Journal of Integrated Science, Technology and Management, 2025; 1(1):49-54

Journal of Integrated Science, Technology and Management

Research Article

Advancing Software Engineering Through Generative AI and Retrieval-Augmented Generation for Automated Code Development and Intelligent Debugging

Shishir Srvastava

Institute of Environment and Management, Lucknow, India

ARTICLE INFO

Article history:

Received: 10 November, 2025

Revised: 15 November, 2025

Accepted: 12 December, 2025

Published: 25 December, 2025

Keywords:

Generative AI, retrieval-augmented generation, software development lifecycle, automated code development, test generation, intelligent debugging, issue resolution, distributed systems, AI adoption, software governance

ABSTRACT

Generative artificial intelligence is changing each stage of the software development lifecycle, from understanding requirements and writing code to generating tests, resolving issues, and maintaining systems in production. Most research and tooling, however, concentrates on isolated tasks such as completing a line of code or solving a self-contained programming problem. Enterprise software development is a lifecycle activity carried out on large, interconnected codebases, and success depends on knowledge that general-purpose models do not have. This article takes a lifecycle perspective on generative AI in software engineering. It traces the progression of AI for code from learned code representations to competition-level generation. It explains why retrieval-augmented generation (RAG) is essential for applying these models to real projects. It then proposes an AI-augmented lifecycle model that assigns each development stage a defined AI role, a retrieval source, and a human checkpoint. The article also introduces a three-level adoption model, from assistive to supervised autonomous use, and illustrates both models with a feature development case in an enterprise asset-monitoring system. It concludes that organizations gain the most from generative AI when they treat it as a lifecycle capability, grounded in project knowledge and matched to appropriate levels of human oversight.

Introduction

The software development lifecycle has long been supported by tools that automate narrow tasks: compilers translate code, build systems assemble it, and test frameworks run checks. Generative AI differs in kind because it can produce the artifacts engineers work with, including code, tests, documentation, and fixes, from natural-language intent. This raises a practical question for organizations: where in the lifecycle should generative AI be used, what knowledge does it need at each stage, and how much should people oversee it?

Most early evidence came from benchmarks built around self-contained problems. Li et al. (2022) showed that AlphaCode could reach a competitive level in programming competitions by generating very large numbers of candidate programs and filtering them. Competition problems, however, arrive with complete specifications and no surrounding codebase. Enterprise work is different. Engineers extend existing systems, follow established interfaces, satisfy performance limits, and fix defects whose causes span many files. Jimenez et al. (2023) captured this gap with SWE-bench, a benchmark of real issues from open-source repositories. Models that performed well on self-contained tasks resolved only a small fraction of these real-world issues.

Retrieval-augmented generation offers a way to bring project knowledge into generation. Parvez et al. (2021) showed that retrieving relevant code and summaries

***Corresponding Author:** Shishir Srvastava

Address: Institute of Environment and Management, Lucknow, India

Relevant conflicts of interest/financial disclosures: The authors declare that the research was conducted in the absence of any commercial or financial relationships that could be construed as a potential conflict of interest.

© 2025, First author, This is an open access journal, and articles are distributed under the terms of the Creative Commons Attribution-NonCommercial-ShareAlike 4.0 License, which allows others to remix, tweak, and build upon the work non-commercially, as long as appropriate credit is given and the new creations are licensed under the identical terms.

improved both code generation and code summarization. The study on *Leveraging Generative Artificial Intelligence and Retrieval-Augmented Generation to Improve Software Engineering, Automated Code Generation, and Intelligent Debugging Systems* (2023) extends this idea to code generation and intelligent debugging across software engineering practice.

This article asks: *How can generative AI and RAG be systematically integrated across the software development lifecycle, and how should organizations calibrate human oversight as they adopt them?* It makes three contributions. First, it traces how AI for code has developed and why retrieval matters. Second, it proposes an AI-augmented lifecycle model that links each stage to an AI role, a retrieval source, and a human checkpoint. Third, it introduces a three-level adoption model and illustrates both models with a feature development case.

The Evolution of AI for Code

Learning representations of code

The transformer architecture (Vaswani et al., 2017) replaced recurrence with self-attention, so models could capture relationships between distant tokens and train efficiently at scale. It became the foundation for modern language models, including those for code. Feng et al. (2020) introduced CodeBERT, a model pre-trained on paired natural language and source code across several programming languages. CodeBERT learned joint representations that improved code search and code documentation tasks. It showed that code could be treated as a language with learnable structure and that meaning could be linked between natural-language descriptions and implementations.

From understanding to generation

Generative models moved beyond representation to producing code. Li et al. (2022) showed with AlphaCode that large-scale sampling combined with filtering against example tests could solve competitive programming problems at roughly the level of a median human competitor. The central lesson was that generation works best as a search process. Many candidates are produced, and verification selects the good ones. Chen et al. (2023) developed this idea with CodeT, which uses the model to generate test cases as well as code, then selects code by how consistently it passes the generated tests. This improved correctness without requiring human-written tests.

From problems to repositories

Real software engineering happens inside repositories. Jimenez et al. (2023) showed with SWE-bench that resolving real issues requires locating relevant code across many files, understanding existing behavior, and producing changes that pass the project's own tests. The low success rates reported for leading models at the time show that context, not raw generative ability, is the main bottleneck in applying AI to real projects. Table 1 summarizes this progression.

Why Retrieval Matters for Software Engineering

The knowledge gap

A general-purpose code model knows patterns common in public code. It does not know a specific organization's internal libraries, interface contracts, coding standards, deployment constraints, or the history of its defects. Without this knowledge it may produce code that looks right but calls nonexistent functions, breaks conventions, or repeats mistakes the organization has already fixed. Retrieval closes this gap by supplying relevant project knowledge when the model is asked to generate (Parvez et al., 2021; *Leveraging Generative AI and RAG*, 2023).

Table 1: Stages in the development of AI for code

Stage	*Representative work*	*Key capability*	*Remaining limitation*
Architecture	Vaswani et al. (2017)	Self-attention for long-range dependencies	Not specific to code
Code representation	Feng et al. (2020)	Joint natural-language and code understanding	Limited generation ability
Retrieval-augmented generation	Parvez et al. (2021)	Grounding generation in retrieved code and summaries	Retrieval quality limits output quality
Competition-level generation	Li et al. (2022)	Large-scale sampling and filtering	Self-contained problems only
Test-driven selection	Chen et al. (2023)	Generated tests used to select correct code	Generated tests may themselves be wrong
Repository-level issue resolution	Jimenez et al. (2023)	Realistic evaluation on real codebases	Low success rates for models of the time

Table 2: Retrieval knowledge sources and the lifecycle tasks they support

Knowledge source	*Examples*	*Lifecycle tasks supported*	*Supporting studies*
Source code	Functions, classes, similar implementations	Code development, refactoring	Parvez et al. (2021)
Interfaces and contracts	API specifications, service contracts, message formats	Integration, distributed development	*Service Discovery and Communication* (2017)
Tests	Existing unit and integration tests	Test generation, validation	Chen et al. (2023)
Defect and issue history	Bug reports, past fixes, incident notes	Debugging, issue resolution	*Leveraging Generative AI and RAG* (2023); Jimenez et al. (2023)
Non-functional constraints	Latency, energy, memory, throughput findings	Development for constrained environments	*Performance Analysis of WSNs* (2016)
Process rules	Review policies, approval requirements	Governance and release	Basireddy (2022)

Knowledge sources for retrieval

Enterprise projects hold several distinct kinds of knowledge, and each supports different lifecycle tasks. Table 2 sets out a practical taxonomy.

Two sources deserve particular attention. First, in distributed enterprise applications, correctness depends on how services find and communicate with one another. The study on *Service Discovery and Communication in Distributed Enterprise Applications* (2017) examines these mechanisms, which include service registration, lookup, and message exchange. Generated code that ignores them often fails at integration time. Second, in monitoring systems built on wireless sensor networks, software must work within limits on energy, bandwidth, and latency. The study on *Performance Analysis of Wireless Sensor Networks for Smart Monitoring Applications* (2016) examines these performance factors. Documented constraints of this kind are exactly what a model needs to generate suitable code for such systems.

Research Methodology

This study uses a design-oriented research approach. The problem, that generative AI is typically applied to isolated tasks rather than across the lifecycle, was identified from the literature on AI for code (Section 2). Requirements for a lifecycle approach were derived from the knowledge gap analysis (Section 3). Two artifacts were then designed: an AI-augmented lifecycle model and an adoption maturity model. Both were demonstrated through an illustrative feature development case. The demonstration shows how the models apply. It does not measure outcomes, which is left to future empirical research.

An AI-Augmented Software Development Lifecycle

The proposed lifecycle model assigns each development stage three elements: the role AI plays, the knowledge it retrieves, and the checkpoint where a person confirms or corrects its work (Table 3).

Table 3: AI-augmented software development lifecycle model

Lifecycle stage	*AI role*	*Primary retrieval source*	*Human checkpoint*
Requirements and design	Summarize related components, propose design options	Architecture documents, interface contracts	Engineer selects and refines the design
Code development	Generate implementation grounded in project conventions	Similar code, interfaces, constraints	Code review of generated changes
Test generation	Generate unit and integration tests, select code by test agreement	Existing tests, specifications	Engineer validates that tests reflect intent
Integration	Generate and check service communication code	Service contracts, discovery configuration	Integration review for cross-service changes
Debugging and issue resolution	Locate faults, explain causes, propose patches	Logs, defect history, related code	Engineer confirms diagnosis and patch
Release and governance	Summarize change risk and route for approval	Process rules, change history	Risk-based approval workflow

Requirements and design

At the start of a feature, AI can retrieve the components, interfaces, and documentation related to the request and summarize them for the engineer. This speeds up the orientation work that takes much of an engineer's time in large codebases. The model may propose design options, but the engineer decides.

Code development

During implementation, retrieval supplies similar existing code, the interfaces to be used, and relevant constraints (Parvez et al., 2021). The model generates code that follows local patterns. Generating several candidates and filtering them, as shown by Li et al. (2022), raises the chance that at least one candidate is correct.

Test generation

Following Chen et al. (2023), the model generates tests alongside code and uses agreement between them to select the most likely correct implementation. Generated tests can themselves be wrong, however, so the checkpoint at this stage is an engineer confirming that the tests reflect the intended behavior. Once confirmed, the tests become part of the knowledge base for future retrieval.

Integration in distributed systems

In service-based systems, generated code must register with, discover, and communicate with other services correctly (*Service Discovery and Communication*, 2017). The model retrieves service contracts and discovery configuration so that its generated client and server code matches them. Changes that cross service boundaries are flagged for integration review, because their failures are often visible only when services interact.

Debugging and issue resolution

When defects appear, the model retrieves logs, related code, and past similar defects, explains the likely cause, and proposes patches (*Leveraging Generative AI and RAG*, 2023). The patches are validated against the project's tests. The benchmark results of Jimenez et al. (2023) show that real issue resolution remains difficult, which supports keeping an engineer responsible for confirming both the diagnosis and the fix.

Release and governance

Before release, AI-generated changes pass through configurable approval workflows (Basireddy, 2022). The model can summarize what changed, which components are affected, and which parts were AI-generated. Workflow rules then decide the approval path by risk. Changes to security-critical code, cross-service communication, or constrained device software receive stricter review. Each step is recorded for audit.

Adoption Maturity Model

Organizations do not adopt generative AI all at once. The appropriate level of autonomy depends on the task, the risk, and the organization's experience. The article proposes three levels.

Level 1: Assistive

AI suggests code, tests, or explanations that engineers accept, edit, or reject one at a time. Retrieval is limited to the files currently open. Human control is complete, and this level suits organizations beginning adoption or working on high-risk code.

Level 2: Augmented

AI performs multi-step tasks such as implementing a feature with tests or proposing a validated patch for an issue. It uses repository-wide retrieval across the knowledge sources in Table 2. Engineers review complete changes at defined checkpoints (Table 3). This level suits most enterprise development once retrieval and automated testing are mature.

Level 3: Supervised autonomous

AI resolves well-defined, low-risk tasks from start to finish, such as dependency updates, routine fixes, or test coverage improvements. It works within limits set by governance workflows (Basireddy, 2022). People approve results according to risk-based rules rather than reviewing every step. This level requires strong automated verification, reliable retrieval, and a track record at Level 2.

Moving between levels should be decided per task category, not for the whole organization, and should be based on measured outcomes such as defect rates and review findings.

Illustrative Case: Adding a Sensor Type to an Asset-Monitoring System

Consider an enterprise that monitors industrial equipment using wireless sensor nodes. The nodes report readings through gateways to a set of backend services that handle ingestion, analytics, and notification, and the services find one another through a service registry. The organization needs to add support for a new vibration sensor. The work involves changes to gateway firmware, the ingestion service, and the analytics service. The team operates at Level 2.

Design

The engineer describes the feature. The model retrieves the existing sensor handling code, the ingestion service's message contract, and the performance documentation for the sensor network. It summarizes these and proposes adding a new message type, not overloading an existing one. The engineer agrees.

Code development

For the gateway firmware, the model retrieves the documented energy and bandwidth constraints (*Performance Analysis of WSNs*, 2016). It generates code that batches vibration readings and transmits summaries rather than raw samples, keeping transmission within the network's limits. For the ingestion service, it generates a handler that follows the retrieved message contract and existing handler patterns.

Test generation

The model generates unit tests for the new handler and uses them to choose among several candidate implementations (Chen et al., 2023). The engineer notices that one generated test assumes readings always arrive in order. Because the network can deliver readings out of order, the engineer corrects the test.

Integration

The analytics service must subscribe to the new message type. The model retrieves the service contracts and registry configuration (*Service Discovery and Communication*, 2017) and generates the subscription code. The change crosses service boundaries, so it is flagged for integration review.

Debugging

During integration testing, the analytics service rejects some messages. The model retrieves the error logs and the relevant contract and identifies a mismatch in a timestamp field format between the ingestion output and the analytics input. It proposes a patch, which passes the tests after the engineer confirms the diagnosis.

Release

The approval workflow routes the firmware change to an embedded systems reviewer and the cross-service change to a senior backend engineer, recording the AI-generated portions of each (Basireddy, 2022). After approval, the feature is released.

The case shows how the lifecycle model distributes AI support across stages, how retrieval of contracts and constraints shapes generated code, and how human checkpoints caught an error that automated generation missed.

Discussion

Implications

For research, the article moves attention from single-task benchmarks to the lifecycle. Its main claim is that the value of generative AI depends on matching each stage with the right knowledge and the right oversight. It also frames autonomy as something that can be raised or lowered for each task category according to evidence. For practitioners, the article offers two tools. The lifecycle model shows where to introduce AI and which knowledge sources to prepare. The adoption model shows how to expand AI's role safely over time. Interface contracts, non-functional constraints, and defect history are the retrieval sources organizations most often overlook, and they are often the ones that matter most.

Limitations and future work

The models were demonstrated through an illustrative case rather than empirical evaluation, and they draw on a focused set of ten studies. Future research should assess the lifecycle and adoption models in real development settings. It should measure productivity, defect rates, review effort, and the accuracy of AI-generated tests and diagnoses. Further work is needed on criteria for moving task categories between adoption levels, on keeping retrieval indexes current as codebases change, and on detecting errors in generated tests before they reinforce incorrect code.

Conclusion

Generative AI for code has progressed from learning representations of code to generating competitive solutions and attempting real repository issues. The main barrier to applying it in enterprise software engineering is context rather than capability. This article argued for a lifecycle approach in which retrieval supplies the project knowledge each stage needs and human checkpoints are placed where AI errors are most likely and most costly. It proposed an AI-augmented lifecycle model, a three-level adoption model, and a feature development case that showed both in action. Organizations that ground generative AI in their own knowledge, verify its output at every stage, and widen its autonomy only as evidence supports will be best placed to turn its promise into dependable engineering practice.

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, B., Zhang, F., Nguyen, A., Zan, D., Lin, Z., Lou, J.-G., & Chen, W. (2023). CodeT: Code generation with generated tests. In *Proceedings of the 11th International Conference on Learning Representations (ICLR)*.
3. Feng, Z., Guo, D., Tang, D., Duan, N., Feng, X., Gong, M., … Zhou, M. (2020). CodeBERT: A pre-trained model for programming and natural languages. In *Findings of the Association for Computational Linguistics: EMNLP 2020* (pp. 1536–1547).
4. Jimenez, C. E., Yang, J., Wettig, A., Yao, S., Pei, K., Press, O., & Narasimhan, K. (2023). *SWE-bench: Can language models resolve real-world GitHub issues?* (arXiv:2310.06770). arXiv.
5. 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
6. Li, Y., Choi, D., Chung, J., Kushman, N., Schrittwieser, J., Leblond,

R., ... Vinyals, O. (2022). Competition-level code generation with AlphaCode. *Science, 378*(6624), 1092–1097.
7. Parvez, M. R., Ahmad, W. U., Chakraborty, S., Ray, B., & Chang, K.-W. (2021). Retrieval augmented code generation and summarization. In *Findings of the Association for Computational Linguistics: EMNLP 2021* (pp. 2719–2734).
8. 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
9. 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
10. Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A. N., Kaiser, Ł., & Polosukhin, I. (2017). Attention is all you need. *Advances in Neural Information Processing Systems, 30*, 5998–6008.

HOW TO CITE THIS ARTICLE: Srvastava, S. (2025). Advancing Software Engineering Through Generative AI and Retrieval-Augmented Generation for Automated Code Development and Intelligent Debugging. *Journal of Integrated Science, Technology and Management*, 1(1), 49-54.