

Harnessing Generative Artificial Intelligence and Retrieval-Augmented Generation for Advanced Software Engineering, Automated Code Synthesis, and Intelligent Debugging

Aman Kamal

Osmania University, Hyderabad, Telangana, India

Abstract

Generative artificial intelligence (AI), particularly large language models (LLMs) trained on code, has quickly become part of everyday software engineering. These models can write functions from natural-language descriptions, explain unfamiliar code, and suggest fixes for defects. Used on their own, however, they often produce code that ignores project-specific conventions, calls nonexistent interfaces, introduces security weaknesses, or misdiagnoses bugs. Retrieval-augmented generation (RAG) addresses part of this problem by grounding generation in relevant project knowledge such as source files, interface definitions, documentation, and past defects. This article develops an integrated framework for applying generative AI and RAG to code synthesis and debugging in enterprise settings. Using a design-oriented approach grounded in ten studies published between 2016 and 2023, the framework links knowledge indexing, context retrieval, generation, automated verification, iterative debugging, and governed approval workflows. The article discusses how the framework applies to distributed service-based systems and resource-constrained monitoring applications, and illustrates it with a debugging scenario in a smart monitoring platform. It argues that generative AI delivers reliable engineering value only when generation is grounded in project context, checked by automated verification, and governed by structured review.

Keywords: generative AI, large language models, retrieval-augmented generation, code generation, automated program repair, intelligent debugging, software engineering, distributed systems, AI governance

Introduction

Software engineering has always sought to automate its own work, from compilers and integrated development environments to static analyzers and continuous integration. Generative AI marks a larger step. Chen et al. (2021) showed that an LLM fine-tuned on public code could solve a substantial share of programming problems from natural-language descriptions, and that sampling several candidate solutions raised the success rate considerably. AI coding assistants based on such models spread rapidly. In a controlled experiment, Peng et al. (2023) found that developers with access to an AI pair programmer completed a programming task about 56% faster than those without.

Speed is not the same as quality. Pearce et al. (2022) assessed code suggested by an AI assistant

in security-relevant scenarios and found that around 40% of the generated programs contained vulnerabilities. Models trained on public code also know nothing about an organization's private codebase, its internal interfaces, or its history of defects. As a result, they may generate code that compiles in isolation but fails in context.

Retrieval-augmented generation offers a way to close this context gap. Lewis et al. (2020) combined a neural retriever with a generator so that outputs are conditioned on documents retrieved at query time, which improved factual accuracy on knowledge-intensive tasks. Applied to code, retrieval can supply the model with relevant parts of the repository. Zhang et al. (2023) showed that iterative retrieval of repository code improved repository-level code completion. The study on *Leveraging Generative*

Artificial Intelligence and Retrieval-Augmented Generation to Improve Software Engineering, Automated Code Generation, and Intelligent Debugging Systems (2023) brings these ideas together for code generation and debugging.

This article asks: *How can generative AI and RAG be integrated into enterprise software engineering so that code synthesis and debugging are context-aware, verifiable, and governed?* It makes three contributions: (1) a synthesis of research on generative AI for code, RAG, program repair, and the risks of AI-generated code; (2) a layered framework that connects retrieval, generation, verification, debugging, and governance; and (3) an application of the framework to distributed and resource-constrained systems, illustrated through a practical debugging scenario.

Background and Related Work

Large language models for code

Chen et al. (2021) introduced Codex and the HumanEval benchmark, which measures whether generated functions pass unit tests. Their results established functional correctness, rather than textual similarity, as the key measure for code generation. They also showed that generating many candidates and selecting among them substantially improves outcomes. That finding motivates the combination of generation with automated testing used in this article.

Retrieval-augmented generation for software

Lewis et al. (2020) showed that retrieval helps models produce more accurate and specific outputs and allows their knowledge to be updated without retraining. In software engineering, relevant knowledge is spread across source files, interface specifications, configuration, documentation, commit history, and issue trackers. Zhang et al. (2023) proposed RepoCoder, which alternates between retrieving similar code from the repository and generating completions. Each round's partial output improves the next retrieval, and the approach clearly outperformed completion without retrieval. The 2023 study on *Leveraging Generative AI and RAG* applies RAG across the software lifecycle. It shows how retrieval of project-specific knowledge can improve

code generation and support intelligent debugging systems that diagnose faults using relevant code and past defect information.

Automated program repair and debugging

Automated program repair aims to fix bugs with little human effort. Xia et al. (2023) conducted an extensive study of large pre-trained language models applied to program repair across several benchmarks and languages. They found that these models, used directly, could fix substantially more bugs than earlier specialized repair techniques, and that giving models the surrounding code context improved results further. Their findings support using LLMs as debugging assistants, especially when combined with context retrieval and test-based validation.

Productivity and risks

The evidence on AI-assisted development is mixed. Peng et al. (2023) report large productivity gains on a well-defined task. Pearce et al. (2022) report frequent security weaknesses in generated code, including issues related to common weakness categories such as injection and improper input validation. Together, these studies suggest that generative AI should be treated as a powerful but fallible contributor whose output needs verification and review, not as an autonomous developer.

Distributed and resource-constrained contexts

Much enterprise software runs as distributed services. The study on *Service Discovery and Communication in Distributed Enterprise Applications* (2017) examines how services locate one another and exchange messages reliably, and the mechanisms and design choices involved. Faults in such systems often come from mismatched interfaces, stale service registrations, timeouts, and failure-handling logic rather than from single functions. This makes them hard for code models that see only local context. Similarly, the study on *Performance Analysis of Wireless Sensor Networks for Smart Monitoring Applications* (2016) examines performance factors in sensor networks used for monitoring. These include energy consumption, latency, throughput, and reliability under resource constraints. Software for such systems must respect tight limits that general-

Table 1: Framework for grounded, verified, and governed AI-assisted software engineering

<i>Layer</i>	<i>Function</i>	<i>Key mechanisms</i>	<i>Supporting studies</i>
1. Knowledge indexing	Make project knowledge searchable	Index source code, interface specifications, service contracts, documentation, defect history, system constraints	<i>Leveraging Generative AI and RAG</i> (2023); Lewis et al. (2020)
2. Context retrieval	Supply the model with relevant context	Similarity search, iterative retrieval, dependency-aware selection	Zhang et al. (2023); Lewis et al. (2020)
3. Generation	Synthesize code, tests, and fixes	Code LLMs, multiple candidate generation	Chen et al. (2021); <i>Leveraging Generative AI and RAG</i> (2023)
4. Automated verification	Filter incorrect or unsafe output	Unit and integration tests, static analysis, security scanning	Chen et al. (2021); Pearce et al. (2022)
5. Intelligent debugging	Diagnose and repair failures iteratively	Failure-context retrieval, candidate patches, test-based validation	Xia et al. (2023); <i>Leveraging Generative AI and RAG</i> (2023)
6. Governed approval	Control what reaches production	Risk-based review workflows, approvals, audit trails	Basireddy (2022)

purpose code models do not know about unless those limits are supplied as context.

Governance of engineering change

Basireddy (2022) examines digitizing risk management through configurable enterprise workflow architecture, in which identification, assessment, approval, escalation, and remediation steps are defined as configurable workflows with audit trails. AI-generated code is a new source of engineering risk, so this approach offers a way to govern it. Changes can be routed through review and approval steps that depend on risk, and each decision can be recorded.

Research gap

Prior work shows that LLMs can generate and repair code, that retrieval improves context-awareness, and that generated code carries real risks. Less attention has been paid to how these elements should be combined into an engineering process that is grounded, verified, and governed, especially for distributed and resource-constrained systems. The framework below addresses this gap.

Research Methodology

The study follows a design-oriented research approach in four steps. First, the core problem was identified from the literature: generative AI for code is productive but often ungrounded, unverified, and ungoverned. Second, design requirements were

derived: project-specific context, functional and security verification, iterative debugging, support for distributed and constrained systems, and auditable governance. Third, a layered framework was designed from the ten studies reviewed. Fourth, the framework was demonstrated through an illustrative debugging scenario. The demonstration shows how the framework works in practice. It does not measure the framework's performance, which is left to future empirical work.

Proposed Framework

The framework has six layers (Table 1). Retrieval grounds generation, verification filters its output, debugging iterates on failures, and governance controls what reaches production.

Knowledge indexing

The framework indexes more than source code. It also includes interface definitions and service contracts, configuration, architecture documentation, past defect reports and their fixes, and system constraints such as memory, energy, or latency budgets. Including service contracts and discovery configuration supports reasoning about distributed systems (*Service Discovery and Communication*, 2017). Including performance constraints supports resource-constrained applications (*Performance Analysis of Wireless Sensor Networks*, 2016).

Table 2: Generative AI capabilities, risks, and framework controls

<i>Capability</i>	<i>Main risk</i>	<i>Framework control</i>	<i>Supporting studies</i>
Code synthesis from natural language	Code that is functionally incorrect or misses context	Retrieval of project context, multiple candidates, test-based selection	Chen et al. (2021); Zhang et al. (2023)
Code completion in large repositories	Calls to nonexistent or outdated interfaces	Repository-level and dependency-aware retrieval	Zhang et al. (2023)
Automated bug fixing	Patches that pass tests but are wrong	Validation against tests, human review of diagnosis	Xia et al. (2023)
Faster development	Security weaknesses introduced unnoticed	Security scanning, risk-based approval workflows	Pearce et al. (2022); Peng et al. (2023); Basireddy (2022)
Distributed system changes	Interface mismatches and communication failures	Retrieval of service contracts and discovery configuration	<i>Service Discovery and Communication</i> (2017)
Constrained device code	Violating resource budgets	Retrieval of performance constraints, constraint-aware verification	<i>Performance Analysis of WSNs</i> (2016)

Context retrieval

For each task, the retriever selects the most relevant knowledge: similar functions, the interfaces the code must call, related tests, and past fixes for similar failures. Following Zhang et al. (2023), retrieval can repeat, with an initial draft used to refine the search for context. Because the model's context window is limited, retrieval should favor direct dependencies and recent changes over loosely related material.

Generation

With retrieved context, the model generates code, tests, or fixes. Following Chen et al. (2021), the framework generates several candidates rather than one, so that verification can select among them. Generated tests are treated as candidates too, and a developer reviews them before they are trusted.

Automated verification

Every candidate passes through automated checks. Unit and integration tests check functional correctness. Static analysis checks style and common defects. Security scanning targets the vulnerability types that Pearce et al. (2022) found in AI-generated code. Candidates that fail are discarded or sent back to the debugging layer. Only verified candidates move on to human review.

Intelligent debugging

When a test fails or a production incident occurs, the debugging layer gathers the failure evidence: error

messages, stack traces, logs, and the failing test. It then retrieves the relevant code and past similar defects (*Leveraging Generative AI and RAG*, 2023). The model proposes an explanation and one or more candidate patches, which are validated against the tests (Xia et al., 2023). The cycle repeats until a patch passes or the case is handed to a developer with the diagnosis attached.

Governed approval

Verified changes enter configurable review workflows (Basireddy, 2022). Risk-based rules decide the review path. A small change to test code may need a single reviewer, while AI-generated changes to authentication, payment logic, or service communication may need senior review and security sign-off. Every step, including which model and context produced the change, is recorded, which supports accountability and later analysis of AI-related defects.

Application to Code Synthesis and Debugging

Context-aware code synthesis

Retrieval makes code synthesis project-aware. When a developer asks for a new function that calls an internal service, the framework retrieves the service's interface definition, examples of existing calls, and the project's error-handling conventions. The generated code then uses real interfaces and

follows local patterns, avoiding plausible-looking but nonexistent calls. Table 2 summarizes how the framework matches common capabilities with their risks and controls.

Debugging distributed services

Distributed faults are hard to debug because symptoms appear far from their causes. A timeout in one service may result from a changed interface, a stale registration, or a retry policy in another. The framework addresses this by indexing service contracts and communication configuration (*Service Discovery and Communication*, 2017) and by retrieving logs and code across service boundaries. The model can then reason about how services interact, not just about one function in isolation.

Resource-constrained monitoring applications

Monitoring applications built on sensor networks must balance data quality against energy use, latency, and network reliability (*Performance Analysis of Wireless Sensor Networks*, 2016). Generated code that samples too often, retransmits too aggressively, or keeps radios active too long can drain batteries or congest the network. The framework counters this by retrieving documented constraints and performance findings as context and by including resource checks, such as simulated energy or message-count limits, in automated verification.

Illustrative Scenario: Debugging a Smart Monitoring Platform

Consider an organization operating a smart environmental monitoring platform. Wireless sensor nodes measure temperature and air quality in several facilities and send readings through gateways to a set of backend services for ingestion, alerting, and reporting. The services locate one another through a service registry.

Symptom

After a release, operators notice that alerts from one facility arrive late or not at all, while dashboards still show data.

Evidence gathering

The debugging layer collects gateway logs, ingestion and alerting service logs, and recent commits. Logs

show intermittent timeouts when the ingestion service calls the alerting service.

Retrieval

The framework retrieves the alerting service's interface contract, the registry configuration, the ingestion service's client code, and a past incident involving similar timeouts. It also retrieves the platform's documented latency and retransmission constraints for sensor data.

Diagnosis and repair

Using this context, the model notes that the release changed the alerting endpoint's request format and added a new instance, but the client was still sending requests in the old format to a stale registered address, with an aggressive retry policy. The model proposes two candidate patches. Both update the client to the new contract and refresh service discovery before retries. One also adds exponential backoff to avoid flooding the network during failures.

Verification

Both patches pass the integration tests. The second also passes a check that limits retransmissions from gateways, consistent with the platform's network constraints, so it is selected.

Governed approval

Because the change affects inter-service communication, the workflow routes it to a senior engineer for review and records the diagnosis, retrieved context, and test results (Basireddy, 2022). After approval, the patch is deployed and alert latency returns to normal.

The scenario shows the framework's central argument. The model's value came from reasoning over retrieved, system-specific context, and the outcome was trustworthy because it was verified and reviewed.

Discussion

Implications

For research, the framework positions generative AI as one component in an engineering system, with retrieval, verification, and governance as equally important parts. This view reconciles the evidence of large productivity gains (Peng et al., 2023) with

the evidence of security weaknesses (Pearce et al., 2022). The gains are realized safely only when output is grounded and checked. For practitioners, the framework suggests clear priorities. Index broad project knowledge, not just code. Generate multiple candidates and let tests choose among them. Put security scanning in the loop. Route AI-generated changes through review paths set according to risk.

Limitations and future work

The framework was demonstrated through an illustrative scenario rather than a measured deployment, and it rests on a focused set of ten studies. Future work should evaluate it empirically by measuring defect rates, security findings, review effort, and time to resolve incidents in real projects. Further research is needed on retrieval strategies for very large, multi-repository systems, on detecting patches that pass tests but are still wrong, on encoding non-functional constraints such as energy and latency in verification, and on how much autonomy AI agents should have in the engineering workflow.

Conclusion

Generative AI has changed what is possible in software engineering, from writing code to diagnosing and repairing faults. On its own, however, it lacks project context and makes errors that can be costly. This article proposed a six-layer framework that grounds generation in indexed project knowledge through retrieval, filters it through automated functional and security verification, iterates on failures through intelligent debugging, and governs the results through configurable, auditable approval workflows. Applied to distributed services and resource-constrained monitoring systems, the framework shows how retrieval of service contracts and system constraints makes AI assistance relevant

to the systems engineers actually build. The central conclusion is that the value of generative AI in software engineering depends as much on the context, checks, and governance around the model as on the model itself.

References

1. Basireddy, S. R. (2022). Digitizing risk management through configurable enterprise workflow architecture. *SAMRIDDHI: A Journal of Physical Sciences, Engineering and Technology*, 14(4), 231–239.
2. Chen, M., Tworek, J., Jun, H., Yuan, Q., Pinto, H. P. de O., Kaplan, J., ... Zaremba, W. (2021). *Evaluating large language models trained on code* (arXiv:2107.03374). arXiv.
3. Cherukuri, M. (2023). Leveraging Generative Artificial Intelligence and Retrieval-Augmented Generation to Improve Software Engineering, Automated Code Generation, and Intelligent Debugging Systems. *International Journal of Humanities and Information Technology*, 5(04), 140-163. <https://doi.org/10.21590/ijhit.05.04.11>
4. Lewis, P., Perez, E., Piktus, A., Petroni, F., Karpukhin, V., Goyal, N., ... Kiela, D. (2020). Retrieval-augmented generation for knowledge-intensive NLP tasks. *Advances in Neural Information Processing Systems*, 33, 9459–9474.
5. Pearce, H., Ahmad, B., Tan, B., Dolan-Gavitt, B., & Karri, R. (2022). Asleep at the keyboard? Assessing the security of GitHub Copilot's code contributions. In *Proceedings of the 2022 IEEE Symposium on Security and Privacy* (pp. 754–768).
6. Peng, S., Kalliamvakou, E., Cihon, P., & Demirel, M. (2023). *The impact of AI on developer productivity: Evidence from GitHub Copilot* (arXiv:2302.06590). arXiv.
7. Sannidhanam, A. H., & Alladi, S. (2016). Performance Analysis of Wireless Sensor Networks for Smart Monitoring Applications. *International Journal of Humanities and Information Technology*, 1(04), 10-29. <https://doi.org/10.21590/ijhit.01.04.04>
8. Sannidhanam, A.H., Alladi, S. (2017). Service Discovery and Communication in Distributed Enterprise Applications. *International Journal of Engineering and Science Research*, 7(4), 01-12. <https://ijesr.org/index.php/ijesr/article/view/1848>
9. Xia, C. S., Wei, Y., & Zhang, L. (2023). Automated program repair in the era of large pre-trained language models. In *Proceedings of the 45th International Conference on Software Engineering (ICSE)* (pp. 1482–1494).
10. Zhang, F., Chen, B., Zhang, Y., Keung, J., Liu, J., Zan, D., Mao, Y., Lou, J.-G., & Chen, W. (2023). RepoCoder: Repository-level code completion through iterative retrieval and generation. In *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing* (pp. 2471–2484).