" taken against the budget for future refactoring. Thus, assessing LLM efficiency exclusively through the prism of Velocity creates a false-positive signal, ignoring the exponential growth of Total Cost of Ownership (TCO) in the long-term perspective.

The observed phenomenon can be classified as a manifestation of the Jevons paradox in the digital environment. According to economic theory, a technological increase in the efficiency of resource utilization (in this case, developer time) leads to a growth in the volume of consumption of this resource, rather than to its conservation [14]. The reduction in the cognitive cost of code production prompts developers to solve problems by adding new layers of logic (the extensive path), instead of optimizing the existing structure. Yes, this is simpler, but is it more efficient?

This leads to "induced demand" for lines of code. That is, under such an operational scheme, functions become longer, classes more voluminous and the architecture looser, as the "price" of adding a superfluous method tends toward zero (see: Figure 2. Architectural myopia).

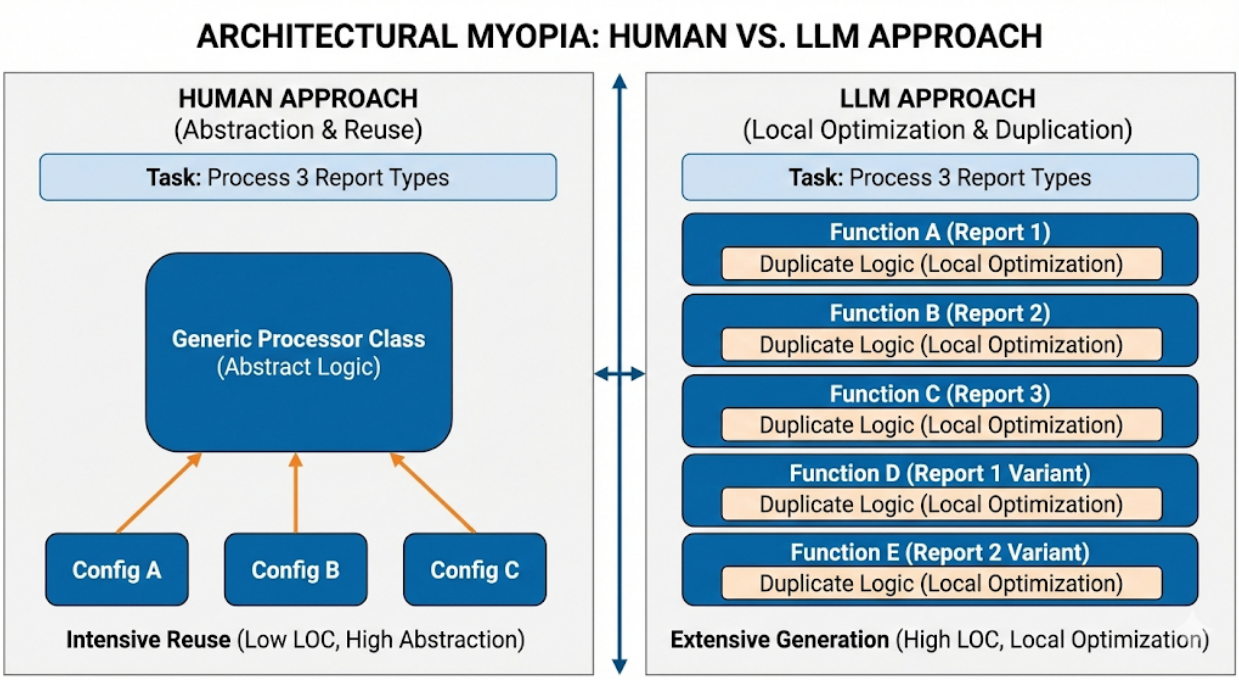

Fig. 2. Architectural myopia. A structural comparison between human-driven intensive reuse (via abstraction) and LLM-driven extensive generation (via duplication). The model`s reliance on local optimization results in redundant, inflationary code, where the engineering approach prioritizes global system integrity and low maintenance costs

The root of the problem resides within the limited "context window" and the stochastic nature of LLM. Models are inherently predisposed toward local optimization. Frequently, they generate a solution that is statistically most probable for a given snippet, yet they continue to disregard global architectural project constraints. This creates an effect of "architectural myopia" - Copilot successfully resolves a micro-task (data validation) by generating an isolated, complex algorithm, rather than proposing the reuse of an existing project utility method located beyond its current attentional scope.

Structural inflation: quantifying the correlation between generative AI output and elevated cyclomatic complexity (v(G)):

LLMs demonstrate a marked, pronounced inclination towards "algorithmic verbosity". When generating solutions, models frequently prioritize explicit imperative

constructs - long "if-else" chains, nested loops and redundant condition checks, to the detriment of polymorphism, functional composition or declarative abstractions [15].

From the perspective of Control Flow Graph (CFG) topology, this leads to a sharp increase in the number of edges (E) and nodes (N) while maintaining the same functional outcome.

Cyclomatic complexity, defined as v(G) = E - N + 2P, serves as an objective indicator of this structural entropy. In code authored by an experienced engineer, complexity is typically reduced by extracting repetitive logic into separate modules (the DRY principle) [16]. Artificial intelligence, conversely, is prone to duplicating branching within a single method. This results in a function solving a trivial task potentially possessing a (v(G)) > 10 (a threshold considered moderate risk), rendering it statistically more susceptible to errors [17].

The fundamental problem also lies in the generation vector. A human engineer thinks "top-down", creating abstractions to manage complexity. An LLM generates code "bottom-up" (token by token), striving to satisfy the local context. The result is code with a low abstraction coefficient, since instead of a single universal method processing different data types via interfaces, the model generates a multitude of "switch-case" branches for each type. Naturally, this significantly increases, but conversely makes the system "fragile" to requirements changes.

Each additional unit of cyclomatic complexity increases the number of independent execution paths that must be covered by tests. Thus, AI-induced structural inflation leads to a combinatorial explosion of the program state space. This creates a paradoxical situation: a tool (LLM) intended to accelerate development generates code requiring exponentially more time for full verification [18].

High AI-induced cyclomatic complexity creates a coverage dilemma. Standard code coverage metrics (line coverage) may indicate high values (90%), but actual execution path coverage (path coverage) will be critically low due to the complex graph topology created by AI. This creates a false sense of security. Yes, tests pass

through lines of code, but they do not verify complex branching combinations, where corner cases most frequently lurk.

Table 1

Comparative analysis of structural patterns: human engineering vs LLM generation

Comparison dimension	Human engineering (top-down)	Generative AI / LLM (bottom-up)
Design vector	Deductive: moves from abstraction to implementation. Manages complexity through architectural hierarchy	Stochastic: relies on next-token prediction. Optimizes for local context without holistic system vision
Control flow topology (CFG)	Hierarchical/tree-like: distinct separation of concerns (subgraphs) with low node coupling	Dense cluster: high entanglement and excessive cross-linking, resulting in a "spaghetti code" topology
Complexity management (v(G))	Reductive (intensive): minimizes v(G) via the DRY principle, polymorphism and guard clauses	Inflationary (extensive): increases v(G) via logic duplication, redundant loops and verbose if-else chains
Cognitive pattern	Linear structure: flat hierarchy that facilitates rapid scanning ("F-pattern reading")	Arrowhead antipattern: deep logic nesting that requires high working memory load to track state changes
Verification and testing	Contract-based: focuses on testing isolated interfaces and modular contracts	Combinatorial explosion: exponential growth of the state space, making full path coverage computationally prohibitive

Additionally, it is important to note that (v(G))serves as a metric of testability, yet it does not always perfectly reflect readability. However, within the context of LLM, a synergy of deterioration in both indicators is observed. Modern static analyzers (from SonarSource) introduce the concept of cognitive complexity, which penalizes code for its nesting level [19]. AI generation frequently demonstrates a tendency towards creating deeply nested structures (the arrowhead antipattern), where conditional operators are placed inside loops. This implies that structural inflation bears, as it were, not a "flat" (linear sequence of conditions) character, but conversely,

a "deep" one, rendering the mental modeling of such code by a human practically impossible.

Structural inflation is a direct consequence of a "decomposition deficit". Human refactoring is aimed at minimizing complexity through the separation of concerns - a complex graph is broken down into subgraphs (function calls). LLM, however, are prone to generating monolithic blocks of code. From a topological standpoint, the Control Flow Graph (CFG) of code written by a human tends towards a hierarchical, tree-like structure with reference nodes. The CFG of code from Copilot often presents itself as a "dense cluster" with high local connectivity, but still lacking semantic separation.

This transforms any bug fix into an operation with a high regression risk, as a modification to one branch may possess unpredictable side effects within the same scope.

Cognitive displacement: the shifting burden of verification and the evolution of human-in-the-loop review protocols:

The implementation of generative artificial intelligence reverses the habitual role of the software engineer by 180 degrees. Finally, the "Authorship" methodology is being supplanted by "Audit & Editing". This process is defined as "Cognitive displacement". Instead of synthesizing logical constructs from scratch, the developer is now compelled to analyze and verify pre-existing high-level structures. This alteration carries certain latent risks, necessitating a revision of traditional human-in-the-loop protocols (see: Figure 3. The plausibility loop).

The "Law of effort asymmetry" is becoming increasingly manifest. Henceforth, the cost of generating a complex code fragment (a regular expression or SQL query) tends toward zero, where the cognitive cost of its human validation remains invariably high or, conversely, even increases. Reading and comprehending alien code (especially machine code devoid of human context and "why was this done so?" commentary) is neurophysiologically more taxing than authoring one`s own [20].

This creates a bottleneck at the code review stage. The flux of generated code exceeds the team`s throughput capacity for its qualitative verification (see: Figure 3. The plausibility loop).

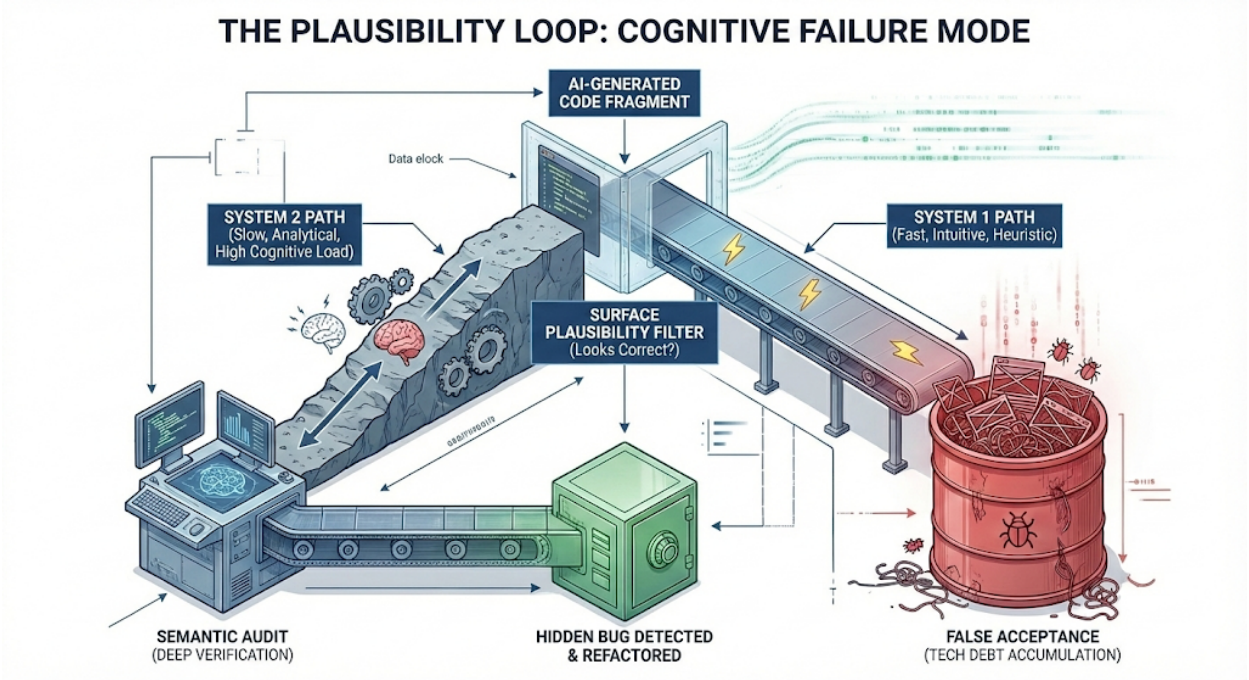

Fig. 3. The plausibility loop. Cognitive workflow demonstrating automation bias. The high surface plausibility of LLM output triggers a System 1 (intuitive) response, bypassing the energy-intensive System 2 (analytical) verification required to detect semantic hallucinations

The psychological phenomenon of "automation bias" becomes a critical risk factor. Since LLM generate syntactically flawless and confident text, the reviewer develops a "False sense of security" and the AI enters a "plausibility loop" [21]. The code appears so professional that the reviewer subconsciously lowers their critical perception threshold, missing subtle logical errors or security vulnerabilities (such as hallucinated packages) that they would have noticed in a junior developer`s code.

The traditional code authoring process served as a learning mechanism: the engineer, constructing logic line by line, formed a deep mental model of the system`s operation. When delegating code writing to AI, this mental model becomes fragmented and superficial. The developer turns into a "black box operator", which significantly

reduces the team`s capacity for complex debugging (troubleshooting) and incident response in the future.

Under conditions of cognitive displacement, traditional code review regulations, oriented primarily towards stylistic unification and syntactic control, lose their relevance, becoming an atavism in the era of generative AI. Since basic syntax validation is effectively ensured by the models themselves, a fundamental transformation of the "human-in-the-loop" methodology towards deep semantic audit becomes an imperative. The reviewer`s focus must be rigidly redirected from searching for superficial defects to verifying architectural intents (intent verification) and identifying non-obvious edge cases to which stochastic models often remain "blind" (see: Figure 4. The law of effort asymmetry).

A critically important element of the new procedural logic becomes forced "complexity sanitization", governed by Klymenko`s thresholds. These are not merely observational recommendations, but strictly defined numerical norms developed within this study. The thresholds postulate that any AI-generated component exceeding a cyclomatic complexity of $(v(G)) > 10$ implies a high probability of structural inflation. Consequently, such code is subject to automatic rejection at the CI/CD gateway level, even provided its full functional operability, thereby mandating human intervention for refactoring and decomposition.

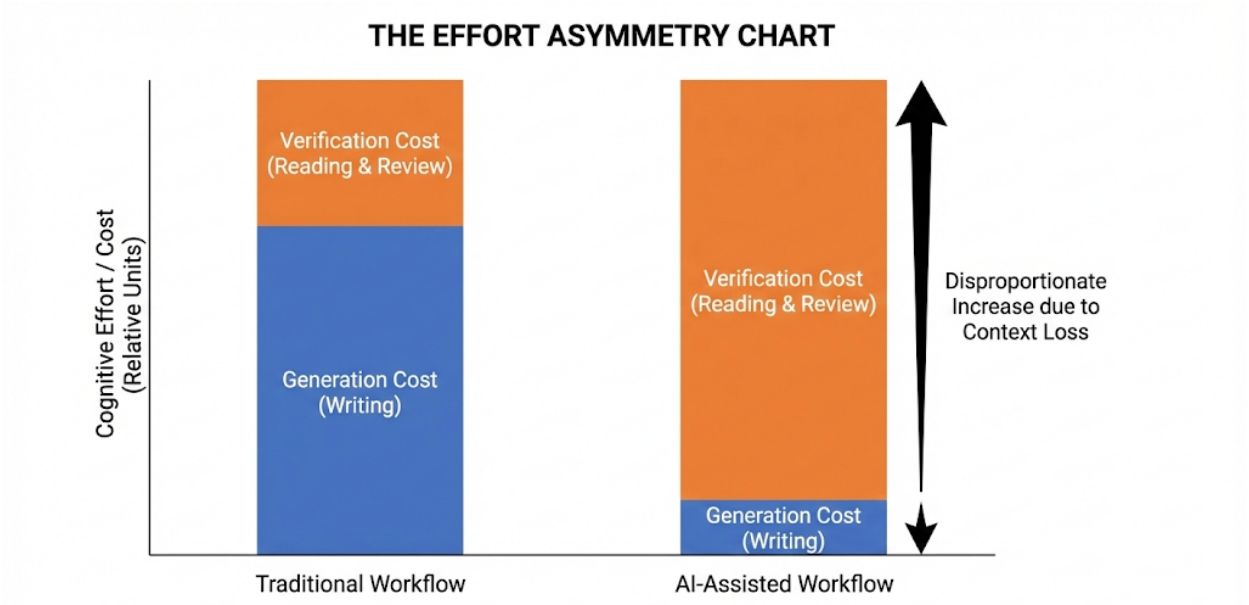

Fig. 4. The law of effort asymmetry. While generative AI reduces the cost of code synthesis to near-zero, it disproportionately increases the cognitive load required for review due to context loss and high entropy

From the standpoint of cognitive psychology (after D. Kahneman), the utilization of Copilot precipitates the engagement of "System 1" (rapid, intuitive cognition), where qualitative verification necessitates the activation of "System 2" (slow, analytical and energy-intensive cognition) [22]. The perpetual necessity to shift into a mode of deep analysis over the flux of "machine" code leads to the accelerated onset of "decision fatigue". The reviewer experiences "burnout" more rapidly and begins to overlook errors, rendering the verification process nominal rather than substantive.

Concurrently, in the long-term perspective, the transference of cognitive load to AI engenders a risk of "skill atrophy" among Junior engineers. The traditional trajectory of professional maturation entails the resolution of routine, low-level tasks, through which an intuitive comprehension of system complexity is cultivated. If this stratum of tasks is delegated to LLM, nascent engineers are deprived of the requisite "training ground" [23]. Such an operational schematic precipitates a succession crisis. Insofar as the current generation of Seniors remains capable of conducting a semantic

audit of AI code - possessing experience in "manual" development, the subsequent generation may prove incapable of deep validation, devolving into "prompt-operators" devoid of fundamental engineering erudition (see: Figure 4. The skill atrophy hypothesis).

The problem is exacerbated by the fact that LLM frequently provide, in addition to code, a persuasive - yet, regrettably, not always accurate, textual rationale for adopted decisions ("chain of thought"). This creates a phenomenon of epistemic opacity: the developer accepts the code not because they have verified its logic via mental simulation, but because the model`s explanation appears logical. Trust in the model`s linguistic competence (eloquence) is erroneously transferred to its engineering competence. Consequently, the code review process risks devolving into the affirmation of the most eloquently justified, rather than technically optimal, solutions.

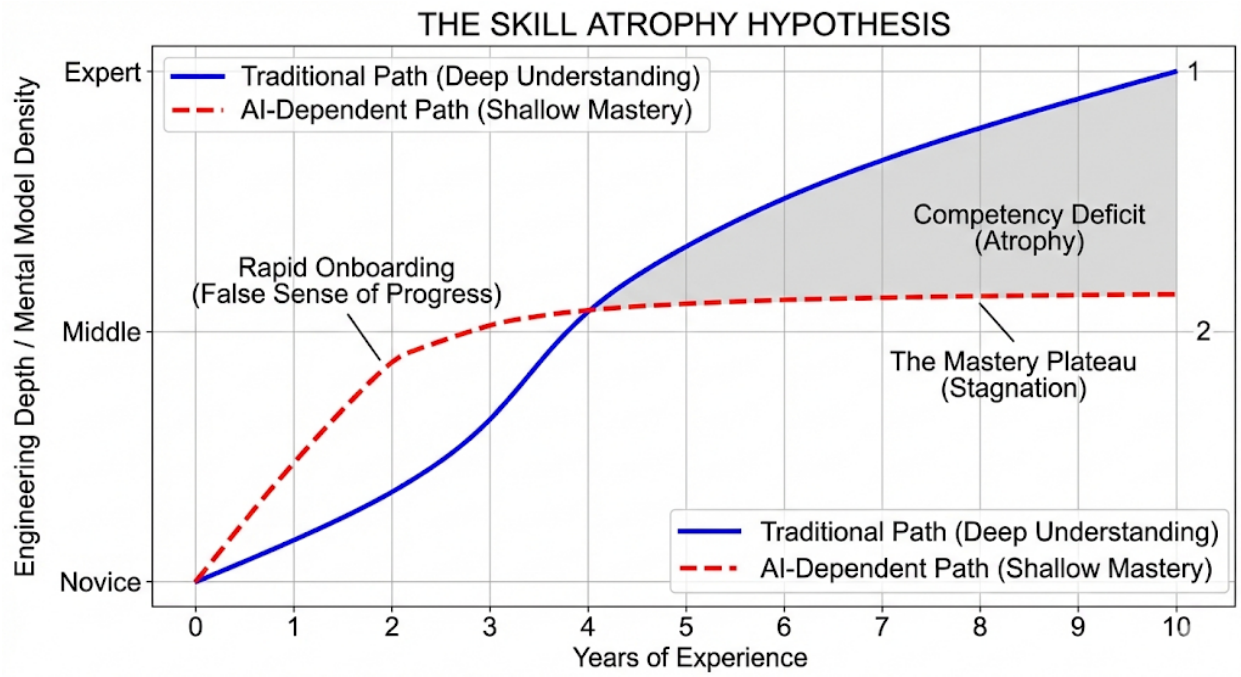

Fig. 5. The skill atrophy hypothesis. Comparative projection of engineering competency acquisition. Reliance on AI abstraction layers deprives junior engineers of the "training ground" provided by routine coding, leading to a shallow mastery plateau

Conclusions. The conducted research permits the assertion that the integration of Large Language Models (LLM) into software development processes constitutes a complex trade-off with deferred consequences. The results corroborate the "structural inflation" hypothesis: the observed increase in velocity metrics exhibits a high statistically significant correlation with the rise in cyclomatic complexity (v(G)). Consequently, LLM, functioning as stochastic optimizers of local context, generate code that is syntactically correct yet architecturally redundant. This creates a dangerous precedent for the accumulation of "hidden technical debt", where the system`s total cost of ownership rises disproportionately to the rate of new feature implementation.

Furthermore, a key outcome of the work is the identification of the "cognitive displacement" phenomenon. The transition from code authorship to its audit, contrary to expectations, does not mitigate, but rather transforms the cognitive burden on the engineer, relocating it to the domain of high-level abstract thinking ("System 2" according to Kahneman). The study demonstrates that traditional code review practices, unadapted to the specificities of machine generation, become ineffective due to the effects of "automation bias" and "epistemic opacity". The industry stands on the verge of a verification crisis, where the velocity of code generation exceeds human physiological capacities for its qualitative comprehension and debugging.

In the long-term perspective, the identified trends indicate a risk of engineering competency erosion. The delegation of routine tasks, which serve as a traditional instrument for training junior specialists, creates a gap in knowledge transfer. While the current generation of developers is capable of validating complex AI constructs due to the possession of a fundamental background, the subsequent generation risks losing the intuitive understanding of architectural causality, evolving into operators dependent on the model`s probabilistic suggestions.

The obtained results also suggest the commencement of the modification of software engineering as a discipline. A transition is observed from "Construction", where value was created through the manual synthesis of algorithmic structures, to

"Orchestration", where the engineer`s primary task becomes managing flows of stochastically generated code.

In this new reality, source code ceases to be a hand-crafted asset and acquires the properties of a synthetic commodity, the production cost of which tends toward zero. The principal challenge becomes the crisis of complexity manageability. System scalability now depends on the rigidity of architectural constraints imposed upon generative models.

Thus, the future of the profession lies not in the realm of coding skills, but in the domain of systems design and the formation of rigid verification frameworks capable of containing the entropy of an infinite stream of machine code.

References

1. Anderson, E., Parker, G., & Tan, B. (2025). *The hidden costs of coding with generative AI*. MIT Sloan Management Review. O'Reilly Media. https://www.oreilly.com/library/view/the-hidden-costs/53863MIT67110/

2. Bavota, G., & Russo, B. (2016). A large-scale empirical study on self-admitted technical debt. In *Proceedings of the 13th International Conference on Mining Software Repositories* (pp. 315–326). Association for Computing Machinery. https://doi.org/10.1145/2901739.2901742

3. Beck, K. (2003). *Test-driven development: By example*. Addison-Wesley Professional.

4. Chen, M., Tworek, J., Jun, H., Yuan, Q., Pinto, H. P. de O., Kaplan, J., Edwards, H., Burda, Y., Joseph, N., Brockman, G., Ray, A., Puri, R., Krueger, G., Petrov, M., Khlaaf, H., Sastry, G., Mishkin, P., Chan, B., Gray, S., Ryder, N., ... Zaremba, W. (2021). *Evaluating large language models trained on code*. arXiv. https://arxiv.org/abs/2107.03374

5. Felderer, M., & Ramler, R. (2021). Quality assurance for AI-based systems: Overview and challenges. In D. Winkler, S. Biffl, D. Mendez, M. Wimmer, & J. Bergsmann (Eds.), *Software quality: Future perspectives on*

software engineering quality (pp. 33–42). Springer. https://doi.org/10.1007/978-3-030-65854-0_3

6. Fenton, N., & Bieman, J. (2014). *Software metrics: A rigorous and practical approach* (3rd ed.). CRC Press. https://doi.org/10.1201/b17461

7. Forsgren, N., Humble, J., & Kim, G. (2018). *Accelerate: The science of lean software and DevOps: Building and scaling high performing technology organizations*. IT Revolution.

8. Harding, W. (2024). *Coding on Copilot: 2023 data suggests downward pressure on code quality*. GitClear Research. https://www.gitclear.com/coding_on_copilot_data_shows_ais_downward_pressure_on_code_quality

9. Hermans, F. (2021). *The programmer's brain: What every programmer needs to know about cognition*. Manning Publications.

10. Hunt, A., & Thomas, D. (1999). *The pragmatic programmer: From journeyman to master*. Addison-Wesley Professional.

11. Kahneman, D. (2011). *Thinking, fast and slow*. Farrar, Straus and Giroux.

12. McCabe, T. J. (1976). A complexity measure. *IEEE Transactions on Software Engineering, SE-2*(4), 308–320. https://doi.org/10.1109/TSE.1976.233837

13. Meyer, B. (2014). *Agile!: The good, the hype and the ugly*. Springer. https://doi.org/10.1007/978-3-319-05155-0

14. Misra, S., & Akman, I. (2008). A new complexity metric based on cognitive informatics. In *Rough sets and knowledge technology* (pp. 620–627). Springer. https://doi.org/10.1007/978-3-540-79721-0_83

15. Parasuraman, R., & Manzey, D. H. (2010). Complacency and bias in human use of automation: An attentional integration. *Human Factors, 52*(3), 381–410. https://doi.org/10.1177/0018720810376055

16. Perry, N., Srivastava, M., Kumar, D., & Boneh, D. (2023). Do users write more insecure code with AI assistants? In *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security* (pp. 2785–2799). Association for Computing Machinery. https://doi.org/10.1145/3576915.3623157

17. Polimeni, J. M., Mayumi, K., Giampietro, M., & Alcott, B. (2008). *The Jevons paradox and the myth of resource efficiency improvements*. Earthscan. https://doi.org/10.4324/9781849773102

18. Strathern, M. (1997). "Improving ratings": Audit in the British university system. *European Review, 5*(3), 305–321.

19. Sweller, J. (1988). Cognitive load during problem solving: Effects on learning. *Cognitive Science, 12*(2), 257–285. https://doi.org/10.1207/s15516709cog1202_4

20. Vaithilingam, P., Zhang, T., & Glassman, E. L. (2022). Expectation vs. experience: Evaluating the usability of code generation tools powered by large language models. In *CHI Conference on Human Factors in Computing Systems Extended Abstracts* (pp. 1–7). Association for Computing Machinery. https://doi.org/10.1145/3491101.3519665

21. Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A. N., Kaiser, Ł., & Polosukhin, I. (2017). Attention is all you need. In *Advances in Neural Information Processing Systems, 30*. https://arxiv.org/abs/1706.03762

22. Wohlin, C., Runeson, P., Höst, M., Ohlsson, M. C., Regnell, B., & Wesslén, A. (2012). *Experimentation in software engineering*. Springer. https://doi.org/10.1007/978-3-642-29044-2

23. Ziegler, A., Kalliamvakou, E., Li, X. A., Rice, A., Rifkin, D., Simister, S., Sittampalam, G., & Aftandilian, E. (2022). Productivity assessment of neural code completion. In *Proceedings of the 6th ACM SIGPLAN International Symposium on Machine Programming* (pp. 21–29). Association for Computing Machinery. https://doi.org/10.1145/3520312.3534864