Architectural Blueprint and Implementation Strategy for a Subject Matter Expert Agent in Software Engineering

I. Strategic Overview and Software Engineering SME Definition

A. Agent Role and Scope Definition

The Subject Matter Expert (SME) Agent operates within the Engineering domain, specializing in Software Engineering (SE). The primary mandate for this agent is three-fold: providing technical problem-solving assistance, generating standardized technical documentation, and creating accessible learning materials for both students and teachers.[1] This scope necessitates a blend of rigorous technical accuracy and pedagogical clarity.

The defined role requires the SE SME Agent to deliver complex, high-value outputs beyond standard question-answering. Specific capabilities include solving intricate technical problems, autonomously generating comprehensive lab assignments, preparing final, structured reports (in formats such as PDF and DOCX), and producing technical presentations (PPT).[1]

A fundamental requirement for any professional system operating in this domain is adherence to established industry guidelines. Consequently, the agent's knowledge base and generation workflows must implicitly uphold recognized professional standards. This includes referencing authoritative documents such as the ACM/IEEE-CS Software Engineering Code of Ethics [2] and various standardized documentation practices, including those defined by the IEEE, such as

IEEE Std 828 for Software Configuration Management and IEEE Std 1012 for Software Verification and Validation.[3] Ensuring compliance with these standards validates the SME Agent's professional relevance and the quality of its output.

The comprehensive nature of the SE domain, spanning both complex technical tasks and educational support, introduces a latent requirement for **dual optimization targets** within the architecture. The system must optimize for functional correctness, precision, and efficiency when performing engineering functions (e.g., generating code or technical specifications). Simultaneously, it must optimize for clarity, simplicity, and accessibility when producing educational outputs. This duality dictates specific design decisions in data processing and, crucially, in the evaluation framework, where metrics must accommodate both technical and pedagogical outcomes.

B. High-Level System Architecture and Workflow Diagram

The system architecture is designed to be highly extensible and modular, fulfilling the project requirement for distinct components handling different stages of processing.[1]

The architecture comprises three principal, decoupled servers: the Main API Server, the Chat/Tool/Agent Server, and the Modular RAG Pipeline. The **Agent Core (or Agent/Brain)** resides within the Chat/Tool/Agent Server and functions as the central coordinator. This component is responsible for conversational planning, memory management, and intelligent routing.[5] It processes user requests, decomposes them into manageable subtasks, and dynamically selects the appropriate tools to execute those tasks.

The system relies heavily on the integration of external tools to fulfill complex requests.[1] Key tools mandated for the SE domain include the Knowledge Retrieval Tool (the RAG pipeline itself), the Document Generation Tool (for producing DOCX, PDF, or PPT files), and the Email Automation Tool (for delivery of generated materials).[1] The modularity ensures that the RAG pipeline operates as a service (retrieve-and-rerank), decoupled from the Agent's high-level reasoning engine, allowing for independent optimization of both retrieval performance and agentic planning logic. The design specifically anticipates complex workflows that necessitate chaining multiple tool calls to fulfill a single, comprehensive user request.[1]

II. Data Engineering: Corpus Acquisition and Preprocessing for SE Content

A. Document Collection Strategy and Heterogeneous Format Handling

A robust RAG system relies on an authoritative and well-organized corpus. For the Software Engineering SME Agent, the source documents must be collected in a root-level directory and support heterogeneous formats, including PDF technical specifications, DOCX reports, PPTX presentations, TXT files, and Markdown (MD) documentation.[1] This necessity for handling varied data formats requires a powerful document loading pipeline capable of automatic type detection and parsing.

The corpus acquisition strategy is guided by the principle of covering the "most important text books and other authoritative content without duplication".[1] For the SE domain, this includes foundational academic texts (e.g., on algorithm design, system architecture), official IEEE and ACM standards documents [2], technical manuals and documentation for widely used software frameworks (e.g., Django, React, Spring [6]), and well-commented source code files from relevant open-source repositories. Every document ingested must be associated with necessary metadata, such as subject, source, context, and timestamp, facilitating later filtering and relevance scoring during retrieval.[1]

B. Preprocessing & Advanced Chunking Strategies for SE Content

Prior to indexing, the data pipeline must perform preprocessing steps, including deduplication, tokenization, lowercasing, and removal of non-informative content.[1] However, the selection of the optimal chunking strategy is paramount for ensuring high-quality retrieval in the technical context of software engineering. The final decision must be justified in detail.[1]

Standard fixed-size or recursive character splitting, while effective for uniform text, risks splitting semantically continuous blocks, such as class definitions, full function bodies, or structured list items.[8] In SE, these logical units are critical. Therefore, the implementation prioritizes **Document-Based Chunking (Content-Aware splitting)**. This approach parses documents based on their intrinsic structure, splitting Markdown documents by headings, HTML by tags, and crucially, splitting programming code by functions or classes (e.g., using def boundaries in Python).[8] This method ensures that the logical unit of code or a technical specification remains intact, dramatically improving the semantic continuity for retrieval and

subsequent LLM processing.

The project mandates segmenting documents at multiple granularities: 2048, 512, and 128 tokens.[1] A highly effective strategy involves prioritizing the Document-Based approach where structure is identifiable, and employing recursive character splitting as a secondary method to precisely achieve the required token counts and ensure a context-aware overlap between chunks.[1]

Table II.1 details the rationale for this multi-granularity approach in the SE domain.

Table II.1: Multi-Granularity Chunking Strategy Justification for SE Content

Granularity (Tokens)	**Intended Use Case**	**Justified Chunking Strategy**	**Rationale for SE Domain**
2048	Broader context for reasoning/planning	Recursive Character Split (large overlap)	Captures full context of large code modules, long algorithms, or entire textbook chapters, supporting multi-step reasoning.
512	Standard Q&A, detailed concept retrieval	Document-Based (splitting by major headings/functions)	Optimized for semantic continuity in technical manuals or capturing full function bodies for accurate context.
128	Fine-grained detail retrieval, error codes, syntax lookup	Fixed-Size (small overlap)	Precise retrieval of definitions, single lines of code, or specific terminology, balancing speed and detail.

C. Hierarchical Indexing and Context Management

The mandate to use multi-granularity chunking inherently requires a robust indexing architecture to maximize retrieval effectiveness without overwhelming the LLM prompt. Simply indexing three sets of chunks independently leads to redundant information retrieval and prompt stuffing.

The solution is implementing a **Hierarchical (Parent-Child) Retrieval Indexing Strategy**, where embeddings are stored with relationships that maintain document structure.[1] In this strategy, the large chunk (2048 tokens) represents the comprehensive "parent context" (e.g., the entire file or class chapter). The smaller chunks (128 and 512 tokens) are the "children" used for the initial, rapid similarity search.

The architectural implication of this approach is significant: when the retrieval system identifies a highly precise, fine-grained chunk (e.g., a specific syntax error message or a small code block) from the 128-token set, the index structure automatically pulls the corresponding 2048-token parent chunk. This ensures that the retrieved context provided to the LLM includes not only the precise detail but also the necessary surrounding context (the full code module or chapter context) required to accurately diagnose and solve a technical problem. This mechanism actively mitigates the loss of context that often occurs when small chunks are used in isolation.[8]

III. RAG Core: Embedding, Indexing, and Retrieval Optimization

A. Embedding Model Selection for Code and Technical Text

The foundation model selection begins with the recommended baseline, all-mpnet-base-v2.[1] However, the core project requirement mandates experimenting with at least one other domain-specific embedding model to assess performance.[1]

General-purpose embedding models have been shown to struggle significantly when applied to specialized domains, such as finance [9], and this performance deficit is particularly pronounced in code-related tasks. Studies indicate that general models lack the capability to capture domain-specific linguistic patterns or the crucial metric of code functional consistency.[10]

For the SE SME, the domain-specific candidate chosen for experimentation is **GraphCodeBERT**. Models like CodeBERT and GraphCodeBERT are specifically pre-trained on programming language (PL) and natural language (NL) pairs, making them suitable for code search and code documentation generation.[11] GraphCodeBERT advances this capability by incorporating data flow learning, allowing it to leverage the inherent structural properties of source code, distinguishing it from models that treat code merely as a sequence of tokens.[13] While cutting-edge models (such as Voyage Code-3) achieve superior results (e.g., 97.3% MRR) in code retrieval [15], GraphCodeBERT provides a robust, open-source alternative to benchmark against the general-purpose baseline, particularly for structured content retrieval.

B. Indexing Strategy: Vector Database Selection

The indexed chunks and their associated embeddings must be stored in a vector database.[1] Suggested options include Elasticsearch, Pinecone, or Milvus. The selection criteria must be based on the project's mandatory requirement for implementing a hybrid retrieval system.[1]

Elasticsearch is selected as the optimal choice. While Milvus is suitable for high-scale enterprise deployments and Pinecone offers managed simplicity [16], Elasticsearch is recognized for its best-in-class implementation of **hybrid search**, natively supporting both dense vector indexing and traditional keyword-based (sparse) search mechanisms like BM25.[16] Selecting Elasticsearch simplifies the overall architectural implementation of the hybrid retrieval layer, avoiding the need to manage separate indexing systems for semantic and keyword search, thereby optimizing performance and maintenance.

C. Hybrid Retrieval Implementation and Score Fusion

A critical component of the RAG pipeline is the mandatory implementation of a hybrid retrieval system, combining dense vector search with a keyword-based approach like BM25.[1] This is essential for the SE domain because users search not only for conceptual similarity (semantic queries like "Explain the design pattern for state management in React") but also for exact

textual matches (e.g., "IEEE Std 982.1-1988").[17] Hybrid search leverages the strengths of both approaches, enhancing precision and recall.[18]

The process requires intelligently assembling the top-K most relevant chunks [1] retrieved by both methods. To fuse the scores generated by the semantic search and the keyword search into a unified relevance ranking, **Reciprocal Rank Fusion (RRF)** is the preferred mechanism.[19] RRF combines the results by aggregating the inverse rankings from each list, ensuring that documents ranked highly in either the keyword or the vector search receive a strong score.

The RRF score is calculated using the formula:

In this formula, is the set of documents, is a constant (often set to 60), and is the rank of document in the respective search outcome list.[19] By placing the document's rank in the denominator, RRF effectively imposes a penalty on documents that appear lower in either list, resulting in a more refined and relevant final ranking.[19]

The implementation of robust hybrid retrieval, fused by RRF and integrated natively within Elasticsearch, serves as a crucial compensating mechanism for architectural limitations. Specifically, if resource constraints or performance requirements necessitate the use of general-purpose embedding models (which struggle with structured SE content), the combined strength of the BM25 keyword matching and the sophisticated ranking of RRF proactively corrects for the known weaknesses of the base embedding model, thereby ensuring higher accuracy and relevance of the retrieved context.[10] RAG systems inherently provide access to up-to-date, authoritative domain data, avoiding the high cost and latency associated with model retraining.[20]

D. Post-Retrieval Enhancement: Reranking

Implementing a reranking step is a recommended enhancement and serves as a vital component for optimizing the final retrieved context.[1] While hybrid search (RRF) prioritizes recall by retrieving a broad set of context documents, the reranking step improves precision.

Using a cross-encoder model, such as the BGE Reranker [1], the retrieved top-K chunks are re-evaluated based on their contextual relationship to the original query. This secondary filtering process ensures that the few most relevant chunks actually passed to the LLM (typically to) are highly focused and precise, which is essential in technical problem-solving where small inaccuracies can lead to incorrect generative outcomes. The final retrieval process intelligently combines vector similarity, reranker scores, and metadata filters to determine the optimal subset of context for prompting.[1]

IV. Agentic Workflow Design and Orchestration

A. Framework Selection and Justification

The project requires the system to handle complex user requests, manage state, and chain multiple tools.[1] The critical requirement for robust error handling and multi-call recovery (e.g., if a DOCX generation fails, retrying with a PDF [1]) mandates a sophisticated orchestration framework.

While LangChain is excellent for linear workflows and rapid prototyping [21], it lacks the explicit control and state management required for complex, production-grade agentic systems involving loops and conditional branching. Therefore, the implementation selects **LangGraph or equivalent Custom Orchestration Logic**. LangGraph is specifically built for complex, multi-agent workflows with dynamic control flows and explicit state persistence.[21]

Opting for LangGraph or custom logic, despite the higher upfront developmental effort, provides the necessary architectural clarity for defining stateful execution paths, which is crucial for achieving high reliability, debugging complex failures, and enforcing the mandatory multi-step recovery policies.[22]

B. Conversational Planning and Reasoning Engine

The Agent/Brain serves as the LLM core, acting as the primary coordinator for the system.[5] This core must utilize sophisticated reasoning techniques to successfully decompose high-level goals and coordinate multi-step processes.[23]

The SE SME Agent's planning engine leverages techniques such as **Reflection or Critic-based frameworks** (like ReAct or Reflexion).[24] These techniques utilize the LLM's generative capabilities to refine its own execution plan based on intermediate feedback (observations) received during task execution. In the SE context, this feedback could include RAG retrieval scores, compilation errors from a coding tool, or validation failures from a structured document generator. This capacity for self-monitoring and adaptation is vital for

achieving complex software development tasks.[23]

The agentic system must be structured around established SE specializations.[25] These functional categories guide the routing and tool selection process:

1. **Code Generation and Software Development:** Automating code, refactoring, and providing recommendations.
2. **Requirement Engineering and Documentation:** Capturing, analyzing, and generating technical documentation.
3. **Autonomous Learning and Decision Making:** Adaptive planning and project management automation.
4. **Software Test Generation:** Generating unit tests and test suites.

C. Agent Routing Strategy for Complex SE Requests

To demonstrate the agent's ability to chain multiple tools [1], consider a complex user request: "Explain the Singleton pattern, generate a documented Python implementation, and create a short quiz to assess understanding, then email the complete package."

1. **Plan & Retrieve (Knowledge Retrieval):** The Agent routes to the RAG/Hybrid Search Tool to retrieve the canonical definition, implementation details, and design context for the Singleton pattern.
2. **Generate Code (Code Agent Tool):** The Agent routes the context to a dedicated internal Code Agent, which uses a specific LLM prompt to generate the Python example and its accompanying documentation.
3. **Create Educational Content (Content Generation Module):** The Agent routes the explanation and code to a Quiz Generation module, leveraging Few-Shot Prompting to produce structured assessment questions (e.g., in JSON format).
4. **Format and Deliver (Tool Chain):** The Agent orchestrates the Document Generation Tool to format the synthesized material (explanation, code, quiz) into a structured DOCX or PDF report. Subsequently, the Email Automation Tool is called to send the generated document to the specified recipient.

The requirement for robust error handling and multi-call recovery necessitates transforming the linear tool chain into a **stateful recovery graph**. If, for instance, the Document Generation Tool encounters a resource error while attempting to create the DOCX file, the agent must read the error (as an observation), update its execution state, and trigger a predefined conditional fallback. The use of LangGraph or similar custom orchestration allows for this explicit management of state and conditional looping, distinguishing it from simpler, less reliable LLM chains.[21]

V. LLM Integration and Adaptive Content Generation

A. Effective Prompt Design Strategy

The quality of the domain-specific outputs is heavily dependent on effective prompt design, leveraging techniques like few-shot prompting and iterative prompt correction.[1]

Few-Shot Prompting is critical for ensuring the agent produces consistent, structured outputs—a requirement for standardized documents like reports, lab assignments, or machine-readable quiz formats (e.g., JSON arrays).[27] By including multiple input-output examples, the model learns the necessary structural patterns, enhancing both reliability and accuracy.[27]

To guarantee technical accuracy and depth, the agent utilizes advanced prompt engineering:

1. **Generate Knowledge Prompting:** This technique forces the LLM to first retrieve and list the core principles, definitions, or technical standards relevant to the query *before* generating the final response.[29] For example, before drafting a security analysis, the agent is prompted to cite the specific clauses from the ACM/IEEE-CS code of ethics, resulting in responses that are significantly more informed and accurate.[29]
2. **Meta Prompting:** In complex or ambiguous problem-solving scenarios, the agent employs meta prompting, asking the model to generate or refine its own internal prompt to optimize the subsequent output.[29] This self-direction capability is essential for handling highly technical or vaguely phrased user requests, ensuring the agent clarifies the objective before committing to execution.

B. Adaptive Explanations and Multi-Step Reasoning Workflows

The requirement for adaptive explanations based on logical workflows [1] is directly linked to the agent's dual role. The system must adapt the complexity of its output based on the implied audience (peer engineer vs. student).

The agent achieves this by incorporating a crucial transformation step into the 'Learning

Material Generator' workflow. When generating notes for K-12 students, the agent first retrieves the complex, authoritative content (which is typically dense with jargon and low in readability). It then utilizes a generative transformation: the LLM is prompted to simplify and rephrase the retrieved text until it meets a defined readability target.

This process is governed by the necessity for pedagogical suitability. The goal is to produce content suitable for a Grade 6–8 US school level, which corresponds to a **Flesch Reading Ease score between 60 and 70**.[30] By actively measuring and correcting the output against this metric using a tool call (discussed in Section VII), the system ensures that complex software engineering concepts are rendered clearly and accessibly, thereby validating the educational mandate of the project.[1]

For high-stakes technical problem-solving, the LLM's reasoning capacity is maximized by applying multi-step approaches. The agent decomposes the problem, coordinates the execution of various tool calls, and monitors the outcome, feeding back observations for refinement. This iterative prompting correction adaptation strategy guides the model to produce high-quality, domain-specific outputs, with specific attention paid to documenting successful strategies and analyzing failure scenarios.[1]

VI. Tool Integration and System Robustness

A. Detailed Implementation of Essential Tool-Using Capabilities

Tool usage is foundational to the SE SME Agent, enabling capabilities far beyond standard text generation.[1] Custom tools can be created in frameworks like LangChain using decorators, abstracting complex external functions into simple, LLM-callable definitions.[31]

1. **Knowledge Retrieval Tool:** This is the modular Hybrid RAG pipeline (Section III), integrated to perform content discovery and provide grounding feedback.[1]
2. **Document Generation Tool:** This tool requires integration with external libraries (e.g., for DOCX and PDF creation) to export technical reports and learning materials.[1]
3. **Email Automation Tool:** This tool manages the sending of generated documents or instructions to specified contacts, requiring an abstracted interface to an external mail server (e.g., IMAP integration or using a toolkit like LangChain's Gmail Toolkit).[1]

B. Design of Complex, Chained Workflows

Complex user requests are fulfilled by designing workflows that intelligently chain these tools. The essential workflow for the Engineering domain is Technical Report Generation, which also serves as the necessary demonstration of robust error handling.

Table VI.1: Complex Tool-Chaining Workflow (Engineering Report Generation)

Step	Agent Action	Tool Used	Intermediate Output/State	Error Handling & Fallback
1	Plan & Retrieve Data	RAG/Hybrid Search Tool	Context documents, raw data points for report (e.g., IEEE standards).	If retrieval score is low, the agent uses Meta-Prompting to rephrase the query and retry (Self-Correction).
2	Draft Content & Structure	LLM (Generator Prompt)	Structured draft report text, technical analysis, and embedded code snippets.	If output fails structured validation (e.g., internal schema check), the agent routes back to the LLM with the error for immediate correction.
3	Format Document	Document Generation	DOCX file generated and saved	**MANDATORY RECOVERY:** If DOCX

	(Primary)	Tool (DOCX)	temporarily.	generation fails, the system captures the exception and immediately retries the formatting step using the PDF output format (Multi-Call Recovery).
4	Deliver Result	Email Automation Tool	Email sent to specified recipient with document attached.	If email fails, the agent logs the error and informs the user via chat that the document is ready for direct download.

C. Robust Error Handling and Multi-Call Recovery

The requirement for multi-call recovery [1] is satisfied by structuring the tool chain within a robust execution graph that anticipates failure. This implementation utilizes framework capabilities, such as LangChain's with_fallbacks mechanism, which defines alternative execution paths if the primary chain fails.[26]

Crucially, the system does not simply restart the process; it intelligently leverages the LLM's reasoning capabilities in the recovery phase. When a tool call fails (e.g., DOCX generation exception), the error message itself is captured and passed back into the LLM's prompt context as an observation (e.g., using a last_output message placeholder).[26] This allows the LLM to observe the failure (e.g., "Tool execution failed: DOCX format error") and deliberately adjust its subsequent plan, selecting the predefined fallback tool (PDF generation) rather than

attempting the failed action again.[1]

Furthermore, robustness extends to handling external environment noise. Drawing from principles of generalized agent failure mitigation, the architecture includes mechanisms such as readiness probing and fallback retry logic before executing tool calls that interface with external resources. This ensures that actions are issued only when target resources (like file systems for document creation or mail servers for email delivery) are confirmed to be available and responsive.[35]

VII. Evaluation and Validation Framework

The rigorous assessment of the SE SME Agent requires a validation framework that addresses both the functional correctness of technical solutions and the pedagogical quality of learning materials.[1]

A. Metrics for Code Generation Accuracy (Functional Correctness)

For technical problem-solving tasks, where the agent generates executable code, simple text-matching metrics are inadequate as they penalize functionally correct solutions that differ syntactically from a reference.[36]

The mandatory metric for this assessment is **Pass@k**. Pass@k measures the probability that a correct and functionally sound solution is generated among attempts.[36] It validates the quality of the generated code based solely on its execution against a comprehensive suite of unit tests.[36]

The calculation of the unbiased Pass@k metric is essential for benchmarking:

Where n is the total number of code samples generated for a problem, and c is the count of samples that successfully pass all test cases.38 The evaluation must specifically report results for k=1,3, and 5 37, indicating the agent's ability to achieve a correct solution in a low number of attempts.
The technical requirement to calculate Pass@k imposes a crucial architectural necessity: the external compiler or runtime environment required to execute the unit tests must be integrated as an internal, callable tool within the agent system.[36] The output of this

Compiler/Runtime Tool (Pass/Fail) is treated as an observation, which is fed back into the LLM's context. This mechanism closes the loop, allowing the SE agent to perform self-correction and iterative refinement of the generated code based on concrete, functional feedback, thereby transforming the evaluation framework into an active component of the autonomous learning process.[23]

B. Metrics for Documentation and Learning Material Quality (Pedagogical Suitability)

The SE SME Agent must effectively aid students and teachers.[1] This requires quantitative verification of the generated content's accessibility and clarity.

Readability Scores are employed to measure the difficulty of the generated explanations, notes, and quiz instructions. The two key metrics are:

1. **Flesch Reading Ease:** This score rates text on a 0–100 scale; a higher score indicates easier reading. The target for general, accessible content, suitable for the K-12 audience defined in the project scope, is 60–70.[30]
2. **Flesch–Kincaid Grade Level:** This metric converts the ease score directly into a US school grade level, with a target range of Grade 6–8.[30]

These metrics are essential for validating the agent's pedagogical output, ensuring that the transformation process described in Section V successfully simplifies complex retrieved SE standards into digestible learning materials.

C. Evaluation Summary and Retrieval Quality

To ensure the reliability of the RAG system, the retrieval component is assessed using standard information retrieval metrics, such as Normalized Discounted Cumulative Gain (NDCG) and Hit Rate.[39] These metrics specifically validate the efficacy of the Hybrid Search and RRF score fusion layer (Section III), confirming that the retrieved context is both relevant and correctly ranked before being passed to the generator.

The following table summarizes the key metrics deployed to validate the agent across its dual functions.

Table VII.1: Evaluation Metrics and Domain Relevance

Metric Category	Target Output	Calculation/Standard	Domain Justification
Functional Correctness	Generated Code Snippets	Pass@k (k=1, 3, 5): Functional correctness via unit tests (external execution).	Essential for the "Technical Problem-Solving" mandate, ensuring generated code is runnable and logically sound.
Pedagogical Suitability	Explanations, Learning Notes	Flesch Reading Ease (Target 60-70) or Grade Level (Target 7-8).	Ensures accessibility and clarity for the student/teacher audience, validating the educational mandate.
Retrieval Efficacy	Retrieved Context Chunks	Hybrid Retrieval Score (RRF) and NDCG/Hit Rate.	Validates the effectiveness of the hybrid search layer in balancing semantic meaning with technical keyword precision.

VIII. System Components & Architecture (Implementation Details)

The physical architecture is separated into distinct, communicative components, ensuring maintainability and scalability.[1]

A. Main API Server

The system requires a Main API Server, preferably built using a lightweight, performant framework such as FastAPI or Flask. This server handles incoming user requests and acts as the gateway to the agentic system. Implementation must include robust input sanitization for guardrails (preventing prompt injection) and output moderation (filtering harmful content), alongside comprehensive input validation.[1] Asynchronous support (using technologies like asyncio) is critical to ensure that blocking operations, such as lengthy LLM calls or external tool executions, do not halt the server's ability to handle concurrent requests.[1]

B. Chat/Tool/Agent Servers

A dedicated component manages the complex logic of the agent. This server manages user-specific chat contexts and conversational memory.[1] As dictated by the necessity for complex control flow, this component uses LangGraph or custom orchestration to manage explicit state, conditional logic, and the multi-step reasoning required for tool chaining and recovery.[21]

C. RAG Pipeline Integration

The modular RAG pipeline, incorporating embedding generation, the Hybrid Search index (Elasticsearch), RRF score fusion, and the post-retrieval reranking mechanism, is integrated as a self-contained service. This service is exposed to the Agent Server as the primary Knowledge Retrieval Tool, ensuring the agent can access grounded, authoritative, and up-to-date information efficiently.[1]

IX. Conclusions and Architectural Recommendations

The development of the Software Engineering Subject Matter Expert Agent requires an

architectural strategy that actively addresses the complexity inherent in handling structured, technical data, while simultaneously meeting a dual audience requirement (professional engineers and K-12 students).

The core recommendation is the strategic adoption of a non-linear, stateful orchestration framework (LangGraph/Custom Logic) to manage complex tool-chaining and, crucially, implement the mandatory multi-call recovery system.[1] This approach, combined with explicit error interpretation by the LLM (passing exceptions as observations), ensures system robustness against tool failure.

Furthermore, the retrieval mechanism must be rigorously optimized to compensate for the known limitations of general-purpose embeddings in the code domain. This is achieved by making the **Hybrid Retrieval (Vector + BM25) fused by RRF** and subsequent Reranking non-negotiable architectural layers.[1]

Finally, the project's success rests on its tailored evaluation framework. By coupling the functional metric **Pass@k** for code correctness with the pedagogical metric **Flesch Readability Scores** for learning materials, the system rigorously validates its capability to fulfill both the technical problem-solving mandate and the educational assistance mandate defined in the project scope.[1] The integration of the compiler/runtime as an agent tool to generate Pass/Fail feedback ensures that the evaluation is an active, self-correcting loop, promoting continuous improvement in the agent's generative accuracy.

Works cited

1. Major Project Document.pdf
2. The Software Engineering Code of Ethics and Professional Practice - ACM, accessed on October 13, 2025, https://www.acm.org/code-of-ethics/software-engineering-code
3. IEEE Recommended Practice for Software Requirements Specifications - Department of Mathematics and Statistics, accessed on October 13, 2025, http://www.math.uaa.alaska.edu/~afkjm/cs401/IEEE830.pdf
4. ACM SIGSOFT Empirical Standards for Software Engineering, accessed on October 13, 2025, https://www2.sigsoft.org/EmpiricalStandards/
5. LLM Agents - Prompt Engineering Guide, accessed on October 13, 2025, https://www.promptingguide.ai/research/llm-agents
6. RAG for Question Answering: Easy Examples & Use Cases | Django Stars, accessed on October 13, 2025, https://djangostars.com/blog/rag-question-answering-with-python/
7. RAG Engine API | Generative AI on Vertex AI - Google Cloud, accessed on October 13, 2025, https://cloud.google.com/vertex-ai/generative-ai/docs/model-reference/rag-api
8. Chunking Strategies to Improve Your RAG Performance - Weaviate, accessed on October 13, 2025, https://weaviate.io/blog/chunking-strategies-for-rag

9. Do we need domain-specific embedding models? An empirical investigation - arXiv, accessed on October 13, 2025, https://arxiv.org/html/2409.18511v3
10. Functional Consistency of LLM Code Embeddings: A Self-Evolving Data Synthesis Framework for Benchmarking - arXiv, accessed on October 13, 2025, https://arxiv.org/html/2508.19558v1
11. Top embedding models on the MTEB leaderboard | Modal Blog, accessed on October 13, 2025, https://modal.com/blog/mteb-leaderboard-article
12. CodeBERT - CodeSerra - Medium, accessed on October 13, 2025, https://codeserra.medium.com/codebert-83171b23c33c
13. Appendix: A Survey on Large Language Models for Software Engineering - arXiv, accessed on October 13, 2025, https://arxiv.org/html/2312.15223v2
14. microsoft/CodeBERT - GitHub, accessed on October 13, 2025, https://github.com/microsoft/CodeBERT
15. Code Isn't Just Text: A Deep Dive into Code Embedding Models | by Abhilasha Singh, accessed on October 13, 2025, https://medium.com/@abhilasha4042/code-isnt-just-text-a-deep-dive-into-code-embedding-models-418cf27ea576
16. We Tried and Tested 10 Best Vector Databases for RAG Pipelines - ZenML Blog, accessed on October 13, 2025, https://www.zenml.io/blog/vector-databases-for-rag
17. Hybrid Search a method to Optimize RAG implementation | by Akash Chandrasekar, accessed on October 13, 2025, https://medium.com/@csakash03/hybrid-search-is-a-method-to-optimize-rag-implementation-98d9d0911341
18. Advanced RAG Implementation using Hybrid Search and Reranking | by Nadika Poudel | Medium, accessed on October 13, 2025, https://medium.com/@nadikapoudel16/advanced-rag-implementation-using-hybrid-search-reranking-with-zephyr-alpha-llm-4340b55fef22
19. Optimizing RAG with Hybrid Search & Reranking | VectorHub by ..., accessed on October 13, 2025, https://superlinked.com/vectorhub/articles/optimizing-rag-with-hybrid-search-reranking
20. What is RAG (Retrieval Augmented Generation)? - IBM, accessed on October 13, 2025, https://www.ibm.com/think/topics/retrieval-augmented-generation
21. LangChain vs. LangGraph: A Developer's Guide to Choosing Your AI Workflow, accessed on October 13, 2025, https://duplocloud.com/blog/langchain-vs-langgraph/
22. LangChain vs Custom Workflows — AI Agents Guide 2025 - Ampcome, accessed on October 13, 2025, https://www.ampcome.com/post/langchain-vs-custom-workflows-ai-agents-2025
23. AI Agentic Programming: A Survey of Techniques, Challenges, and Opportunities - arXiv, accessed on October 13, 2025, https://arxiv.org/html/2508.11126v1
24. Introduction to LLM Agents | NVIDIA Technical Blog, accessed on October 13, 2025, https://developer.nvidia.com/blog/introduction-to-llm-agents/

25. From LLMs to LLM-based Agents for Software Engineering: A Survey of Current, Challenges and Future - arXiv, accessed on October 13, 2025, https://arxiv.org/html/2408.02479v2
26. How to handle tool errors - LangChain, accessed on October 13, 2025, https://python.langchain.com/docs/how_to/tools_error/
27. Zero-Shot, One-Shot, and Few-Shot Prompting, accessed on October 13, 2025, https://learnprompting.org/docs/basics/few_shot
28. Include few-shot examples | Generative AI on Vertex AI - Google Cloud, accessed on October 13, 2025, https://cloud.google.com/vertex-ai/generative-ai/docs/learn/prompts/few-shot-examples
29. Prompt Engineering Techniques | IBM, accessed on October 13, 2025, https://www.ibm.com/think/topics/prompt-engineering-techniques
30. Readability Score in AI Content: Why It Matters? - Wellows, accessed on October 13, 2025, https://wellows.com/content/readability-score-in-ai-content/
31. How to use tools in a chain | 🦜 LangChain, accessed on October 13, 2025, https://python.langchain.com/docs/how_to/tools_chain/
32. Using LangChain Tools to Build an AI Agent with Granite | IBM, accessed on October 13, 2025, https://www.ibm.com/think/tutorials/using-langchain-tools-to-build-an-ai-agent
33. Building your first Agent with LangChain and LangGraph: AI Email Agent | by Parth Sharma, accessed on October 13, 2025, https://medium.com/@parthshr370/building-your-first-agent-with-deepseek-ai-email-agent-e6f17d3c290e
34. Tools | 🦜 LangChain, accessed on October 13, 2025, https://python.langchain.com/docs/integrations/tools/
35. SHIELDA: Structured Handling of Exceptions in LLM-Driven Agentic Workflows - arXiv, accessed on October 13, 2025, https://arxiv.org/pdf/2508.07935
36. Pass@k: A Practical Metric for Evaluating AI-Generated Code | by Ipshita - Medium, accessed on October 13, 2025, https://medium.com/@ipshita/pass-k-a-practical-metric-for-evaluating-ai-generated-code-18462308afbd
37. Top Pass: Improve Code Generation by Pass@k-Maximized Code Ranking - arXiv, accessed on October 13, 2025, https://arxiv.org/html/2408.05715v1
38. Evaluating Large Language Models in Class-Level Code Generation, accessed on October 13, 2025, https://mingwei-liu.github.io/assets/pdf/ICSE2024ClassEval-V2.pdf
39. Retrieval Augmented Generation (RAG) for LLMs - Prompt Engineering Guide, accessed on October 13, 2025, https://www.promptingguide.ai/research/rag