State-RAG: A Two-Layer Retrieval System for Large Language Model-Driven Website Builders

Cheemaladinne Vengaiah
Assistant Professor, Department of CSE (DS)
B V Raju Institute of Technology
Narsapur, India
vengaiah.cse@gmail.com

Pulakani Sidhanth Reddy
Department of CSE (DS)
B V Raju Institute of Technology
Narsapur, India
22211a67a1@bvrit.ac.in

Markuk Bhanu Prakash Swamy
Department of CSE (DS)
B V Raju Institute of Technology
Narsapur, India
22211a6776@bvrit.ac.in

Mandapati Sai Tiru Charan Reddy
Department of CSE (DS)
B V Raju Institute of Technology
Narsapur, India
22211a6771@bvrit.ac.in

***Abstract*—Large Language Models (LLMs) perform well on isolated code generation. Interactive website development presents different challenges. Projects evolve through many iterations, combining AI-generated code with manual edits. Current systems struggle to maintain consistent project state. Conversational memory degrades over time. Traditional RAG treats retrieved code as suggestions rather than authoritative state. This leads to overwritten user edits, hallucinated dependencies, and unpredictable behavior. This paper introduces State-RAG, a dual-layer retrieval architecture separating reusable design knowledge (Global RAG) from authoritative project state (State-RAG). The State-RAG layer stores validated, versioned software artifacts as complete snapshots of components, pages, and configuration files. Each artifact represents definitive project state, not probabilistic suggestions. At each interaction, relevant artifacts are projected into the LLM context, transforming the model into a pure transformation engine. Authority, validation, and conflict resolution happen externally. Manual user edits automatically become authoritative and cannot be accidentally overwritten. Validation prevents invalid code from entering project state. Behavior becomes deterministic. The paper describes the artifact model, retrieval contract, and execution pipeline enabling this hybrid workflow for scalable website construction.**

***Index Terms*—Large language models, retrieval-augmented generation, state-aware systems, AI-assisted website development, software artifacts, human-AI collaboration, deterministic code generation**

I. Introduction

LLM-assisted code generation has changed developer workflows. Tools like GitHub Copilot and ChatGPT are now standard practice. Developers rely on these systems for everything from writing boilerplate to debugging complex logic. When these systems handle interactive website construction through dozens of iterations mixing AI generation with manual edits, they fail in predictable ways.

Consider a common scenario: a developer builds a landing page, starting with a navigation bar. The LLM generates clean code. The developer manually adjusts colors, spacing, and hover states to match brand guidelines. The result looks perfect. Later, when requesting a footer, the LLM regenerates the navbar with original styling, silently overwriting manual changes. The carefully tuned brand colors revert to generic defaults. The model may hallucinate components that don't exist, reference files never created, or assume fictional state. A button click handler might reference a modal that was discussed but never implemented. Import statements point to non-existent utility files. Each interaction compounds these errors. By the tenth iteration, the project exists in a superposition of what the model thinks is there and what actually is.

Modern LLMs write excellent code in isolation. Given a clear specification and no dependencies, they produce working implementations. The issue is architectural. Current approaches treat LLMs as stateful agents relying on conversation history or as knowledge-enhanced generators using traditional RAG. Both miss what matters for iterative development, which is authoritative access to current state, not probabilistic guesses about what might exist.

Conversational memory creates an illusion of persistence. The model appears to remember previous decisions and build coherently on them. Context windows fill quickly. Earlier details compress or get pruned. Manual edits made outside conversation remain invisible to the model. A developer might spend an hour refactoring component structure in their editor, but the next LLM interaction assumes the old architecture still exists. The system thinks it knows project state, but that knowledge is incomplete and increasingly stale. This gap between perceived state and actual state grows with every manual edit and every context window rotation.

Traditional RAG retrieves relevant code snippets but treats them as suggestions. Retrieved context informs generation but carries no special authority. The model fills remaining gaps with assumptions, often incorrect ones. There is no mechanism to declare retrieved artifacts as ground truth versus advisory context. A retrieved component might represent the current implementation or might be an example from documentation. The model has no way to distinguish. Without that distinction, even perfect retrieval cannot guarantee state consistency. You might retrieve exactly the right files, and the model might still choose to regenerate them based on outdated assumptions baked into its weights.

This paper introduces State-RAG to address these limitations through explicit state management and authority boundaries.

II. Related Work

State-RAG draws from retrieval-augmented generation, code generation with language models, and state management in interactive systems. Existing work addresses or fails to address the challenge of maintaining consistent project state during iterative AI-assisted development.

A. Retrieval-Augmented Generation

Lewis et al. [6] introduced RAG by combining dense retrieval with seq2seq models, demonstrating that external knowledge could reduce hallucinations in knowledge-intensive tasks. REALM [11] and RETRO [12] integrated retrieval into pre-training itself. Recent surveys [1] show RAG evolving from simple kNN retrieval toward sophisticated multi-hop reasoning. Self-RAG [13] introduced reflection tokens, letting models critique their own retrieved context. However, these advances focus on improving retrieval quality and reasoning about uncertainty. There is no mechanism for establishing authoritative state boundaries. Existing RAG systems treat all retrieved content as equally provisional. For interactive development, there is a need to distinguish between ground truth artifacts and advisory knowledge.

B. Code Generation and Repository Understanding

Code generation benchmarks traditionally focused on standalone functions. HumanEval [10] and MBPP measure whether models can write isolated Python functions. StarCoder [14] and Code Llama [15] improved generation quality through massive code pre-training but still target single-file scenarios. Repository-level generation emerged as models grew larger. SWE-bench [16] requires models to resolve real GitHub issues involving multiple files and complex dependencies. CodeRAG [8] uses bigraph retrieval to find relevant context across repositories. Shrivastava et al. [17] studied how retrieval helps with cross-file code completion, finding that naive similarity search often misses structural dependencies. Yang et al. [3] conducted an empirical study identifying key challenges including outdated snippets, semantically similar but structurally incompatible code, and incomplete information. These systems retrieve relevant code but do not distinguish between code that exists in the current project and example code that might be helpful.

C. LLM Memory and Interactive Systems

Conversational models implicitly maintain state through context windows with well-documented limitations [18]. As conversations lengthen, earlier information gets compressed or pruned. Zhong et al. [19] proposed MemPrompt, augmenting prompts with explicitly stored user feedback, but still relying on heuristic selection of what to remember. Generative Agents [20] introduced a sophisticated memory architecture with separate stores for observations, reflections, and plans, but targeted open-ended simulation, not deterministic software construction. The memory remains probabilistic without guarantees of consistency. Software requires completeness guarantees, not relevance ranking.

D. Human-AI Collaborative Development

Studies of GitHub Copilot usage [21] reveal a consistent pattern where developers appreciate suggestion quality but struggle with control and predictability. Vaithilingam et al. [22] found that programmers expect AI tools to respect their edits and maintain project context, but current systems frequently violate these expectations in subtle, hard-to-debug ways. Barke et al. [23] introduced Grounded Copilot, grounding suggestions in the actual codebase by analyzing imports and definitions. This moves toward authoritative state but grounds suggestions rather than managing state evolution. MetaGPT [24] proposes multi-agent collaboration with role-based specialization but lacks explicit state boundaries, making coordination increasingly fragile as projects grow.

III. System Architecture

The State-RAG architecture explicitly separates responsibilities for state, retrieval, and generation. The core design goal is preventing the LLM from implicitly defining or remembering project state while enabling effective participation in multi-step website construction.

A. Architectural Overview

State-RAG follows a dual-layer retrieval design with Global RAG for advisory knowledge and State-RAG for authoritative state, coordinated by an external orchestration layer. The LLM is treated as a stateless transformation engine, never storing, inferring, or reconstructing project state internally. That responsibility lies entirely with the system architecture. Figure 1 illustrates the high-level architecture and interaction flow between user, orchestration layer, retrieval systems, validation logic, and the LLM.

The orchestration layer acts as the single source of truth, mediating all interactions between components. It enforces invariants that cannot be violated through LLM hallucination or creative interpretation. When the LLM proposes changes, those proposals remain hypothetical until explicitly validated and committed by the orchestrator. This separation prevents the model from accidentally corrupting project state through plausible but incorrect assumptions. The architecture treats LLM outputs as untrusted suggestions requiring verification, not as authoritative declarations of reality.

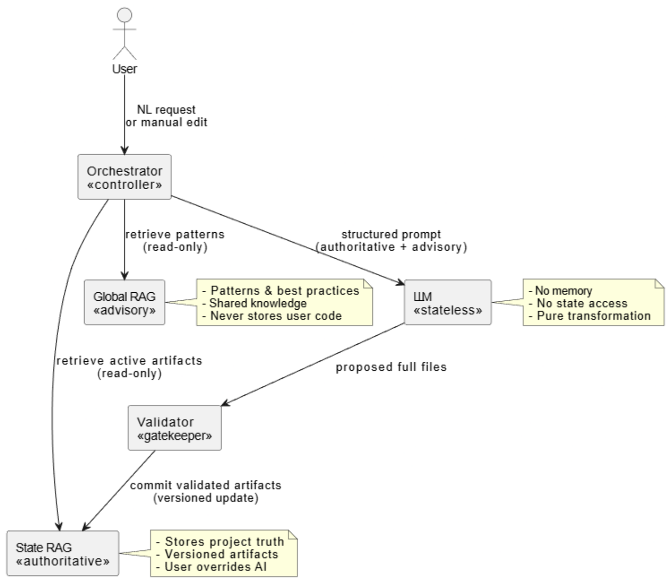
User
NL request or manual edit
Orchestrator «controller»
retrieve patterns (read-only)
structured prompt (authoritative + advisory)
Global RAG «advisory»
- Patterns & best practices
- Shared knowledge
- Never stores user code
LLM «stateless»
- No memory
- No state access
- Pure transformation
retrieve active artifacts (read-only)
proposed full files
Validator «gatekeeper»
commit validated artifacts (versioned update)
State RAG «authoritative»
- Stores project truth
- Versioned artifacts
- User overrides AI

Fig. 1. High-level architecture of the State-RAG system

B. Dual-RAG Separation

Strict separation between general knowledge and project-specific state is foundational to how the system establishes authority. The Global RAG stores reusable, slow-changing knowledge such as layout conventions, accessibility guidelines, and framework-specific best practices. Contents are advisory, never authoritative. It does not store user code and plays no role in state definition or conflict resolution.

State-RAG maintains authoritative project representation, storing validated, versioned software artifacts. Each artifact is a complete snapshot of a project element such as component, page, or configuration file. At any point, exactly one version of an artifact is active. User-modified artifacts take precedence over AI-generated or AI-modified ones, establishing a clear authority hierarchy: User $>$ System $>$ AI. This separation ensures Global RAG retrieval guides generation without overriding project state, while State-RAG retrieval always reflects current ground truth.

C. Artifact-Centric State Representation

Project state is represented using an artifact abstraction combining source code with structured metadata. Artifacts are created only through concrete system events, which are manual user edits or validated AI outputs. Partial codes and invalid syntaxes are not persisted. This ensures State-RAG always represents a consistent, buildable project state.

Before admission into State-RAG, artifacts undergo multi-stage validation. Syntactic checks use AST parsing for the target language. Semantic validation includes type resolution, import verification, and dependency graph consistency. Framework-specific checks apply linting rules. Only validated artifacts enter the authoritative state. Figure 2 shows artifact lifecycle and versioning model. Each accepted modification creates a new version while preserving history, enabling roll-back without re-invoking the LLM.

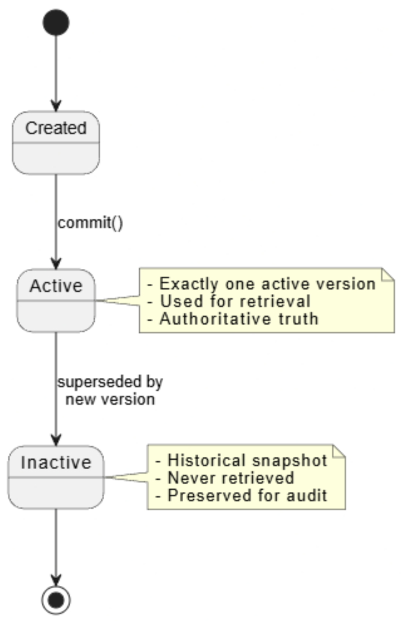

Fig. 2. Artifact lifecycle and versioning model

D. Design Rationale

By externalizing state and enforcing artifact-level validation and authority, the architecture avoids treating retrieved context as probabilistic suggestions. Each LLM invocation operates over a clearly defined, authoritative state snapshot, not a best-effort approximation. This design enables scalable, reproducible workflows that safely combine manual edits with AI-assisted generation. The system remains debuggable even after hundreds of interactions because state transitions are explicit, validated, and auditable.

IV. Execution Pipeline

State-RAG processes user interactions through a controlled execution pipeline ensuring every state transition is explicit, validated, and reproducible. The pipeline handles both natural language requests and manual code edits using a unified abstraction.

A. Pipeline Overview

Each interaction begins with a user action, either a natural language request or a manual code edit, and ends with a validated update to project state. The LLM never modifies state directly and has no memory of previous interactions beyond what is explicitly provided in current context. Figure 3 presents the full execution flow showing decision points where validation can reject proposals before they affect authoritative state.

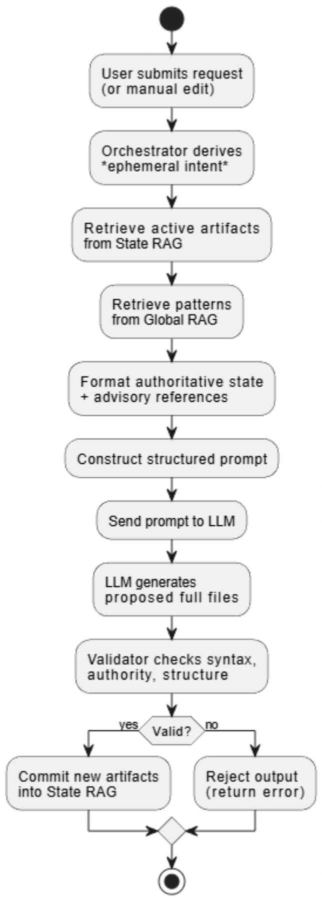
User submits request (or manual edit)
Orchestrator derives *ephemeral intent*
Retrieve active artifacts from State RAG
Retrieve patterns from Global RAG
Format authoritative state + advisory references
Construct structured prompt
Send prompt to LLM
LLM generates proposed full files
Validator checks syntax, authority, structure
Valid?
yes
no
Commit new artifacts into State RAG
Reject output (return error)

Fig. 3. End-to-end execution pipeline for State-RAG

B. Intent Interpretation and Retrieval

For natural language requests, the orchestrator derives a temporary intent specification including action type, target scope, and constraint information. This intent guides retrieval and prompt construction but is never persisted as project state. Based on resolved scope, the system retrieves relevant active artifacts from State-RAG. Retrieval is constrained by scope and dependency relationships. If the user wants to modify the navigation bar, the system retrieves that component plus any components it imports or that import it. In parallel, relevant advisory patterns are retrieved from Global RAG. Manual code edits bypass intent interpretation entirely, with the edited file directly triggering artifact creation after validation.

C. Prompt Construction and Generation

Retrieved artifacts and advisory references are assembled into a structured prompt with explicit authority markers. State-RAG artifacts are labeled as [CURRENT STATE] while Global RAG content is labeled as [REFERENCE]. The LLM generates complete proposed files, not partial diffs, simplifying validation. The model is explicitly constrained to operate only on provided artifacts. If a required component is not in retrieved context, the model must declare a dependency rather than hallucinating implementation.

D. Validation and State Commitment

LLM outputs are untrusted proposals until validation. Each output undergoes syntactic validation, semantic validation, and authority validation. Outputs failing validation are rejected outright and never enter project state. The user receives an error explaining what went wrong, and system state remains unchanged. Validated outputs create new artifact versions, replacing previously active versions in State-RAG. The versioning mechanism preserves complete history. If the user dislikes a change, they can roll back to any previous version without regenerating code.

V. Experimental Results

The evaluation focuses on system-level behavior under realistic, multi-step website construction workflows. Since the primary contribution is architectural rather than model-centric, evaluation examines how different architectural approaches handle persistent state, user authority, and iterative evolution. Four approaches are compared using the same underlying LLM (Gemini-2.5-flash) on an identical test suite.

A. Evaluation Protocol

The evaluation uses a comprehensive test suite comprising over fifty test cases grouped into six categories covering basic CRUD operations, multi-file workflows, authority and safety constraints, versioning behavior, dependency management, and edge cases. Each test case defines initial project state, a user request, and expected outcomes plus forbidden outcomes. For each test case, all four methods are initialized with identical initial files.

B. Metric Definitions

We use six system-level metrics, each corresponding to a known failure mode in interactive LLM-driven development:

Hallucination Rate measures frequency of invalid dependency references. A hallucination occurs when generated code imports a non-existent artifact, a file neither present in initial state nor created during the current interaction. Computed as invalid references divided by total references. Lower is better.

Consistency Score captures output stability under repeated execution. For fixed project state and identical request, we run the interaction multiple times and measure the proportion of identical outputs. Normalized to [0,1]. This measures surface-level consistency and identical code but not semantic equivalence.

Authority Preservation evaluates whether user-modified artifacts are respected. For State-RAG, this records whether the system correctly rejects unauthorized modifications to user-controlled files. Baseline approaches lack explicit authority mechanisms and therefore score zero by definition as this metric represents a capability distinction, not a performance comparison.

Scope Adherence measures whether changes remain confined to intended targets. A test case scores positively if no files outside the allowed set are created or modified. Any modification to explicitly forbidden files constitutes a scope violation. For example, if the user asks to "add a button to the header" but the system also modifies the footer, that's a violation.

Versioning Correctness applies only to systems with explicit versioned state. Correct behavior requires that exactly one active version exists for each modified artifact and version identifiers evolve monotonically with each accepted change. Systems without versioning score zero.

Dependency Tracking assesses whether generated artifacts explicitly reference required dependencies via import statements. This captures dependency relationship presence, not semantic optimality. We're measuring "did it import what it needs" not "did it import the minimum necessary."

Together, these metrics emphasize correctness, control, and long-term usability—properties difficult to evaluate using isolated prompt-level benchmarks.

C. Results Overview

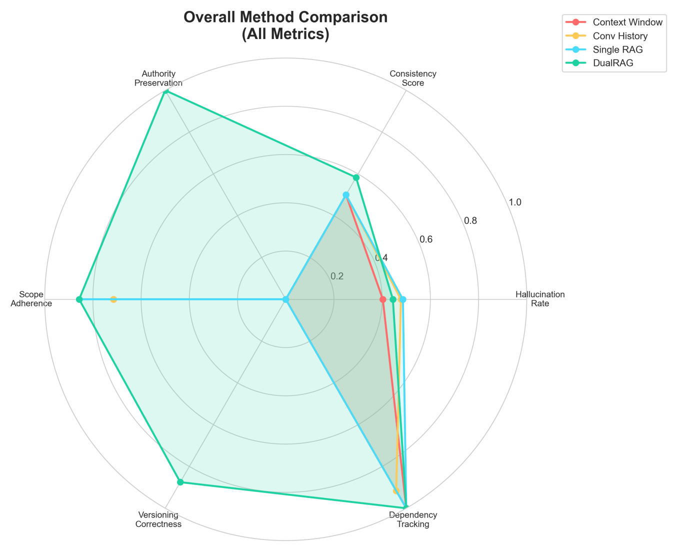

Fig. 4. Radar Chart (Overall Method Comparison)

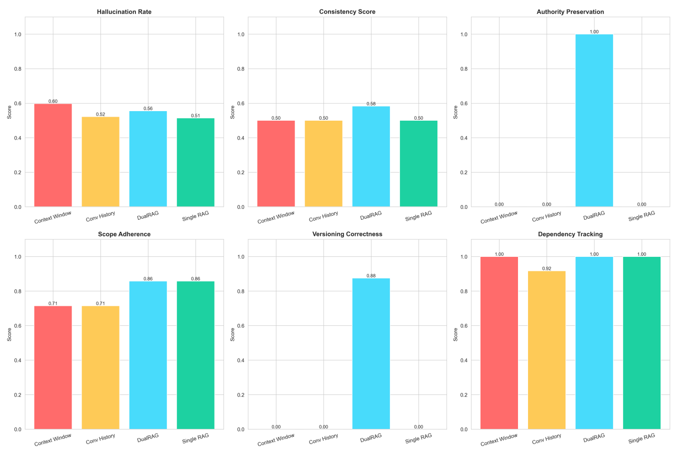

Fig. 5. Per-Metric Bar Plots

Figure 5 presents a radar comparison across all evaluated metrics, while Figure 6 provides per-metric bar plots. State-RAG outperforms other approaches in authority preservation, versioning correctness, and scope adherence, which are metrics directly reflecting its explicit state and authority model. Context-window and conversational-history approaches degrade as interactions accumulate, particularly when manual edits occur outside immediate prompt context.

TABLE I
CONSISTENCY AND STATE MANAGEMENT COMPARISON

Approach	**State**	**Auth.**	**Consistency**
Conv. Memory	Implicit	None	Degrades
Traditional RAG	Advisory	Equal	Partial
Grounded Copilot	Explicit (R)	None	Single-turn
State-RAG	**Explicit (R/W)**	**User>AI**	**Guaranteed**

Retrieval-based methods reduce hallucination rates relative to pure context-based approaches. However, the most substantial reduction occurs when retrieval is combined with authoritative state enforcement. This indicates that retrieval alone is insufficient and how retrieved information is treated matters more than retrieval quality itself.

D. Discussion

Consistency scores reveal an important distinction between stability and correctness. Single RAG achieves relatively high consistency, producing similar outputs across runs, but qualitative inspection shows this often reflects repeated reproduction of the same incorrect assumptions. State-RAG exhibits slightly lower raw consistency as outputs vary across runs, but maintains correct state evolution. Variability in generated code does not translate into uncontrolled state changes because all outputs pass validation and reconcile against authoritative artifacts before commitment. State-RAG constrains state transitions, not model outputs. The evaluation demonstrates that reliability improvements stem primarily from architectural choices rather than model capabilities.

VI. CONCLUSION

This work introduced State-RAG, an artifact-centric retrieval architecture for reliable, multi-step interaction in LLM-driven website builders. Unlike conversational memory or traditional RAG, State-RAG externalizes project state as validated and versioned artifacts, treating this state as authoritative rather than advisory. By separating reusable design knowledge from project-specific state, the architecture constrains the language model to controlled transformation over explicit inputs.

Through formalizing an artifact model, authority hierarchy, and retrieval contract, State-RAG avoids common failure modes in interactive development including unintended overwrites of user-modified code, dependency hallucinations, and unstable behavior across iterations. The evaluation shows that reliability improvements arise from architectural enforcement of state and authority, not from improvements in generation quality alone.

State-RAG does not assume deterministic language model outputs. It enforces deterministic state evolution. Given the same project state, user intent, and validation rules, state transitions remain explicit, reproducible, and auditable. Variability in generated code is contained by validation and versioning, preventing uncontrolled drift over long interaction sequences.

While this work focuses on interactive website construction, the underlying principles of authoritative state projection, explicit authority boundaries, and controlled retrieval apply to a broader class of AI-assisted software development systems. Future work will explore extending to collaborative multi-user scenarios with merge conflict resolution, investigating learned validation models that predict code correctness, and applying State-RAG principles to other domains such as mobile app development and backend service construction.

References

[1] Y. Gao, Y. Xiong, X. Gao, K. Jia, J. Pan, Y. Bi, Y. Dai, J. Sun, M. Wang, and H. Wang, “Retrieval-Augmented Generation for Large Language Models: A Survey,” arXiv preprint arXiv:2312.10997, 2024.

[2] Y. Tao, Y. Qin, and Y. Liu, “Retrieval-Augmented Code Generation: A Survey with Focus on Repository-Level Approaches,” arXiv preprint arXiv:2510.04905, 2025.

[3] Z. Yang, S. Chen, C. Gao, Z. Li, X. Hu, K. Liu, and X. Xia, “An Empirical Study of Retrieval-Augmented Code Generation: Challenges and Opportunities,” ACM Transactions on Software Engineering and Methodology, vol. 34, no. 7, Article 188, 2025.

[4] Y. Dong, X. Jiang, J. Qian, T. Wang, K. Zhang, Z. Jin, and G. Li, “A Survey on Code Generation with LLM-Based Agents,” arXiv preprint arXiv:2508.00083, 2025.

[5] U. Umama, K. U. Danyaro, M. Nasser, A. Zakari, S. Abdullahi, A. Khanzada, M. M. Yakubu, and S. Shoaib, “LLM-Based Code Generation: A Systematic Literature Review with Technical and Demographic Insights,” IEEE Access, 2024.

[6] P. Lewis et al., “Retrieval-Augmented Generation for Knowledge-Intensive NLP Tasks,” Advances in Neural Information Processing Systems (NeurIPS), 2020.

[7] S. M. Abtahi and A. Azim, “Augmenting Large Language Models with Static Code Analysis for Automated Code Quality Improvements,” arXiv preprint arXiv:2506.10330, 2025.

[8] J. Li, X. Shi, K. Zhang, G. Li, Z. Jin, L. Li, and C. Tao, “CodeRAG: Supportive Code Retrieval on Bigraph for Real-World Code Generation,” arXiv preprint arXiv:2504.10046, 2025.

[9] I. Saberi and F. Fard, “Context-Augmented Code Generation Using Programming Knowledge Graphs,” in Proceedings of the 42nd International Conference on Machine Learning (ICML), 2025.

[10] M. Chen et al., ”Evaluating Large Language Models Trained on Code,” arXiv:2107.03374, 2021.

[11] K. Guu et al., ”REALM: Retrieval-Augmented Language Model Pre-Training,” ICML, 2020.

[12] S. Borgeaud et al., ”Improving Language Models by Retrieving from Trillions of Tokens,” ICML, 2022.

[13] A. Asai et al., ”Self-RAG: Learning to Retrieve, Generate, and Critique through Self-Reflection,” arXiv:2310.11511, 2023.

[14] R. Li et al., ”StarCoder: May the Source Be with You!” arXiv:2305.06161, 2023.

[15] B. Rozière et al., ”Code Llama: Open Foundation Models for Code,” arXiv:2308.12950, 2023.

[16] C. Jimenez et al., ”SWE-bench: Can Language Models Resolve Real-World GitHub Issues?” ICLR, 2024.

[17] S. Shrivastava et al., ”Repository-Level Prompt Generation for Large Language Models of Code,” arXiv:2206.12839, 2023.

[18] L. Wang et al., ”Unleashing Infinite-Length Input Capacity for Large-scale Language Models,” arXiv:2308.16137, 2023.

[19] W. Zhong et al., ”MemPrompt: Memory-assisted Prompt Editing with User Feedback,” EMNLP, 2023.

[20] J. S. Park et al., ”Generative Agents: Interactive Simulacra of Human Behavior,” UIST, 2023.

[21] N. Vaithilingam et al., ”Expectation vs. Experience: Evaluating the Usability of Code Generation Tools Powered by Large Language Models,” CHI, 2022.

[22] N. Vaithilingam et al., ”Expectation vs. Experience,” CHI, 2022.

[23] S. Barke et al., ”Grounded Copilot: How Programmers Interact with Code-Generating Models,” OOPSLA, 2023.

[24] S. Hong et al., ”MetaGPT: Meta Programming for Multi-Agent Collaborative Framework,” arXiv:2308.00352, 2024.