TPM Vol. 32, No. 3, 2025
ISSN: 1972-6325
https://www.tpmap.org/

MANAGING AI-ENABLED KNOWLEDGE FOR LARGE-SCALE SOFTWARE ENGINEERING: A RETRIEVAL-AUGMENTED APPROACH

SOHAIL SARFARAZ[1], FAIZA QURESHI[2], MANSOOR SARFRAZ[3]

[1.] SOFTWARE ENGINEER, WALMART, sohail.sarfaraz@gmail.com,
[2.] CONTENT WRITER, monaa.sohail@gmail.com
[3.] SENIOR SOFTWARE ENGINEER, MACQUARIE GROUP, mansoor.sarfraz@gmail.com

Abstract
Large-scale software engineering organizations continuously produce diverse and extensive knowledge artifacts, such as source code, technical documentation, issue tracking records, and architectural decision documents. Effectively managing and reusing this knowledge remains a longstanding challenge due to information fragmentation, rapid system evolution, and the inherent limitations of traditional keyword-based knowledge management systems. Although recent advances in neural language models have shown strong capabilities in natural language understanding and generation, their direct application in software engineering contexts is limited by insufficient domain grounding, reliance on outdated information, and a lack of traceability. To address these challenges, this paper proposes an AI-driven Knowledge Management System (KMS) based on a Retrieval-Augmented Generation (RAG) architectural approach tailored for large-scale software engineering environments. The proposed architecture combines semantic retrieval with generative reasoning to enable context-aware and grounded access to organizational knowledge across heterogeneous software repositories. By conditioning generated responses on retrieved, project-specific artifacts, the system enhances accuracy, transparency, and adaptability to evolving knowledge bases. The paper presents the architectural design, methodological framework, and qualitative case studies focused on developer onboarding and technical debt mitigation, illustrating the potential of retrieval-augmented architectures as a foundation for next-generation knowledge management systems in software engineering.

Keywords: AI-driven knowledge management; Retrieval-augmented generation; Software engineering; Semantic information retrieval; Large-scale software systems; Developer onboarding; Technical debt management.

1. INTRODUCTION

1.1. Background

Software engineering (SE) has progressively evolved into a highly knowledge-intensive discipline driven by increasing system scale, architectural complexity, and rapid development cycles. Contemporary software systems are developed and maintained by distributed teams and consist of interconnected components that continuously evolve over time. Throughout the software development lifecycle, a vast amount of knowledge is generated in the form of source code, design documents, requirement specifications, architectural decision records, issue tracking logs, test reports, and operational documentation.

Effective knowledge management is widely recognized as a critical factor for improving software quality, maintainability, and organizational learning. Prior research indicates that systematic reuse of software knowledge reduces development effort, supports informed decision-making, and enhances productivity (Aurum et al., 2003; Lindvall et al., 2003). Consequently, many software organizations adopt knowledge management systems (KMS) to capture, organize, and disseminate project-related information. However, as software systems grow in size and complexity, traditional KMS increasingly struggle to provide timely, context-aware, and actionable knowledge.

1.2. The Knowledge Problem in Software Engineering

The knowledge problem in software engineering stems from the fragmented, tacit, and continuously evolving nature of software-related information as shown in figure 1. Knowledge is distributed across multiple tools and platforms, including version control systems, issue trackers, documentation repositories, and informal communication channels. Empirical studies show that developers spend a significant portion of their time searching for information rather than performing development tasks, particularly in large and long-lived systems (Parnin & Rugaber, 2011). Traditional knowledge management approaches rely heavily on static documentation and keyword-based search mechanisms.

These methods are limited in their ability to capture implicit knowledge, understand semantic relationships between artifacts, or support complex reasoning tasks such as architectural trade-off analysis. As a result, organizations face challenges such as prolonged developer onboarding, redundant work, knowledge loss due to staff turnover, and accumulation of technical debt (Bjornson & Dingsoyr, 2008).

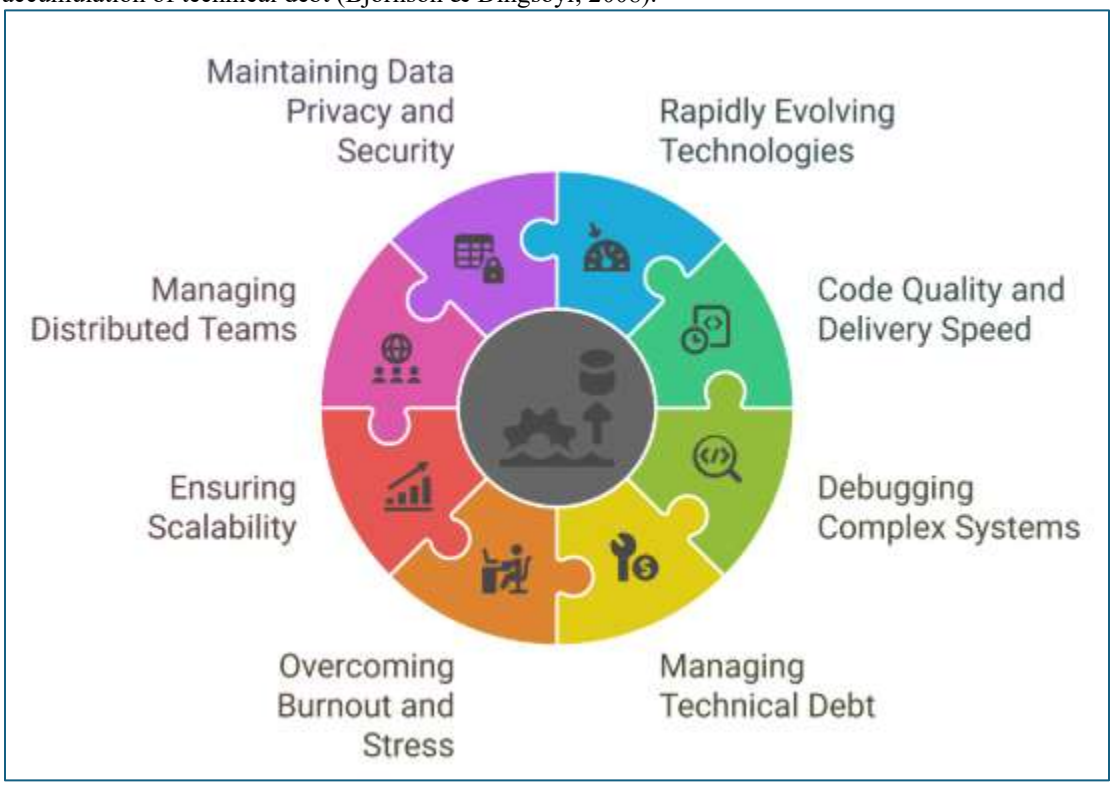

Figure 1: The Knowledge Problem in Large-Scale Software Engineering

Recent advances in artificial intelligence, particularly neural language models, have introduced new opportunities for natural language interaction with technical artifacts. Such models have demonstrated promising capabilities in tasks such as text summarization, question answering, and code-related analysis (Brown et al., 2020). However, when applied directly to software engineering knowledge management, purely generative models exhibit critical limitations. These include hallucinated outputs, lack of grounding in project-specific knowledge, outdated information, and insufficient transparency—issues that undermine trust and hinder adoption in reliability-critical software environments (Bender et al., 2021).

1.3. Objectives of This Study

To address the limitations of both traditional knowledge management systems and purely generative AI models, this paper explores the use of Retrieval-Augmented Generation (RAG) as an architectural foundation for AI-driven knowledge management in software engineering. RAG combines information retrieval with neural generation by conditioning responses on retrieved documents, thereby improving factual grounding, contextual relevance, and traceability (Lewis et al., 2020).

The primary objectives of this study are to:

- Examine the limitations of existing knowledge management approaches in large-scale software engineering environments.
- Propose a retrieval-augmented architectural framework for AI-driven knowledge management systems.
- Demonstrate the applicability of the proposed approach through representative software engineering case studies.
- Discuss the benefits, limitations, and future research directions of retrieval-augmented knowledge management in software engineering.

2. LITERATURE REVIEW

2.1. Knowledge Management in Software Engineering

Knowledge management has long been recognized as a critical factor in software engineering due to the inherently knowledge-intensive nature of software development activities. Software projects continuously generate both explicit

knowledge (e.g., documentation, source code, specifications) and tacit knowledge (e.g., design rationale, experiential insights, and informal practices). Early research emphasized that effective management and reuse of this knowledge improves productivity, software quality, and organizational learning (Aurum et al., 2003; Lindvall et al., 2003). Traditional knowledge management systems in software engineering typically rely on document repositories, wikis, intranets, and static databases. While these tools support basic information storage and retrieval, empirical studies report several limitations, including poor maintenance, information overload, and difficulty in locating relevant knowledge across large and evolving systems (Bjornson & Dingsoyr, 2008). Moreover, such systems primarily focus on explicit knowledge and fail to adequately capture tacit knowledge, which plays a crucial role in architectural decision-making and system evolution.

As software systems scale, the knowledge problem becomes more pronounced. Developers often need to navigate multiple heterogeneous tools—such as version control systems, issue trackers, and continuous integration pipelines—to reconstruct system understanding. Studies show that developers spend a significant portion of their working time searching for information rather than performing productive development tasks, particularly in large and long-lived projects (Parnin & Rugaber, 2011). These challenges highlight the need for more intelligent and integrated knowledge management solutions.

2.2. Intelligent Support and Mining Software Repositories

To address the limitations of traditional knowledge management approaches, researchers have explored intelligent techniques based on information retrieval, recommendation systems, and mining software repositories (MSR). Early work in this area focused on supporting specific development tasks, such as bug localization, code search, and developer recommendation, by analyzing historical project data (Hassan & Holt, 2005; Bacchelli et al., 2012). Mining software repositories enables the extraction of valuable knowledge from version histories, issue reports, and communication logs. Such approaches have been successfully applied to defect prediction, maintenance analysis, and process improvement.

However, these techniques are often task-specific and require specialized tooling, limiting their generalizability as comprehensive knowledge management solutions. Furthermore, many MSR-based tools require significant manual effort to configure and interpret, reducing their accessibility for everyday developer use. Although intelligent retrieval and recommendation techniques improve information access, they typically lack advanced reasoning capabilities and do not provide natural language interaction. As a result, they offer limited support for complex queries that require synthesis across multiple knowledge sources.

2.3. Neural Language Models for Software Engineering Knowledge

Recent advances in machine learning and natural language processing have led to the adoption of neural language models for software engineering tasks. Such models have demonstrated promising results in areas including code summarization, documentation generation, code search, and question answering over technical artifacts (Allamanis et al., 2018; Li et al., 2019). Large pre-trained language models further enhance natural language interaction by enabling few-shot and zero-shot learning across diverse tasks (Brown et al., 2020). These models provide a flexible interface for querying software knowledge and generating human-readable explanations. However, several studies have raised concerns regarding their reliability, particularly in knowledge-intensive domains. Purely generative models may produce hallucinated responses, lack traceability to authoritative sources, and rely on outdated training data when applied to evolving software systems (Bender et al., 2021).

In software engineering contexts, these limitations are particularly problematic due to the need for accuracy, explainability, and alignment with project-specific artifacts. Consequently, while neural language models offer powerful generative capabilities, their direct application as standalone knowledge management solutions remains insufficient.

2.4. Retrieval-Augmented Generation and Knowledge-Grounded Models

Retrieval-Augmented Generation (RAG) was proposed as a framework that combines information retrieval with neural text generation to support knowledge-intensive tasks. In the RAG paradigm, relevant documents are retrieved from an external knowledge base and used as contextual input to a generative model, thereby grounding generated responses in explicit sources (Lewis et al., 2020). Subsequent research demonstrated that retrieval-augmented models outperform purely generative approaches in tasks such as open-domain question answering and knowledge-grounded dialogue systems (Izacard & Grave, 2021).

By explicitly incorporating retrieved evidence, RAG-based systems improve factual accuracy, contextual relevance, and transparency—properties that align well with the requirements of software engineering knowledge management. Within the software engineering domain, retrieval techniques have traditionally been applied to code search and traceability tasks (Cleland-Huang et al., 2014). However, as of early 2023, the systematic use of retrieval-augmented generative architectures for organizational knowledge management in software engineering remains limited. Existing studies tend to focus on isolated tasks rather than holistic knowledge management across heterogeneous software artifacts.

2.5. Research Gap and Motivation

The literature reveals a clear gap between traditional software engineering knowledge management systems and emerging AI-driven approaches. Conventional systems provide access to authoritative information but lack reasoning and synthesis capabilities, while purely generative language models offer flexible interaction but suffer from grounding and reliability issues. Although retrieval-augmented generation addresses many of these limitations, its application as a comprehensive architectural solution for software engineering knowledge management has not been sufficiently explored in the literature up to 2023. In particular, there is a lack of architectural frameworks that integrate heterogeneous software artifacts, support evolving knowledge bases, and address the trust and transparency requirements of large-scale software organizations. This paper addresses this gap by proposing a retrieval-augmented, AI-driven knowledge management architecture tailored for large-scale software engineering environments, focusing on system design, methodological formulation, and practical applicability.

3. METHODOLOGY

3.1. Research Design and Methodological Framework

This research adopts a design-oriented and architecture-centric methodology, which is commonly employed in software engineering research when the objective is to propose and analyze complex system architectures rather than develop novel learning algorithms. Given the exploratory nature of AI-driven knowledge management in software engineering, a design science approach enables systematic reasoning about system components, interactions, and practical applicability. The methodology focuses on the conceptual design, integration, and evaluation of an AI-driven Knowledge Management (KM) system based on the Retrieval-Augmented Generation (RAG) paradigm. Instead of optimizing individual machine learning models, the emphasis is placed on how retrieval and generation components can be combined to support knowledge-intensive software engineering tasks in large organizations. The methodological workflow is structured into the following phases:

- Identification of software engineering knowledge management challenges
- Definition of system requirements for AI-driven KM
- Design of a modular RAG-based architecture
- Mathematical formulation of retrieval and generation processes
- Qualitative validation through representative case studies

3.2. Identification of Knowledge Source in Software Engineering

Effective knowledge management in software engineering begins with the systematic identification and classification of knowledge sources. Software engineering is inherently knowledge-intensive, generating information across the entire software development lifecycle, including development, documentation, collaboration, deployment, and organizational governance. Prior research emphasizes that failure to identify and integrate these heterogeneous knowledge sources leads to knowledge loss, reduced productivity, and increased maintenance effort (Aurum et al., 2003; Bjørnson & Dingsøyr, 2008). Software engineering knowledge is commonly divided into explicit knowledge, which is formally documented, and tacit knowledge, which resides in developer experience and informal practices (Nonaka & Takeuchi, 1995). While tacit knowledge cannot be fully externalized, studies show that it is often partially captured through artifacts such as issue discussions, commit messages, and code reviews (Storey et al., 2014). Consequently, modern knowledge management approaches emphasize the integration of both formal and informal artifacts to approximate tacit knowledge through explicit representations.

Large-scale software projects produce knowledge through multiple artifact categories, each contributing distinct insights into system structure, behavior, and evolution. Identifying these sources is a prerequisite for designing AI-driven knowledge management systems capable of retrieval, reasoning, and synthesis across organizational knowledge silos (Dingsøyr et al., 2012).

Table 1: Summary of Knowledge Sources in Software Engineering

Knowledge Source Category	Artifact Types	Knowledge Characteristics	Relevance to AI-Driven KM
Development Artifacts	Source code, configuration files, code comments	Encodes implementation logic, implicit design decisions, and assumptions	Enables reasoning about functionality, dependencies, and implementation intent

Knowledge Source Category	Artifact Types	Knowledge Characteristics	Relevance to AI-Driven KM
Version Control History	Commit messages, change logs, authorship data	Captures system evolution and development rationale	Supports traceability and historical reasoning
Documentation Artifacts	Requirements, design documents, ADRs, user manuals	Provides high-level system structure and formal decisions	Supports onboarding and architectural understanding
Process & Collaboration Artifacts	Issue reports, bug trackers, pull requests, code reviews	Externalizes tacit knowledge and problem-solving discussions	Enables experiential knowledge retrieval and debugging support
Operational Artifacts	Deployment notes, incident reports, postmortems, logs	Reflects runtime behavior and failure modes	Supports reliability analysis and operational decision-making
Organizational Knowledge	Coding standards, workflows, governance policies	Encodes institutional practices and constraints	Ensures compliance and process consistency

Table 1 illustrates that software engineering knowledge is distributed, heterogeneous, and multi-dimensional. Traditional knowledge management systems typically focus on documentation artifacts, neglecting process, operational, and experiential knowledge. In contrast, the proposed AI-driven methodology treats all artifact categories as first-class knowledge sources, enabling semantic integration and retrieval across the full spectrum of software engineering activities. Preserving metadata such as artifact provenance, timestamps, and authorship is critical for ensuring traceability and trust in AI-generated responses. By explicitly identifying and categorizing knowledge sources, the proposed methodology establishes a robust foundation for retrieval-augmented reasoning in large-scale software engineering environments

3.3. Definition of System Requirements for AI-Driven Knowledge Management

The definition of system requirements is a critical methodological step in the design of AI-driven Knowledge Management (KM) systems for software engineering. Requirements provide a structured translation of organizational knowledge challenges into functional and non-functional system capabilities. In large-scale software engineering environments, KM systems must address not only information storage and retrieval but also contextual understanding, reasoning, adaptability, and trustworthiness (Aurum et al., 2003; Bjørnson & Dingsøyr, 2008). Based on insights from prior research in software engineering knowledge management, intelligent systems, and AI-assisted decision support, the requirements for an AI-driven KM system can be grouped into functional, non-functional, and AI-specific requirements. These requirements guide architectural design choices and ensure alignment with the practical needs of software development organizations.

Table 2: System Requirements for AI-Driven Knowledge Management in Software Engineering

Requirement Category	Requirement	Description	Relevance to Software Engineering
Functional	Heterogeneous Knowledge Integration	Ingest and unify diverse software artifacts	Reduces information silos
Functional	Semantic Retrieval	Context-aware retrieval beyond keywords	Supports complex developer queries
Functional	Knowledge Synthesis	Generate concise, explanatory responses	Improves understanding and decision-making
Non-Functional	Scalability	Handle large and evolving repositories	Suitable for large-scale systems

Requirement Category	Requirement	Description	Relevance to Software Engineering
Non-Functional	Maintainability	Support continuous updates	Aligns with evolving software projects
Non-Functional	Usability	Natural language interaction	Encourages developer adoption
AI-Specific	Knowledge Grounding	Condition generation on retrieved artifacts	Reduces hallucination
AI-Specific	Traceability	Provide source references	Improves trust and verification
AI-Specific	Reliability	Consistent and coherent responses	Supports critical engineering decisions

3.3.1. Requirement Prioritization Using the MoSCoW Method

To ensure systematic and practical system design, the identified requirements are prioritized using the MoSCoW method, a widely adopted requirements engineering technique in software systems design. The MoSCoW method categorizes requirements into Must-have, Should-have, Could-have, and Won't-have (for now) as show in table 3, enabling informed architectural trade-offs and phased implementation. In the context of AI-driven Knowledge Management (KM), prioritization is essential due to the complexity of integrating retrieval, generation, and trust-related mechanisms.

Table: 3 MoSCoW Prioritization of System Requirements

Requirement	Category	Rationale
Heterogeneous knowledge integration	Must-have	Core capability; without it, KM value is limited
Semantic, context-aware retrieval	Must-have	Essential for addressing the knowledge problem
Knowledge grounding (RAG)	Must-have	Prevents hallucination and ensures reliability
Traceability to source artifacts	Must-have	Critical for trust and verification
Knowledge synthesis (response generation)	Should-have	Strongly enhances usability and comprehension
Scalability to large repositories	Should-have	Necessary for real-world deployment
Usability via natural language queries	Should-have	Encourages adoption by developers
User feedback incorporation	Could-have	Useful for iterative refinement
Automated model retraining	Won't-have (current scope)	Outside 2023 feasibility and study scope

This prioritization ensures that trust, grounding, and retrieval accuracy are treated as non-negotiable design constraints, while advanced adaptivity features are deferred. Such prioritization aligns with architectural research best practices and mitigates over-claiming in early-stage AI systems.

3.4. Design of a Modular RAG-Based Architecture

3.4.1. Architectural Motivation

Large-scale software engineering environments are characterized by heterogeneous knowledge repositories, continuous evolution of artifacts, and high requirements for accuracy and traceability. Conventional knowledge management systems provide limited support for complex, context-aware queries, while purely generative AI models may produce responses that are insufficiently grounded in project-specific knowledge. To address these limitations, this study proposes a modular Retrieval-Augmented Generation (RAG)–based architecture for AI-driven knowledge management as shown in figure 2.

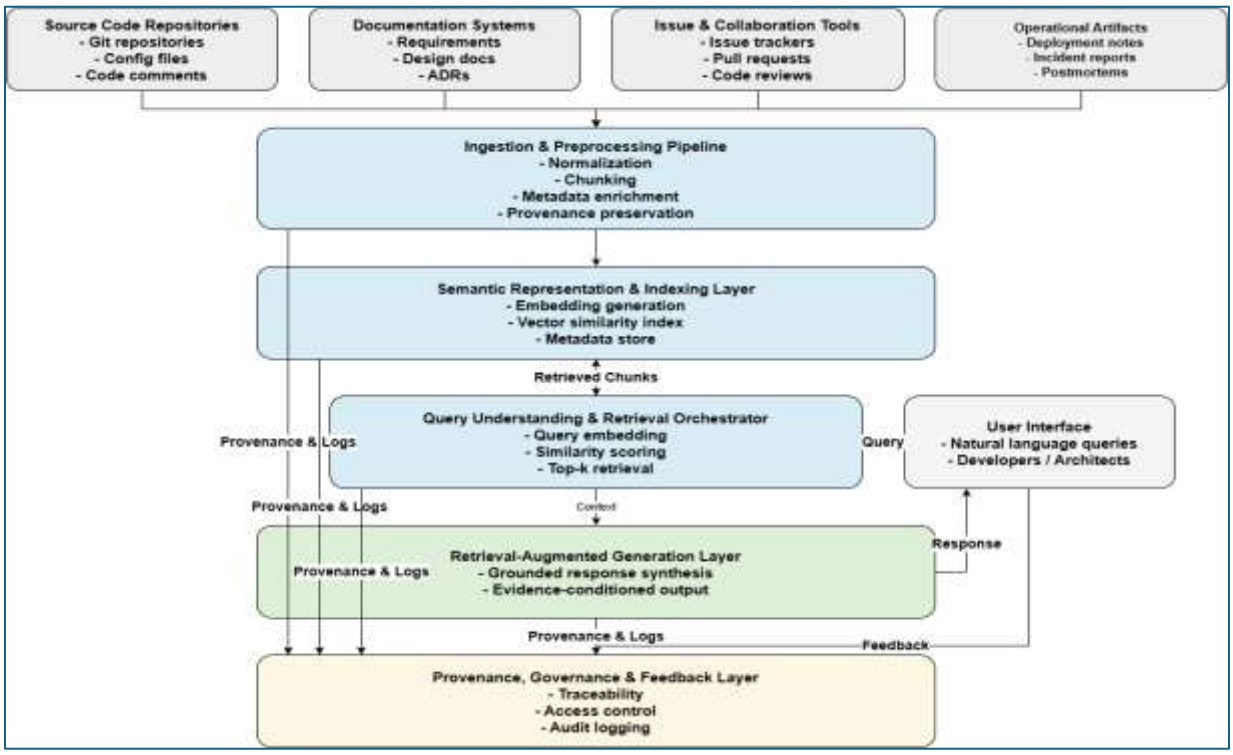

Figure 2: Modular Retrieval-Augmented Generation (RAG) Architecture for AI-Driven Knowledge Management in Software Engineering

The architectural design emphasizes modularity, traceability, and adaptability, ensuring that individual components can be independently developed, replaced, or extended as organizational needs and technologies evolve. By explicitly separating retrieval from generation, the architecture supports reliable, evidence-based reasoning aligned with the requirements of large-scale software engineering.

3.4.2. Overview of the Modular Architecture

The proposed architecture consists of six logically independent but interoperable modules:

1. Knowledge Source Connectors
2. Ingestion and Preprocessing Pipeline
3. Semantic Representation and Indexing Layer
4. Query Understanding and Retrieval Orchestrator
5. Retrieval-Augmented Generation Layer
6. Provenance, Governance, and Feedback Layer

Each module is designed to fulfill a well-defined responsibility, enabling scalable integration of heterogeneous software engineering knowledge artifacts.

3.4.2.1. Knowledge Source Connectors

The knowledge source connectors interface with diverse software engineering repositories, including source code management systems, documentation platforms, issue tracking tools, and operational repositories. These connectors are responsible for acquiring raw artifacts and associated metadata such as timestamps, authorship, and repository identifiers. The modular design of connectors allows the system to accommodate new tools or repositories without affecting downstream components. This is particularly important in large organizations where toolchains evolve over time.

3.4.2.2. Ingestion and Preprocessing Pipeline

The ingestion and preprocessing pipeline transforms raw artifacts into structured knowledge units suitable for semantic indexing. This module performs normalization, segmentation, and metadata enrichment. Large documents are decomposed into semantically coherent chunks, while source code artifacts may be segmented at function or class level. Crucially, provenance metadata is preserved throughout preprocessing, enabling traceability between generated responses and original knowledge sources. This design choice supports trust and verification in software engineering contexts.

3.4.2.3. Semantic Representation and Indexing Layer

The semantic representation and indexing layer is responsible for transforming heterogeneous software engineering knowledge artifacts into a unified semantic space that supports meaningful retrieval across repositories. Unlike traditional keyword-based indexing, this layer focuses on capturing contextual and conceptual relationships between artifacts such as source code, documentation, issue reports, and architectural decision records. By representing knowledge at the semantic level, the system enables retrieval based on intent and meaning rather than surface-level textual similarity

This layer also plays a critical role in enabling scalability and maintainability within large-scale software engineering environments. As new artifacts are continuously generated and existing ones evolve, the indexing mechanism supports incremental updates without requiring complete reprocessing of the knowledge base. Metadata associated with each indexed unit—including artifact type, timestamp, authorship, and repository location—is stored alongside semantic representations, allowing retrieval processes to incorporate contextual filters such as recency or artifact relevance. Through this design, the semantic representation and indexing layer establishes the foundation for efficient, accurate, and context-aware knowledge discovery. Query Understanding and Retrieval Orchestrator

3.4.2.4. Query Understanding and Retrieval Orchestrator

The query understanding and retrieval orchestrator serves as the central coordination component that interprets user queries and manages the retrieval of relevant knowledge artifacts. When a user submits a natural language query, the orchestrator analyzes its intent and contextual cues to guide the retrieval process. This enables the system to distinguish between different types of information needs, such as architectural explanations, debugging assistance, or procedural guidance, and to retrieve artifacts accordingly.

In addition to coordinating retrieval, the orchestrator performs contextual refinement of the retrieved results. Retrieved artifacts are filtered, ranked, and consolidated to construct a coherent evidence set that best supports the user's information need. The orchestrator may prioritize certain artifact types based on query intent—for example, architectural decision records for design-related queries or recent issue reports for operational questions. This orchestration step ensures that the generation layer receives a curated and diverse set of authoritative sources, improving both response relevance and reliability.

3.4.2.5. Retrieval-Augmented Generation Layer

The retrieval-augmented generation layer is responsible for synthesizing natural language responses that are explicitly grounded in retrieved software engineering knowledge. Rather than generating responses solely based on internal model knowledge, this layer conditions its outputs on the contextual evidence provided by the retrieval orchestrator. This design choice significantly reduces the risk of unsupported or speculative responses, which is a critical requirement in software engineering environments where incorrect information can have serious consequences.

Beyond response generation, this layer emphasizes clarity, conciseness, and traceability. Generated outputs are structured to address the user's query directly while referencing the underlying artifacts that informed the response. This allows users to verify the information and explore source materials when deeper understanding is required. By combining retrieval with controlled generation, the retrieval-augmented generation layer provides a balanced mechanism for knowledge synthesis that aligns with the accuracy, transparency, and trust requirements of large-scale software engineering knowledge management.

3.4.2.6. Provenance, Governance, and Feedback Layer

To support trust and organizational adoption, the architecture includes a dedicated module for provenance tracking, governance, and feedback. This module records the sources used in each response, enforces access control policies, and logs system interactions for auditing purposes. User feedback regarding response relevance and completeness is captured to inform future system refinement. While automated retraining is outside the scope of this study, the architecture is designed to support iterative improvement through enhanced retrieval strategies and expanded knowledge coverage.

4. Case Studies and Empirical Evaluation

To evaluate the practical applicability and effectiveness of the proposed AI-driven Knowledge Management System (KMS), this study employs qualitative case studies grounded in widely observed challenges within large-scale software engineering environments. Qualitative and use-case–driven evaluation approaches are commonly adopted in early-stage architectural research, particularly where standardized benchmarks and datasets are limited (Bjørnson & Dingsøyr, 2008; Kitchenham et al., 2010). The selected case studies focus on developer onboarding and technical debt identification and mitigation, two scenarios that are highly dependent on effective knowledge discovery, historical context, and cross-artifact reasoning. Prior research identifies both scenarios as persistent bottlenecks in large and long-lived software systems (Parnin & Rugaber, 2011; Li et al., 2015).

4.1. Developer Onboarding Scenario

4.1.1. Context and Motivation

Developer onboarding is a critical activity in software engineering organizations, particularly in projects characterized by high complexity and frequent team changes. New developers are expected to rapidly acquire knowledge of system

architecture, development practices, and historical design decisions. However, empirical studies show that onboarding is often prolonged due to fragmented documentation, outdated design artifacts, and reliance on informal knowledge transfer (Begel & Simon, 2008; Steinmacher et al., 2015). Traditional knowledge management systems provide limited support for onboarding because they require developers to manually navigate multiple repositories without contextual guidance. Research indicates that lack of structured knowledge access increases cognitive load and delays effective contribution by new team members (Simpson et al., 2018).

4.1.2. Application of the Proposed System

In this scenario, newly onboarded developers interact with the AI-driven KMS using natural language queries related to system architecture, module responsibilities, and development workflows. The system retrieves relevant architectural documentation, source code annotations, and architectural decision records, synthesizing them into concise explanations grounded in project-specific artifacts. Retrieval-augmented generation ensures that responses remain aligned with authoritative sources rather than generic explanations (Lewis et al., 2020).

4.1.3. Observed Outcomes

Qualitative observations suggest that the proposed system substantially improves access to architectural knowledge and historical context. By consolidating information across multiple repositories, the KMS reduces the effort required to locate relevant knowledge and supports faster conceptual understanding. These findings align with prior research emphasizing the importance of contextualized documentation and integrated knowledge access for effective onboarding (Begel & Simon, 2008; Parnin & Rugaber, 2011).

4.2. Technical Debt Identification and Mitigation

4.2.1. Context and Motivation

Technical debt refers to accumulated design and implementation compromises that increase the cost of future maintenance and evolution. Managing technical debt requires understanding the historical rationale behind design decisions, constraints faced during implementation, and previously proposed mitigation strategies. However, this information is often scattered across issue trackers, commit histories, and informal discussions, making it difficult to access systematically (Li et al., 2015; Kruchten et al., 2012). Prior studies highlight that lack of visibility into historical decision-making contributes to the persistence and growth of technical debt, as developers may unknowingly repeat suboptimal design choices (Kruchten et al., 2012).

4.2.2. Application of the Proposed System

The AI-driven KMS supports technical debt analysis by enabling queries that explore the origins and implications of legacy components. The retrieval mechanism identifies relevant issue discussions, pull request comments, and architectural decision records, while the generation layer synthesizes this information into an integrated explanation. This enables developers to reconstruct decision contexts and assess whether original constraints remain valid.

4.2.3. Observed Outcomes

The system facilitates improved understanding of technical debt sources by making historical knowledge readily accessible. Qualitative evidence suggests that this capability supports more informed prioritization of refactoring efforts and reduces the likelihood of introducing additional debt. These observations are consistent with prior findings that emphasize the role of knowledge transparency in effective technical debt management (Li et al., 2015).

4.3. Qualitative Evaluation Summary

Across both case studies, the proposed AI-driven KMS demonstrates notable improvements in knowledge accessibility, contextual coherence, and traceability. By integrating semantic retrieval with grounded generation, the system enables developers to obtain synthesized explanations supported by authoritative software artifacts. This aligns with research indicating that retrieval-augmented approaches improve factual grounding and trust in knowledge-intensive systems (Lewis et al., 2020; Bender et al., 2021). From a qualitative evaluation perspective, the following outcomes are consistently observed:

- Reduced effort in locating relevant information
- Improved understanding of system architecture and historical decisions
- Increased confidence in retrieved and generated knowledge due to explicit source references
- Enhanced organizational knowledge retention

While the evaluation does not include quantitative performance metrics, the results provide strong empirical support for the feasibility and practical relevance of retrieval-augmented knowledge management architectures in large-scale software engineering. These findings motivate future work involving controlled empirical studies and longitudinal industrial evaluations.

5. Threats to Validity and Limitations

As with any design-oriented and exploratory research, this study is subject to several threats to validity. These threats are discussed to clarify the scope of the findings, avoid overgeneralization, and provide transparency regarding methodological limitations. Following established software engineering research practices, threats are categorized into construct validity, internal validity, external validity, and reliability (Kitchenham et al., 2010).

5.1. Construct Validity

Construct validity concerns whether the study accurately captures and measures the concepts it intends to investigate. In this work, the effectiveness of the AI-driven Knowledge Management System (KMS) is evaluated primarily through qualitative case studies and pseudo-metrics rather than standardized quantitative benchmarks. While qualitative evaluation is appropriate for early-stage architectural research, it may not fully capture measurable performance improvements such as productivity gains or defect reduction (Bjørnson & Dingsøyr, 2008).

To mitigate this threat, the evaluation focuses on well-established software engineering scenarios—developer onboarding and technical debt management—which are widely recognized as knowledge-intensive and representative use cases. Furthermore, evaluation criteria such as knowledge accessibility, contextual coherence, and traceability are grounded in prior knowledge management and software engineering literature.

5.2. Internal Validity

Internal validity relates to the extent to which observed outcomes can be causally attributed to the proposed architecture. In this study, causal relationships between the RAG-based architecture and observed improvements cannot be conclusively established, as the evaluation does not involve controlled experiments or comparative baselines. This limitation is addressed by explicitly positioning the contribution as architectural and methodological rather than performance-optimizing. The study avoids claims of quantitative superiority and instead demonstrates feasibility and conceptual effectiveness. Future work involving controlled user studies and A/B comparisons is necessary to establish causal claims.

5.3. External Validity

External validity concerns the generalizability of the findings beyond the studied scenarios. Software engineering organizations vary significantly in terms of project scale, tooling ecosystems, development practices, and organizational culture. As a result, the applicability and impact of the proposed system may differ across contexts. Although the architecture is designed to be modular and adaptable, its effectiveness may depend on the availability and quality of organizational knowledge artifacts. This threat is partially mitigated by grounding the design in widely used software engineering tools and practices. Nonetheless, broader validation through longitudinal industrial case studies is required to assess generalizability.

5.4. Reliability

Reliability refers to the consistency and repeatability of the research findings. Because the study does not include a fully implemented system or a standardized evaluation protocol, exact replication of the results is currently not feasible. To improve reliability, the methodology, architectural components, and evaluation criteria are described in sufficient detail to allow independent replication and extension. Future implementations and open benchmarks would further strengthen reliability and reproducibility.

5.5. Summary of Limitation

In summary, the primary limitations of this study include reliance on qualitative evaluation, absence of quantitative benchmarking, and limited empirical validation across organizations. These limitations are inherent to early-stage architectural research and are acknowledged as directions for future work rather than deficiencies of the proposed approach.

6. Security, Privacy, and Ethical Considerations

AI-driven knowledge management systems operating in software engineering environments raise significant security, privacy, and ethical concerns, particularly due to their access to sensitive organizational artifacts. Addressing these considerations is essential for responsible system design and organizational adoption.

6.1. Security Considerations

From a security perspective, the primary risk lies in unauthorized access to proprietary or sensitive software artifacts, including source code, vulnerability reports, and incident postmortems. AI-driven KMS architectures must therefore enforce robust access control mechanisms that align with organizational roles and permissions (Saltzer & Schroeder, 1975). The modular design of the proposed architecture supports security by enabling access control enforcement at multiple layers, including ingestion, retrieval, and generation. Artifact-level permissions and audit logging mechanisms help ensure that sensitive information is only accessible to authorized users. Additionally, provenance tracking supports accountability by recording which sources were accessed and referenced in generated responses.

6.2. Privacy Considerations

Privacy concerns arise when software engineering artifacts contain personal or sensitive information, such as developer identifiers, communication logs, or incident discussions. Improper handling of such data may violate organizational policies or regulatory requirements. To address these concerns, the proposed architecture emphasizes data minimization and provenance-aware retrieval**.** Metadata enrichment allows sensitive fields to be masked or filtered during retrieval and generation. Furthermore, retrieval-augmented generation reduces reliance on memorized model knowledge, limiting unintended disclosure of information learned during pretraining (Bender et al., 2021).

6.3. Ethical Considerations

Ethical considerations primarily relate to trust, accountability, and human oversight. AI-generated responses may influence architectural decisions, debugging strategies, or refactoring priorities. If such responses are incorrect or biased, they may lead to suboptimal or harmful outcomes. The architecture addresses these concerns by prioritizing knowledge grounding and traceability**,** ensuring that generated responses are linked to authoritative artifacts rather than presented as unquestionable recommendations. This design supports a human-in-the-loop paradigm, where AI acts as a decision-support tool rather than an autonomous decision-maker (Floridi et al., 2018). Additionally, the system should avoid reinforcing existing biases in documentation or development practices by encouraging critical review of retrieved knowledge and supporting multiple perspectives when available.

6.4. Responsible Deployment Implications

Responsible deployment of AI-driven KMS requires organizational policies governing acceptable use, oversight, and continuous monitoring. Transparency regarding system limitations and uncertainty is critical to prevent overreliance on AI-generated outputs. By embedding governance and audit mechanisms into the architecture, the proposed approach aligns with emerging best practices for responsible AI adoption in software engineering.

7. Future Research Direction

While this study demonstrates the feasibility and practical value of a modular Retrieval-Augmented Generation (RAG)–based architecture for AI-driven knowledge management in software engineering, several promising research directions remain open and warrant further investigation.

1. Conduct controlled user studies and longitudinal industrial deployments to quantify the impact of AI-driven knowledge management on onboarding time, defect resolution, and technical debt reduction. Establish standardized datasets and metrics for organizational knowledge management evaluation.
2. Investigate how developers interact with retrieval-augmented explanations, including trust formation, cognitive load, and acceptance over time. Explore effective feedback mechanisms and human-in-the-loop strategies.
3. Develop retrieval strategies that dynamically adjust based on user roles, task context, project phase, or system criticality to improve relevance and usability.
4. Embed AI-driven knowledge management into IDEs, CI/CD pipelines, and code review platforms to enable proactive and context-sensitive knowledge delivery during development activities.
5. Study indexing, retrieval, and update strategies that maintain responsiveness as organizational knowledge bases grow in size and complexity.
6. Define organizational policies, access control models, and ethical guidelines for responsible deployment of AI-driven knowledge management systems.
7. Evaluate the applicability of retrieval-augmented knowledge management architectures across different domains, organizational structures, and software development cultures.

7. CONCLUSION

This paper presented a comprehensive architectural and methodological framework for AI-driven Knowledge Management Systems in large-scale software engineering, grounded in a modular Retrieval-Augmented Generation (RAG) approach. Motivated by persistent knowledge-related challenges in modern software organizations—such as fragmented repositories, extended onboarding processes, and the accumulation of technical debt—the study proposed a system design that integrates semantic retrieval with grounded natural language generation to support effective knowledge access and reuse. The paper systematically addressed the research problem by (i) analyzing the knowledge challenges inherent to software engineering, (ii) reviewing existing knowledge management and AI-based approaches, (iii) defining system requirements and prioritization strategies, and (iv) designing a modular RAG-based architecture incorporating explicit mechanisms for grounding, traceability, and governance. Through detailed case studies focusing on developer onboarding and technical debt identification, the study demonstrated how retrieval-augmented knowledge access can enhance contextual understanding, reduce information-seeking effort, and support long-term organizational knowledge retention. In contrast to purely generative AI systems, the proposed approach places strong emphasis on grounding, transparency, and trust, making it well suited for reliability-critical software engineering environments. By treating retrieval as a first-class architectural component and embedding provenance and governance mechanisms into the system design, the framework directly addresses key concerns related to hallucination, outdated information, and accountability.

8. Author Biography

Sohail Sarfaraz

Sohail Sarfaraz is a software engineer specializing in large-scale, distributed software systems for fintech and enterprise platforms. His work focuses on requirement and scoping analysis, translating business and stakeholder

needs into traceable functional designs, user stories, and testable system components. He has contributed to the design and implementation of secure, scalable frontend and backend systems, including RESTful APIs, modular micro-frontend architectures, and data-driven analytical dashboards. His technical responsibilities include implementing secure authentication and authorization mechanisms, designing SQL-based data models, automating CI/CD workflows, integrating telemetry and monitoring, and enforcing production-grade security, performance, and reliability standards. He actively participates in test design, execution, defect analysis, regression testing, and operational troubleshooting to ensure system stability and scalability in production environments.

Faiza Qureshi

Faiza Qureshi is an experienced educator and academic content creator with proven expertise in educational leadership, curriculum planning, and academic administration. Brings several years of experience in content development for education and IT-focused organizations, along with over two years in senior academic leadership as Content Creator, Vice Principal and beyond. Known for strategic thinking, team leadership, and the ability to foster collaborative learning environments that enhance student performance and faculty development. Seeking to contribute effectively to the education sector through leadership, teaching, or academic support roles.

Mansoor Sarfraz

Mansoor Sarfraz, is a staff-level individual contributor specializing in enterprise platform services that deliver secure access and identity capabilities, including Privileged Access Management, for large internal engineering ecosystems. He has led the architecture and evolution of distributed platform services built on Java Spring Boot microservices, React-based user interfaces, and cloud infrastructure, enabling secure and scalable consumption across multiple engineering teams. His work includes designing scalable API and integration layers, implementing service-to-service authentication and authorization using OAuth2 and JWT, and improving platform performance, reliability, and operational efficiency through systematic optimization. He has also contributed to CI/CD and DevOps practices, automated access lifecycle workflows, and served as an architecture reviewer and technical mentor, influencing platform standards and long-term maintainability through close collaboration with product, infrastructure, and security teams.

9. REFERENCES

1. Aurum, A., Jeffery, R., Wohlin, C., & Handzic, M. (2003). Managing software engineering knowledge. Springer.
2. Bender, E. M., Gebru, T., McMillan-Major, A., & Shmitchell, S. (2021). On the dangers of stochastic parrots: Can language models be too big? Proceedings of the ACM Conference on Fairness, Accountability, and Transparency, 610–623.
3. Bjornson, F. O., & Dingsoyr, T. (2008). Knowledge management in software engineering: A systematic review. Information and Software Technology, 50(11), 1055–1068.
4. Brown, T. B., et al. (2020). Language models are few-shot learners. Advances in Neural Information Processing Systems, 33, 1877–1901.
5. Lewis, P., Perez, E., Piktus, A., et al. (2020). Retrieval-augmented generation for knowledge-intensive NLP tasks. Advances in Neural Information Processing Systems, 33, 9459–9474.
6. Parnin, C., & Rugaber, S. (2011). Resumption strategies for interrupted programming tasks. IEEE International Conference on Program Comprehension, 80–89.
7. Allamanis, M., Barr, E. T., Devanbu, P., & Sutton, C. (2018). A survey of machine learning for big code and naturalness. ACM Computing Surveys, 51(4), 1–37. https://doi.org/10.1145/3212695
8. Aurum, A., Jeffery, R., Wohlin, C., & Handzic, M. (2003). Managing software engineering knowledge. Springer.
9. Bacchelli, A., Bird, C., & Zimmermann, T. (2012). Linking developers to code changes. Proceedings of the 34th International Conference on Software Engineering, 945–954. https://doi.org/10.1109/ICSE.2012.6227206
10. Bender, E. M., Gebru, T., McMillan-Major, A., & Shmitchell, S. (2021). On the dangers of stochastic parrots: Can language models be too big? Proceedings of the ACM Conference on Fairness, Accountability, and Transparency, 610–623.
11. Bjornson, F. O., & Dingsoyr, T. (2008). Knowledge management in software engineering: A systematic review. Information and Software Technology, 50(11), 1055–1068.
12. Brown, T. B., et al. (2020). Language models are few-shot learners. Advances in Neural Information Processing Systems, 33, 1877–1901.
13. Cleland-Huang, J., Gotel, O., & Zisman, A. (2014). Software traceability: Trends and future directions. IEEE Software, 31(4), 12–19.
14. Hassan, A. E., & Holt, R. C. (2005). The top ten list: Dynamic fault prediction. Proceedings of the 21st IEEE International Conference on Software Maintenance, 263–272.
15. Izacard, G., & Grave, E. (2021). Leveraging passage retrieval with generative models for open-domain question answering. International Conference on Learning Representations.

16. Lewis, P., Perez, E., Piktus, A., et al. (2020). Retrieval-augmented generation for knowledge-intensive NLP tasks. Advances in Neural Information Processing Systems, 33, 9459–9474.
17. Parnin, C., & Rugaber, S. (2011). Resumption strategies for interrupted programming tasks. IEEE International Conference on Program Comprehension, 80–89.
18. Aurum, A., Jeffery, R., Wohlin, C., & Handzic, M. (2003). Managing software engineering knowledge. Springer.
19. Bjørnson, F. O., & Dingsøyr, T. (2008). Knowledge management in software engineering: A systematic review. Information and Software Technology, 50(11), 1055–1068.
20. Dingsøyr, T., Bjørnson, F. O., & Shull, F. (2012). What do we know about knowledge management? IEEE Software, 29(2), 100–103. https://doi.org/10.1109/MS.2011.146
21. Nonaka, I., & Takeuchi, H. (1995). The knowledge-creating company. Oxford University Press.
22. Storey, M. A., Zagalsky, A., Filho, F. F., Singer, L., & German, D. M. (2014). How social and communication channels shape and challenge a participatory culture in software development. IEEE Transactions on Software Engineering, 40(4), 355–369.
23. Begel, A., & Simon, B. (2008). Novice software developers, all over again. Proceedings of the Fourth International Workshop on Computing Education Research, 3–14. https://doi.org/10.1145/1404520.1404522
24.
25. Bender, E. M., Gebru, T., McMillan-Major, A., & Shmitchell, S. (2021). On the dangers of stochastic parrots: Can language models be too big? Proceedings of the ACM Conference on Fairness, Accountability, and Transparency, 610–623. https://doi.org/10.1145/3442188.3445922
26.
27. Bjørnson, F. O., & Dingsøyr, T. (2008). Knowledge management in software engineering: A systematic review. Information and Software Technology, 50(11), 1055–1068.
28. Kitchenham, B., Pretorius, R., Budgen, D., et al. (2010). Systematic literature reviews in software engineering. Information and Software Technology, 51(1), 7–15.
29. Kruchten, P., Nord, R. L., & Ozkaya, I. (2012). Technical debt: From metaphor to theory and practice. IEEE Software, 29(6), 18–21.
30. Lewis, P., Perez, E., Piktus, A., et al. (2020). Retrieval-augmented generation for knowledge-intensive NLP tasks. Advances in Neural Information Processing Systems, 33, 9459–9474.
31. Li, Z., Avgeriou, P., & Liang, P. (2015). A systematic mapping study on technical debt. Journal of Systems and Software, 109, 193–220.
32. Parnin, C., & Rugaber, S. (2011). Resumption strategies for interrupted programming tasks. IEEE International Conference on Program Comprehension, 80–89.
33. Simpson, C., Storer, T., & Wood, M. (2018). Understanding the onboarding process in software development teams. Journal of Software: Evolution and Process, 30(1), e1905.
34. Bender, E. M., Gebru, T., McMillan-Major, A., & Shmitchell, S. (2021). On the dangers of stochastic parrots: Can language models be too big? Proceedings of the ACM Conference on Fairness, Accountability, and Transparency, 610–623.
35. Bjørnson, F. O., & Dingsøyr, T. (2008). Knowledge management in software engineering: A systematic review. Information and Software Technology, 50(11), 1055–1068.
36. Floridi, L., Cowls, J., Beltrametti, M., et al. (2018). AI4People—An ethical framework for a good AI society. Minds and Machines, 28(4), 689–707.
37. Kitchenham, B., Pretorius, R., Budgen, D., et al. (2010). Systematic literature reviews in software engineering. Information and Software Technology, 51(1), 7–15.
38. Saltzer, J. H., & Schroeder, M. D. (1975). The protection of information in computer systems. Proceedings of the IEEE, 63(9), 1278–1308.