UDC 004.8: 004. 42

THE HIDDEN COST OF ARTIFICIAL INTELLIGENCE IN PROGRAMMING

Shibaev I.S.

Belarusian State University of Informatics and Radioelectronics, Minsk, Republic of Belarus

Likhtarovitch I. I. – Senior Lecturer at the Department of Foreign Languages

Annotation. This article examines a structural contradiction in AI-assisted programming. While large language models accelerate code generation, their improvement is constrained by the exhaustion of high-quality human-written training data. As synthetic data increasingly enters training pipelines, recursive learning from model-generated content may create feedback loops leading to model collapse, reduced code diversity, and hallucinations in software development.

Keywords: large language models, programming, synthetic data, model collapse, feedback loop, hallucinations, code generation.

***Introduction*.** AI code assistants accelerate drafting, refactoring, and test writing. Yet their own development now depends on a resource that is becoming scarce: high-quality human-written text and code. The paradox is straightforward. The more programming is delegated to large language models, the more the public code ecosystem becomes filled with synthetic output, and the less stable the knowledge base for the next generation of models becomes. Research on data scaling suggests that the stock of public human-generated text is finite and may become a serious bottleneck for further LLM growth [1]. When that limit is approached, training pipelines inevitably lean harder on synthetic data, creating a recursive feedback loop in which models learn from their own approximations instead of from original human practice [2, 3].

Main part. The shortage should not be understood as the literal disappearance of human-written text. Repositories, forums, documentation, issue trackers, and technical blogs still exist. The problem is narrower and more serious: the supply of open, legally usable, deduplicated, and genuinely informative data is finite, while the demand of frontier models is enormous. Villalobos et al. estimate that public human-generated text may become a binding constraint for scaling between 2026 and 2032 [1]. In programming, this matters even more because not every snippet has equal value. A copied template adds little. A bug report explaining why a design failed, an edge-case discussion, or a niche library example contains rare signal that is hard to reproduce artificially.

Synthetic data appears to solve this problem. It is cheap, fast, filterable, and easy to adapt to target tasks. Models can generate code samples, explanations, tests, and debugging dialogues at an industrial scale. However, the large-scale reuse of such synthetic content gradually changes the statistical distribution of programming patterns available in public repositories. But synthetic data only imitates the statistical surface of programming discourse. It does not reproduce the real conditions under which robust code is written: failure, revision, disagreement, practical constraints, and tacit expertise. This difference becomes crucial once synthetic material is fed back into the training set.

Shumailov et al. describe model collapse as a process in which the tails of the original data distribution disappear when generative models are repeatedly trained on generated data [2]. In programming, those tails include unusual but valid API uses, minority languages, domain-specific idioms, defensive coding practices learned from outages, and nonstandard optimizations. They are rare, but they often distinguish production-quality engineering from a neat classroom answer. When recursive training suppresses these low-frequency patterns, the model becomes smoother and more uniform, yet also narrower.

This is the “dog chasing its tail” effect. A model produces plausible code; that code is published, indexed, and later recrawled; the next model then learns from it as if it were an independent human-generated contribution. The loop rewards fluency and repetition. Frequency begins to masquerade as correctness. A later study by a group including Stanford researchers shows that if original human data are preserved and synthetic data are accumulated instead of substituted, collapse can be mitigated [3].

The key point, however, remains unchanged. Synthetic data may support training, but it cannot safely replace the human-written core on which diversity and robustness depend.

For programmers, the most visible result of this process is hallucination in code generation. In software development, hallucinations are not only false statements in prose. They are invented APIs, nonexistent methods, wrong parameter names, misleading explanations of framework behavior, or code that looks coherent while failing at runtime. Recent work on code hallucinations shows that such failures are systematic enough to require dedicated benchmarks and detection methods [4]. The problem is amplified by the presentation style of LLMs: clean formatting, confident comments, and plausible naming make uncertainty look like competence.

Recursive dependence on synthetic data intensifies this risk. The model sees more polished but weakly verified code and fewer examples rooted in real debugging and production constraints. At the same time, reduced diversity pushes it toward the most probable and frequently repeated solutions. The result is not only more hallucinations, but also code homogenization. Generated programs converge on the same architectural clichés, the same shallow Annotationions, and the same safe-looking averages. As a result, developers may increasingly encounter similar solution patterns even in situations that require more specialised or context-dependent approaches. That may save time in the short term, but it narrows the design space from which programmers learn and innovate.

In practical terms, the hidden cost of AI in programming is not limited to isolated coding mistakes. It also changes how future engineers learn to think about software. If developers rely too heavily on assistants that consistently propose the most probable answer, they encounter fewer unconventional but justified solutions and fewer opportunities to understand why one design choice may be preferable to another in a specific context. Over time, this can weaken independent debugging habits, reduce attention to edge cases, and make engineering culture more dependent on machine-generated conventions. Moreover, when similar generated patterns dominate tutorials, repositories, and classroom exercises, the apparent consensus they create may conceal the fact that such solutions have never been tested under diverse real-world conditions. For universities and software teams, this means that the use of large language models should be balanced by practices that preserve genuine human expertise: code review, analysis of failed implementations, manual testing of critical paths, and careful study of documentation. Without such effort, the apparent productivity gain may conceal a gradual erosion of the intellectual skills on which reliable, diverse, and adaptable software depends. Maintaining a balance between automated assistance and independent engineering judgement therefore becomes an important challenge for both software education and professional practice.

Conclusion. The promise of AI in programming is significant, but it also has a hidden cost. Once high-quality human-written data becomes scarce, the temptation to replace it with synthetic data grows. If that substitution turns into a closed feedback loop, model collapse, weaker code diversity, and a higher rate of hallucinations become structural rather than accidental effects [2, 3]. For software practice, the conclusion is clear: LLMs can accelerate routine work, but they cannot be treated as autonomous authorities. Their output must be read, tested, profiled, and checked against documentation and human judgment. Human control is not a temporary inconvenience. It is the condition that keeps programming knowledge from degrading into an echo of machine-generated averages.

References

1. Villalobos P., Ho A., Sevilla J., et al. Will we run out of data? Limits of LLM scaling based on human-generated data [Electronic resource] / arXiv, 2024. – Mode of access: https://arxiv.org/abs/2211.04325 – Date of access: 06.03.2026.

2. Shumailov I., Shumaylov Z., Zhao Y., et al. The Curse of Recursion: Training on Generated Data Makes Models Forget [Electronic resource] / arXiv, 2024. – Mode of access: https://arxiv.org/abs/2305.17493 – Date of access: 06.03.2026.

3. Gerstgrasser M., Schaeffer R., Dey A., et al. Is Model Collapse Inevitable? Breaking the Curse of Recursion by Accumulating Real and Synthetic Data [Electronic resource] / arXiv, 2024. – Mode of access: https://arxiv.org/abs/2404.01413 – Date of access: 06.03.2026.

4. Agarwal V., Pei Y., Alamir S., Liu X. CodeMirage: Hallucinations in Code Generated by Large Language Models [Electronic resource] / arXiv, 2025. – Mode of access: https://arxiv.org/abs/2408.08333 – Date of access: 06.03.2026.