Knowledge Retrieval Architectures for AI-Assisted Software Development: From Static Context to Autonomous Development Agents

Norbert Fijałek [0009-0004-2756-0451],
Ilona Bluemke [0000-0002-2894-5976],
Piotr Gawrysiak [0000-0002-9647-6761]

Faculty of Electronics and Information Technology
Warsaw University of Technology
Warsaw, Poland
norbert.fijalek.dokt@pw.edu.pl

Abstract-**The integration of Large Language Models (LLMs) into software development workflows has fundamentally transformed how developers interact with knowledge repositories, codebases, and documentation. However, traditional knowledge retrieval mechanisms face significant challenges when applied to the dynamic, multi-modal, and contextually-rich environment of software engineering. This survey provides a comprehensive analysis of knowledge retrieval architectures specifically designed for AI-assisted software development, examining the evolution from static context provision to autonomous development agents. Building upon this systematic examination, the paper delineates critical research directions that advance the theoretical and practical foundations of AI-assisted software development.**

Keywords-Knowledge Retrieval, Large Language Models, Retrieval-Augmented Generation, Agentic RAG, Graph RAG, AI-Assisted Software Development, AI SDLC.

I. INTRODUCTION

The rapid advancement of Large Language Models has created opportunities for AI-assisted software development, yet their effectiveness fundamentally depends on how they access and utilize knowledge. While LLMs demonstrate remarkable capabilities in code generation, their reliance on static parametric knowledge creates limitations in software engineering's dynamic environment, where frameworks and libraries evolve continuously. This challenge intensifies when enterprise codebases contain millions of lines of code with complex interdependencies that exceed even advanced models' context windows. This survey addresses a critical gap in existing literature: while numerous studies examine individual retrieval architectures or specific AI coding assistants, no comprehensive analysis systematically compares knowledge retrieval approaches specifically designed for software development contexts. This work provides a unified framework for understanding how different architectural approaches handle the distinct challenges of software development knowledge retrieval. This survey employs a systematic literature review methodology inspired by the guidelines proposed by Kitchenham [30], adapted to the specific characteristics of knowledge retrieval research in AI-assisted software development.

The review was guided by three research questions: (RQ1) What knowledge retrieval architectures have been proposed or applied in AI-assisted software development, and what are their core mechanisms? (RQ2) What challenges do these architectures face when applied to software engineering contexts? (RQ3) What gaps and future directions emerge from the current body of work? The search was conducted across three databases: IEEE Xplore, ACM Digital Library, and arXiv, covering publications from 2020 to 2025 to capture the evolution following transformer-based models' widespread adoption. The search strategy employed combinations of the following terms: "retrieval-augmented generation" AND "software development"; "knowledge retrieval" AND "code generation"; "graph RAG" OR "agentic RAG"; "LLM" AND "software engineering" AND "retrieval"; and "AI coding assistant" AND "knowledge". Additionally, supplementary sources were identified through citation tracking of key foundational works-including seminal contributions on few-shot learning capabilities of large language models [5], prompt engineering patterns and programming paradigms [3, 4], chain-of-thought reasoning [6], and the foundational retrieval-augmented generation framework [7]-as well as through targeted searches in Google and Google Scholar for works not indexed in the primary databases. Subsequent survey literature examining RAG system design and limitations [8] further informed the scope of the review. Two foundational Polish-language textbooks on artificial intelligence for engineers [1-2] were included as they provide comprehensive theoretical frameworks for AI paradigms.

The initial search yielded approximately 100 candidate sources. Inclusion criteria required that works: (1) address knowledge retrieval mechanisms in the context of AI-assisted software development or closely related code generation tasks; (2) be published in peer-reviewed venues, established preprint repositories, or constitute recognized reference works; (3) be written in English or Polish. Exclusion criteria removed works that: (1) focused exclusively on general-purpose RAG without software engineering application; (2) were short workshop abstracts or position papers lacking technical depth; (3) were duplicates or earlier versions superseded by more comprehensive publications; (4) lacked methodological substance. After applying these criteria, 50 candidate sources remained for full-text review. From these, a final set of 30

Manuscript received June 22, 2026; revised July 08, 2026;
accepted August 07, 2026; published August 25, 2026.
Issue category: Special Issue on DSD/SEAA 2026 on Works in Progress (WiP) Session, Krakow, Poland, Sept. 2026.
Paper category: Full
DOI: doi.org/10.64552/wipiec.v12i2.147

works was selected based on their direct relevance to knowledge retrieval architectures for software development, methodological rigor, and contributions to system design principles, prioritizing works addressing software engineering's unique challenges. The classification of architectures into four paradigms (Context Window and Prompting, Traditional RAG, Graph RAG, and Agentic RAG) emerged from iterative analysis of the extracted data, with each work assigned to one or more categories based on its primary contribution, where section II establishes foundational concepts of LLM knowledge handling and retrieval-augmented approaches. Section III analyzes five interconnected challenge categories: information veracity, contextual understanding, scale and performance, workflow integration, and knowledge representation. Section IV surveys available architectures-Context Window and Prompting, traditional RAG, Graph RAG, and Agentic RAG-examining their mechanisms, strengths, and limitations. Section V presents future directions of research in terms of hybrid agentic architectures.

II. FUNDAMENTALS

Richard Hamming's words that "the purpose of computing is insight, not numbers" [20] has never been more relevant than in the current era of Large Language Models, which have fundamentally transformed how artificial intelligence systems store and access knowledge. Traditional models keep all their information in parameters learned during training, which works well for general language tasks but creates significant problems in specialized fields like software development [5]. The challenge becomes particularly apparent in software engineering, where programming languages, frameworks, and libraries evolve much faster than any training process can capture.

This limitation of static knowledge has led to the development of RAG (Retrieval-Augmented Generation), which represent a major advancement in AI systems [7]. The necessity for such approaches becomes evident when considering that classical computer science methods, where deterministic algorithms are constructed to solve problems, have in many cases reached a development barrier [1]. While traditional approaches rely solely on pre-trained knowledge, retrieval-augmented systems can access external information sources to stay current with rapidly changing technical domains. The knowledge encoded during training becomes like a snapshot frozen in time, making it impossible to incorporate new developments that are essential for effective software development support.

The foundation of modern retrieval systems lies in vector space models and sophisticated knowledge representation techniques. These systems create high-dimensional spaces where related concepts cluster together, allowing computers to calculate similarity using methods such as cosine similarity. Knowledge representation involves symbolic encoding of facts that agents recognize as true, with mapping between facts and their representations in given systems [2]. This approach helps identify relevant code examples based on their meaning rather than just matching keywords. Current code embeddings tend to capture surface-level similarities more effectively than deeper semantic relationships [8]. This creates a particular challenge when working with different types of information-code, documentation, and architectural diagrams all require different approaches to knowledge extraction and representation. Context window limitations present another significant obstacle for software engineering applications. Modern software projects often require understanding context that far exceeds what current language models can handle [21]. Even with recent advances like Gemini 1.5, which can process millions of tokens [10], the technology still falls short of enterprise-scale requirements.

The evolution from static to dynamic AI systems reflects broader patterns in artificial intelligence development. The field of natural language processing has evolved from classical rule-based approaches to statistical and deep learning methods, with historical approaches using formal language knowledge while modern approaches rely on statistical analysis of large text datasets and neural networks trained on massive corpora [2]. Modern IDEs (Integrated Development Environment) improved with semantic analysis understanding variable scope and type systems, but remained fundamentally reactive without broader architectural understanding. AI-powered coding assistants mark a major shift toward sophisticated knowledge retrieval systems that address the mechanization and engineering instrumentation challenges inherent in creating methods for solving technical problems [1]. These systems must handle the full complexity of modern software development: code generation, transformation, testing, analysis, and configuration management [18]. Each area requires different strategies, showing that single-approach solutions aren't adequate. This diversity raises questions about whether there exists an optimal architecture for all use cases, or whether specialized architectures would be more appropriate.

Recent studies have documented these limitations in real-world scenarios. Research demonstrates how AI coding assistants encounter difficulties with enterprise-scale codebases that exceed current context windows by several orders of magnitude [17]. Similarly, studies found that effectiveness varies significantly based on programming language coverage in training data and the amount of contextual information that can be accommodated within model constraints [16]. These findings raise important questions about whether specialized training approaches might be needed for software development contexts.

Modern retrieval systems must also handle the multi-modal nature of software engineering knowledge. Effective development assistance requires combining textual documentation, code examples, architectural diagrams, API specifications, and configuration files. Traditional text-based retrieval methods prove insufficient for this diverse information environments and need specialized architectures that can navigate complex knowledge landscapes.

These foundational concepts reveal both the promise and limitations of current knowledge retrieval approaches in software engineering. The subsequent sections examine the

practical challenges these systems face (Section III) and how different architectural paradigms address these constraints (Section IV).

III. Unified Challenges in Knowledge Retrieval for AI-Assisted Software Development

Building AI systems that can effectively help software developers faces several interconnected challenges. These difficulties arise from both the limitations of current AI technology and the specific nature of software development work. Importantly, the complexity and severity of these challenges differ significantly between two levels of AI assistance [17]: basic coding assistants (focused primarily on code completion and syntax suggestions) and comprehensive software engineering support (requiring architectural understanding, system design decisions, and cross-component reasoning) [18]. Understanding these problems is important because they affect each other and cannot be solved separately, and their manifestations vary dramatically depending on whether the AI aims to assist with isolated coding tasks or complete software engineering workflows. This section systematically examines five interconnected challenge categories that emerge from the literature. We begin with Information Veracity and Reliability Challenges, exploring how AI systems struggle with accuracy and explainability in code generation. Next, Contextual Understanding and Integration Challenges addresses the complexities of combining multi-modal knowledge across different technical domains. Scale and Performance Challenges examines computational limitations and sustainability concerns when processing enterprise-scale codebases. Development Workflow Integration Challenges investigates the practical barriers to incorporating AI systems into existing development ecosystems. Finally, Knowledge Representation and Modeling Challenges analyzes the unique characteristics of software engineering knowledge that distinguish it from other AI application domains. Each subsection identifies key works from our systematic literature review that document these challenges, providing a comprehensive foundation for understanding why current retrieval architectures face fundamental limitations (examined in Section IV) and why hybrid approaches become necessary (explored in Section V).

A. Information Veracity and Reliability Challenges

Software development requires precise information because wrong answers can lead to serious problems like system crashes or security breaches. Recent studies of AI coding assistants in production environments reveal consistent patterns of accuracy challenges, particularly in code generation hallucinations [8] and performance degradation at enterprise scale [17]. AI systems make mistakes in different ways when helping developers. Sometimes they create code that looks correct but breaks programming rules or uses features that don't exist [8]. This happens more often with less popular programming languages because the AI systems have less training data about them, causing them to incorrectly apply patterns from other well-known languages like Python or JavaScript [18].

Another common problem occurs when AI systems generate code that follows grammar rules but doesn't work in the real context. For example, they might suggest using a function that doesn't exist or use an existing function in the wrong way [16]. Real-world studies show that AI-generated code often contains subtle errors that developers need to catch and fix manually [17]. Making matters worse, software technologies change constantly. New versions of programming languages and frameworks regularly introduce changes that can make existing knowledge outdated within just a few months [18].

A fundamental challenge lies in the explainability of AI-generated solutions. Often, problems are caused by the so-called explainability issue-demonstrating why the result of the AI mechanism for a given set of data is this and not another [1]. This lack of transparency becomes particularly problematic in software development contexts where developers need to understand not just what code to write, but why specific approaches are recommended. Responsible AI principles emphasize ensuring that AI system outputs are beneficial and non-intrusive for users, especially when dealing with large populations [2]. Systems must be designed to prevent the perpetuation of existing inequalities or discriminatory decision-making. Ensuring data diversity and representativeness used for training AI systems is crucial for limiting bias [2]. In software development contexts, this translates to ensuring that AI coding assistants don't perpetuate poor coding practices or introduce systematic biases that could affect software quality or security.

B. Contextual Understanding and Integration Challenges

Modern software development requires combining knowledge from many different technical areas. A single programming task might need understanding of how code works, specific framework patterns, database design, security requirements, and performance optimization [7]. Research examining multi-modal knowledge processing and enterprise codebase navigation [2, 7, 10] reveals that the challenge isn't just finding information about each area-it's combining this knowledge while avoiding conflicts between different approaches. The complexity of contextual understanding is further complicated by the inherent characteristics of natural language processing in software contexts. Natural language is characterized by changing vocabulary and semantic functions, fluidity and incompleteness of syntax rules [2]. This means that natural language is neither a closed nor finite system. The semantics of sentences can change over time, and it's never possible to definitively determine whether a sentence belongs to the language [2]. When AI systems must interpret developer queries, documentation, and code comments, these linguistic challenges compound the technical complexity of providing accurate assistance.

Additionally, software development happens in constantly changing environments. Code repositories grow and evolve every day, and software dependencies get updated regularly. This creates complex relationships between different parts of a system that are rarely documented properly [10]. Current AI systems struggle to understand these complex ecosystems

where multiple code repositories and services depend on each other in complicated ways.

C. Scale and Performance Challenges

Today's software projects are often much larger than what current AI models can handle. Large enterprise applications can contain millions of lines of code spread across thousands of files with complex dependencies [9]. Studies of large codebase processing and real-time response optimization [9-11] demonstrate that even the most advanced AI models like Sonnet 3.5 [28], which can process millions of tokens, still face practical limitations when dealing with real-world enterprise software [10].

The computational demands of modern AI systems present significant sustainability challenges. Training large language models requires energy comparable to what's needed for flights between Pacific and Atlantic coasts of the USA. Energy consumption for larger models like GPT-3 or GPT-4 is comparable to that needed to light a small city [2]. This creates a critical tension between the desire for more capable AI coding assistants and the environmental impact of the computational resources required to train and operate them.

Speed remains a critical issue. Developers expect quick responses from their tools. If an AI assistant takes more than a few seconds to respond, it significantly hurts productivity. This creates pressure to optimize systems in ways that might reduce the quality of answers [11]. Designing new AI methods to ensure high-quality operation while maintaining appropriate energy efficiency is essential [2] for creating sustainable and practical development assistance tools.

D. Development Workflow Integration Challenges

Integrating AI systems into existing development processes involves more than just technical challenges. Modern software development relies on complex tool ecosystems including code editors, version control systems, and automated testing and deployment systems [12]. Research on tool ecosystem compatibility, security concerns, and developer adoption patterns [12, 14, 17] shows that AI systems must work smoothly with all these tools while maintaining compatibility across different technology stacks, yet technical capabilities alone cannot ensure successful deployment. The sensitive nature of software development information creates significant privacy and security concerns. Code repositories often contain proprietary business logic and confidential information that needs protection [14]. Building trust among developers presents another barrier, with concerns about job security and becoming too dependent on automated systems [17].

E. Knowledge Representation and Modeling Challenges

Software engineering knowledge has unique characteristics that make it different from other areas where AI is used. Code serves both as instructions for computers and as communication between developers, creating complexities that traditional text-based approaches cannot handle properly [5]. Analysis of code semantics, multi-level dependency structures, and multi-modal information integration [3, 5-6, 13] demonstrates that current analysis methods focus mainly on grammar and structure, but effective assistance requires understanding deeper relationships that go beyond correct syntax [3]. Software systems show complex relationship patterns that exist at multiple levels. These include direct connections between modules, indirect coupling through shared data structures, and semantic relationships based on similar functionality [13]. Software knowledge also exists in multiple formats: documentation, code, diagrams, and configuration files. Effective systems need approaches that maintain consistency while preserving the unique characteristics of each type of information [6]. These challenges in AI-assisted software development form a connected web where problems with information accuracy, context understanding, scale limitations, workflow integration, and knowledge representation make each other worse [18, 11]. These fundamental limitations require specialized architectural approaches that can handle software engineering's unique characteristics: multiple types of knowledge, constantly changing information, and complex relationships across different levels of abstraction [10, 12].

IV. Overview of Available Architectures

The systematic literature review reveals a clear evolutionary trajectory in knowledge retrieval architectures for AI-assisted software development, progressing from simple static approaches toward increasingly sophisticated autonomous systems. Analysis of selected works demonstrates that researchers and practitioners have consistently sought to address the fundamental tension between system capability and implementation complexity. The literature shows four distinct architectural paradigms that have emerged in response to software engineering's unique challenges: Context Window & Prompting, which dominated early LLM applications; Retrieval-Augmented Generation (RAG), which gained prominence following the 2021 introduction of the RAG framework [7]; Graph RAG, which emerged in 2024 as researchers recognized the limitations of vector-based approaches for capturing complex software dependencies [9]; and Agentic RAG, which represents the current frontier with multi-agent systems appearing predominantly in 2023-2024 literature [12-14]. Each architectural paradigm addresses specific limitations of its predecessors while introducing new complexities, creating what the literature characterizes as a fundamental trade-off between sophisticated capabilities and practical deployment feasibility. This section examines each architecture systematically, analyzing how they handle the five challenge categories identified in Section III, and explaining why no single approach currently provides a complete solution for all software development contexts.

A. Context Window and Prompting

The simplest approach involves putting all necessary information directly into the AI model's input window. This method doesn't retrieve external knowledge but instead relies on carefully written prompts that include relevant context alongside the user's question [5]. This approach aligns with the symbolic artificial intelligence paradigm, which was dominant in early AI development-GOFAI (Good Old-fashioned Artificial Intelligence) [24] and uses high-level, human-understandable representation of problems [1]. Smart

prompting has become quite sophisticated over the years. Developers can create templates that work well with different programming languages by including language-specific patterns and common frameworks in their prompts [3]. These templates help maintain consistency across different coding tasks, whether generating new code, debugging problems, or writing documentation. Chain-of-thought prompting represents another advancement, where the AI breaks down complex problems into smaller steps, making the reasoning process more transparent [6]. Recent research has demonstrated that effective prompt engineering can significantly improve code generation quality and reduce the need for post-generation corrections [4].

The key strength of this approach lies in its simplicity and speed. When developers need quick answers to straightforward questions, context window prompting delivers immediate results. The technique works particularly well with few-shot learning, where providing a few examples helps the AI understand patterns and apply them to new situations [5]. However, the limitations become obvious when working with large codebases. As noted earlier, modern enterprise software projects contain millions of lines of code spread across thousands of files, far exceeding what any current AI model can process at once [10]. This forces developers to carefully select which information to include, often missing important context that could affect the AI's recommendations. Studies have shown that context window limitations significantly impact the quality of AI assistance in complex software engineering tasks [18].

B. Retrieval-Augmented Generation (RAG)

RAG systems take a different approach by first searching through external knowledge sources before generating responses. When a developer asks a question, the system finds relevant information from documentation, code repositories, or other knowledge bases, then provides this context to the AI model along with the original question [7]. This approach leverages statistical artificial intelligence methods, utilizing mathematical tools that allow for solving problems such as vector similarity and retrieve relevant information [1].

Building effective RAG systems for software development requires understanding how code differs from regular text. Code serves both as instructions for computers and communication between developers, creating unique challenges for knowledge retrieval [8]. These systems need specialized embedding models that can capture programming concepts and multiple indexing strategies to handle different types of relationships in software projects. The integration of external knowledge sources has been shown to significantly improve the accuracy and relevance of AI-generated code compared to purely parametric approaches [7]. Modern RAG systems employ reasoning engines that derive new facts from existing explicitly represented facts and axioms, enabling them to make intelligent connections between retrieved information [2].

One critical decision involves how to break down large codebases into manageable pieces. Function-level chunking treats each function as a separate unit, which works well for specific implementation questions but struggles with broader architectural concerns. File-level chunking keeps entire classes and modules together, preserving more design context but potentially including irrelevant information. Repository-level chunking captures the most context but risks overwhelming the system with too much information. RAG systems show significant improvements over basic prompting for many software development tasks. They can access current documentation, find relevant code examples from large repositories, and stay updated with recent changes. However, they still operate reactively, responding to specific queries rather than proactively understanding the developer's broader goals. Empirical evaluations have demonstrated that RAG-based systems consistently outperform traditional prompting approaches in code completion and documentation generation tasks [11]. Recent evaluation frameworks (such as RAGAS) [25] have established standard ways to measure RAG system performance across different tasks like fact verification and reasoning capabilities [11]. These metrics help organizations choose appropriate systems for their specific needs and optimize performance over time.

C. Graph RAG

While traditional RAG systems treat knowledge as composed of independent pieces of information, Graph RAG recognizes that software development involves complex relationships between different components. This approach builds on semantic networks, which provide graphical notation for knowledge represented as sets of nodes (concepts) connected by labeled arcs describing relationships between nodes [2]. Knowledge graphs serve as large, graph-structured knowledge bases that represent facts as relationships between entities, with basic components including entities expressed as graph nodes, their properties (attributes), and relations connecting nodes expressed as graph edges [2]. Creating these graphs requires sophisticated algorithms that can identify software engineering entities and understand how they relate to each other. Direct dependencies between modules represent the most obvious relationships, but software systems also have semantic connections based on shared functionality and temporal associations that track how code evolves over time. Entities in these knowledge graphs can have semantic types represented through is-a relationships between entities and their types, enabling more sophisticated understanding of code architecture [2]. Research has shown that graph-based representations can capture complex software dependencies more effectively than traditional flat document structures [9]. Graph RAG systems excel in scenarios that require understanding of these relationships. When developers need to assess the impact of proposed changes, these systems can trace dependency paths across multiple levels of software architecture. They help identify potential breaking changes, security implications, and performance considerations that might not be obvious from looking at individual code pieces. The contemporary manifestation of semantic networks through technologies like RDF, RDFS, and OWL enables the creation and dissemination of semantic networks and ontologies in standard form, facilitating knowledge sharing across

development teams [2]. The ability to traverse graph structures also enables comprehensive architectural understanding. Developers can explore system architecture from multiple perspectives, understanding not just what code does but how different parts interact and depend on each other. Query-focused summarization capabilities allow these systems to generate summaries that capture complex relationships and dependencies that simpler vector-based approaches would miss [9].

Despite these advantages, Graph RAG systems face significant challenges in maintenance and scalability. Building accurate graphs requires substantial computational resources, and keeping them updated as codebases evolve presents ongoing technical challenges. Studies indicate that the overhead of maintaining graph structures can be prohibitive for rapidly changing software projects [14]. These maintenance challenges echo earlier Semantic Web initiatives, where ontology evolution and curation consistently emerged as primary barriers to adoption [29]. Manual construction and maintenance of formal knowledge structures proved unsustainably labor-intensive, particularly in dynamic domains-challenges that software engineering's rapid technological evolution amplifies further.

D. Agentic RAG

The most advanced approach integrates autonomous agents with knowledge retrieval, creating systems that can reason independently and handle complex, multi-step tasks [12]. This approach exemplifies the subsymbolic artificial intelligence paradigm, which unlike symbolic and statistical approaches, does not assume any specific form of knowledge representation, allowing for more flexible and adaptive reasoning capabilities [1]. Unlike previous approaches that simply respond to developer questions, agentic systems maintain persistent memory, break down complex objectives into smaller tasks, and coordinate multiple operations over extended periods.

These systems represent a fundamental shift in how AI assists with software development. Rather than functioning as advanced search engines, they operate more like collaborative team members who can understand project goals, plan implementation strategies, and execute comprehensive workflows [12]. Multi-agent frameworks enable different specialized agents to work together, each contributing specific expertise while coordinating toward common objectives. Research has demonstrated that multi-agent systems can handle complex software engineering tasks that exceed the capabilities of single-agent approaches [13].

Agentic RAG systems can integrate directly with development tools, version control systems, testing frameworks, and deployment pipelines. This integration transforms them from passive information providers into active participants in the development process. They can autonomously execute testing procedures, manage code deployments, and even monitor system performance. Advanced implementations include learning mechanisms that enable continuous improvement through experience and feedback [13]. These systems remember successful strategies, learn from common failure patterns, and develop more effective approaches over time. The reasoning capabilities inherent in these systems allow them to derive new facts and insights from existing knowledge, enabling them to make intelligent decisions about development workflows [2]. The rapid progress in multi-agent systems research demonstrates significant potential while highlighting ongoing challenges in coordination, communication, and scalability [14].

The emergence of these sophisticated systems has already begun changing how developers work. Organizations report that AI systems increasingly function as team members rather than tools, creating new collaboration patterns that represent a significant shift in professional software development [17].

E. Architectural Evolution and Analysis

The progression from simple prompting to agentic systems shows clear advancement in both knowledge representation and autonomous capabilities. Each approach builds on previous paradigms while introducing new ways to handle software engineering's complex knowledge requirements. This evolution reflects the broader development of artificial intelligence approaches, from symbolic through statistical to subsymbolic paradigms, each offering distinct advantages for different aspects of software development assistance [1]. To systematically compare the characteristics of the identified architectural paradigms, Table I summarizes their key properties across four evaluation dimensions: knowledge access method, setup complexity, hallucination risk, and autonomy level. This comparison enables a structured assessment of the trade-offs between capability sophistication and practical deployment feasibility.

TABLE I. A COMPARISON ACROSS KEY DIMENSIONS OF DIFFERENT KNOWLEDGE RETRIEVAL ARCHITECTURES.

Architecture	Knowledge Access	Setup Complexity	Hallucination Risk	Autonomy Level
Context Window & Prompting	Manual/Static	Low	Medium	Low
Traditional RAG	Automated/Limited	Medium	Medium	Low
Graph RAG	Relational/Structured	High	Low-Medium	Medium
Agentic RAG	Multi-source/Dynamic	High	Low	High

This comparison highlights a fundamental pattern: more sophisticated architectures offer superior capabilities but require significantly more complex implementation and maintenance. Simple prompting and traditional RAG provide immediate practical value with straightforward deployment, while Graph RAG and Agentic RAG deliver advanced knowledge representation and autonomous capabilities at much higher implementation costs. Recent evaluation frameworks (such as RAGAS) [25] provide systematic approaches for comparing these architectures across multiple dimensions, helping organizations make evidence-based decisions about which approaches best fit their specific software development contexts [11]. While each architectural paradigm offers distinct

advantages for particular scenarios, none completely addresses all the complex requirements of modern software development environments. The limitations of individual approaches point toward the need for hybrid systems that combine strengths from multiple paradigms while maintaining practical deployment feasibility.

V. New Directions-Hybrid Agentic Architectures

Current knowledge retrieval systems for software development work well in some situations but fall short when developers need comprehensive assistance with complex projects. Each architecture examined in Section IV has clear strengths, yet none can handle all the challenges that modern software engineering presents. This gap points toward the need for hybrid approaches that combine different retrieval methods within intelligent agent systems-an area that remains largely unexplored and represents a critical direction for future research.

A. Theoretical Foundations for Hybrid Systems

Traditional RAG systems struggle to maintain context when working with large codebases that span multiple files and complex relationships [8]. Future hybrid agentic approaches could potentially address this limitation by employing autonomous agents that intelligently select which retrieval strategies matter most for any given task. These agents would need to maintain context across extended development sessions-a capability current systems lack. Developing such context-aware orchestration represents a fundamental research challenge requiring advances in multi-strategy coordination and adaptive retrieval.

B. Unified Knowledge Representation

A promising research direction involves investigating unified data processing approaches that convert all software engineering artifacts-code files, documentation, diagrams, configuration files, and version control history-into standardized formats before retrieval processing begins. This addresses a fundamental principle in artificial intelligence: representation matters [1]. Just as the multiplication of numbers is trivial in Arabic numerals but nearly impossible in Roman numerals, the effectiveness of knowledge processing depends critically on how information is represented [1]. By establishing common semantic foundations, future systems could enable both vector-based embeddings and graph-based relationship extraction to work with consistent data structures. The multi-modal nature of software engineering knowledge presents unique challenges that hybrid architectures would be well-positioned to address [2]. Modern software development involves processing diverse information types simultaneously-code, documentation, architectural diagrams, and configuration files-each requiring specialized processing methods yet ultimately working together to provide comprehensive development assistance [2]. This would allow seamless integration of insights from multiple information types when responding to developer queries-something current systems cannot achieve due to format inconsistencies across different retrieval methods.

C. Multi-Engine Integration Strategies

Future hybrid architectures should explore integration of multiple complementary engines: vector-based systems for semantic similarity, graph-based systems for relationship analysis, and agent-based systems for intelligent orchestration. The central research challenge lies in developing effective coordination mechanisms that can determine which retrieval strategy best fits specific query types, merge results from different retrieval methods intelligently, resolve conflicts, and assess confidence levels across heterogeneous information sources [11]. Such systems would need to run multiple retrieval operations in parallel while synthesizing results through sophisticated decision-making algorithms [12]-capabilities that require significant algorithmic advances beyond current state-of-the-art.

D. Agent-Based Knowledge Management

Integration of agent-based systems could draw from established principles in robotics and knowledge representation [2]. Frame-based knowledge representation might help autonomous agents understand their environment and interact effectively with software development contexts [2]. These agents would need to continuously maintain relationship mappings while keeping embeddings updated, both working from unified source representations. However, developing efficient parallel processing strategies and adaptive learning mechanisms represents substantial research challenges. Agents must be capable of managing knowledge base evolution independently-incorporating new information sources, updating existing structures, and retiring outdated information while maintaining semantic coherence [13].

E. Critical Evaluation Dimensions

Evaluating future hybrid agentic architectures will require assessing performance across multiple critical dimensions. These include strategy selection effectiveness-how well agents choose optimal retrieval method combinations; cross-modal integration quality-how successfully agents synthesize results from vector and graph-based approaches; learning and adaptation-how quickly agents improve strategy selection through experience; resource optimization-how efficiently agents balance computational costs across different retrieval modes; and temporal awareness-how effectively systems track knowledge evolution and maintain currency.

VI. Future Work

This survey points to several important research directions that require further investigation. A key area involves developing algorithms that can select the best retrieval strategy in real time, combined with techniques for merging information from different sources while keeping results consistent [11]. Equally important is finding ways to track rapidly changing software knowledge and making retrieval and reasoning processes more transparent and explainable [1]. Knowledge representation must move beyond current embeddings and graph structures to capture how software evolves over time-including changing requirements, code transformation patterns, and technical decision cascades. Successful deployment

requires not only technical progress but also organizational adaptation [17], while the regulatory landscape-particularly GDPR [26] and the EU AI Act [27]-has made explainable AI a critical requirement. These directions converge toward building genuinely collaborative systems that work alongside human developers rather than replacing them [17], maintaining transparency and meaningful human control while serving the broader goal of creating better software that benefits society.

VII. Conclusions

The evolution from static documentation to autonomous coding agents mirrors a broader truth about technological progress: each generation of tools doesn't simply do more-it fundamentally reshapes what becomes possible, transforming yesterday's architectural visions into tomorrow's reality. This survey systematically provides a unified theoretical and empirical foundation for understanding knowledge retrieval architectures in AI-assisted software development. The principal contributions are threefold: (1) a unified taxonomy of challenges across five interconnected dimensions demonstrating how limitations in one dimension exacerbate others; (2) a systematic comparative analysis of four architectural paradigms revealing a fundamental trade-off between capability and implementation complexity; (3) the identification of critical research directions toward hybrid agentic architectures requiring algorithmic innovations beyond the current state-of-the-art. The analysis reveals that retrieval architectures have progressed through distinct evolutionary stages-from symbolic paradigms through statistical approaches to subsymbolic methods-mirroring broader AI development patterns [1], with no single approach adequately addressing all modern software development requirements. Current systems demonstrate measurable improvements in code quality and developer productivity [16], yet struggle with complex architectural reasoning that experienced developers perform intuitively [15, 17]. This gap points toward hybrid architectures combining multiple retrieval mechanisms within autonomous agent frameworks. Research on memory-augmented systems and continual learning [19]-showing that effective retrieval must span factual memory, sense-making, and associative reasoning-provides evidence that such approaches are theoretically sound, though significant technical barriers remain.

As software systems continue to grow in scale and complexity, advancing knowledge retrieval architectures from static context provision to genuinely autonomous development agents remains both a promising and necessary direction for research and practice.

References

[1] M. Muraszkiewicz and R. Nowak (eds.), Sztuczna inteligencja dla inżynierów - metody ogólne. Warsaw, 2022 (in Polish).

[2] M. Muraszkiewicz and R. Nowak (eds.), Sztuczna inteligencja dla inżynierów - istotne obszary i zastosowania. Warsaw, 2023 (in Polish).

[3] J. White et al., "A prompt pattern catalog to enhance prompt engineering with ChatGPT," in PLoP '23, Article 5, pp. 1–31, 2023.

[4] L. Reynolds and K. McDonell, "Prompt programming for large language models: beyond the few-shot paradigm," in CHI EA '21, Article 314, pp. 1–7, 2021.

[5] T. Brown et al., "Language models are few-shot learners," in NIPS'20, Article 159, pp. 1877–1901, 2020.

[6] J. Wei et al., "Chain-of-thought prompting elicits reasoning in large language models," in NIPS'22, Article 1800, pp. 24824–24837, 2022.

[7] P. Lewis et al., "Retrieval-augmented generation for knowledge-intensive NLP tasks," in NIPS'20, Article 793, pp. 9459–9474, 2020.

[8] Y. Gao et al., "Retrieval-augmented generation for large language models: a survey," arXiv:2312.10997, 2023.

[9] D. Edge et al., "From local to global: a Graph RAG approach to query-focused summarization," arXiv:2404.16130, 2024.

[10] Google Gemini Team, "Gemini 1.5: unlocking multimodal understanding across millions of tokens of context," arXiv:2403.05530, 2024.

[11] S. Krishna et al., "Fact, fetch, and reason: a unified evaluation of retrieval-augmented generation," in Proc. NAACL-HLT, vol. 1, 2025.

[12] Q. Wu et al., "AutoGen: enabling next-gen LLM applications via multi-agent conversation," in COLM, Philadelphia, 2024.

[13] T. Guo et al., "Large language model based multi-agents: a survey of progress and challenges," in IJCAI '24, Article 890, pp. 8048–8057, 2024.

[14] Z. Xi et al., "The rise and potential of large language model based agents: a survey," Sci. China Inf. Sci., vol. 68, 121101, 2025.

[15] A. M. Dakhel et al., "GitHub Copilot AI pair programmer: asset or liability?" J. Syst. Softw., vol. 203, 2023.

[16] V. Viswanadhapalli, "AI-augmented software development: enhancing code quality and developer productivity using large language models," IJNRD, vol. 9, no. 8, pp. e382–e396, 2024.

[17] N. S. Shashidhara, "AI in software engineering - how intelligent systems are changing the software development process," EJCSIT, vol. 13, no. 29, pp. 28–39, 2025.

[18] A. Gu et al., "Challenges and paths towards AI for software engineering," arXiv:2503.22625, 2025.

[19] B. Gutierrez et al., "From RAG to memory: non-parametric continual learning for large language model," in ICML, PMLR 267, pp. 21497–21515, 2025.

[20] R. W. Hamming, Introduction to Applied Numerical Analysis. New York: Dover, 1989.

[21] Interesting Engineering, "What's the biggest software package by lines of code?" 2021. [Online]. Available: https://interestingengineering.com/lists/whats-the-biggest-software-package-by-lines-of-code

[22] FAISS Documentation. [Online]. Available: https://faiss.ai/index.html

[23] Neo4j Documentation. [Online]. Available: https://neo4j.com/docs/

[24] The Cambridge Handbook of Artificial Intelligence. Cambridge: Cambridge University Press, 2014.

[25] RAGAS Documentation. [Online]. Available: https://docs.ragas.io/en/stable/

[26] European Parliament, "The impact of the General Data Protection Regulation (GDPR) on artificial intelligence," 2020.

[27] Regulation (EU) 2024/1689 (Artificial Intelligence Act), 2024. [Online]. Available: https://eur-lex.europa.eu/eli/reg/2024/1689/oj/eng

[28] E. Alor et al., "Evaluating the use of LLMs for documentation to code traceability," arXiv:2506.16440, 2025.

[29] G. Flouris et al., "Ontology change: classification and survey," The Knowledge Engineering Review, vol. 23, no. 2, Cambridge University Press, 2008.

[30] B. Kitchenham, "Procedures for performing systematic reviews," Joint Technical Report, Software Engineering Group, Dept. of Computer Science, Keele University, 2004.