

PIU Smart Intelligence Agents: Design and Implementation of a Security-Aware Multi-Agent Architecture for Enterprise Intelligence

Sujit Shibaprasad Maity¹

¹Senior Data Scientist, Mumbai, Maharashtra, India

Publication Date: 2026/10/22

Abstract: Enterprises increasingly rely on heterogeneous documents, relational databases, and analytical systems to support information retrieval and business decision-making. Integrating large language models (LLMs) with these sources introduces challenges in evidence reliability, analytical correctness, access control, and operational resilience. This paper presents PIU Smart Intelligence Agents, a security-aware multi-agent architecture designed to integrate enterprise document intelligence, retrieval-augmented generation (RAG), semantic search, document summarization, natural-language-to-SQL analytics, key performance indicator (KPI) generation, and conversational AI within a common application and governance framework.

The proposed architecture organizes specialized AI capabilities around shared services for authentication, authorization, request validation, provider integration, observability, and controlled data access. Its document-processing workflow encompasses file validation, text extraction, optical character recognition for scanned documents, normalization, chunking, embedding generation, and vector-based indexing. The structured-analytics workflow separates schema-aware query generation from SQL validation, restricted database execution, and result interpretation. The architecture also addresses provider compatibility, timeout handling, failure recovery, document lifecycle management, and traceability of generated outputs.

The source implementation reports 1,242 automated tests passed and two integration tests skipped. These results are reported implementation evidence and have not been independently reproduced. The available evidence does not establish comprehensive external RAG benchmark performance, quantitative correctness across all agents, independently validated security effectiveness, or production-scale performance and recovery guarantees. Accordingly, this study presents an integrated engineering architecture and identifies a reproducible evaluation framework for assessing its effectiveness. The principal contribution is the systematic integration of heterogeneous enterprise AI capabilities with explicit governance and validation requirements, rather than the introduction of a new learning or retrieval algorithm. Future work should prioritize agent-specific correctness testing, retrieval and answer-quality benchmarks, adversarial security evaluation, controlled load testing, and recovery verification.

Keywords: Enterprise Artificial Intelligence, Multi-Agent Systems, Retrieval-Augmented Generation, Intelligent Document Processing, Semantic Search, Natural-Language-To-SQL, KPI Generation, Conversational AI, Optical Character Recognition, Prompt Injection, Provider Resilience, Enterprise Security, AI Governance.

How to Cite: Sujit Shibaprasad Maity (2026) PIU Smart Intelligence Agents: Design and Implementation of a Security-Aware Multi-Agent Architecture for Enterprise Intelligence. *International Journal of Innovative Science and Research Technology*, 11(10), 1219-1270. <https://doi.org/10.38124/ijisrt/26oct511>

I. INTRODUCTION

➤ Background and Motivation

Enterprise organizations depend on information distributed across heterogeneous sources, including policies, contracts, standard operating procedures, technical documents, spreadsheets, operational databases, and business performance reports. Although these sources contain information relevant to operational planning and decision-making, their diversity of formats, storage systems, access

restrictions, and terminology can make it difficult to retrieve evidence and convert it into actionable insights.

Traditional enterprise information systems address these requirements through a combination of keyword-based search, document management systems, predefined reports, business intelligence dashboards, and database queries. These approaches remain valuable for structured and well-defined tasks, but they can require users to understand document repositories, database schemas, query languages,

and organization-specific reporting procedures. Consequently, answering a business question may involve multiple tools and manual steps, particularly when the answer requires both documentary evidence and structured numerical analysis.

Recent advances in large language models (LLMs) have enabled natural-language interfaces for information retrieval, summarization, and analytical assistance. Retrieval-augmented generation (RAG) extends this capability by incorporating retrieved information from external knowledge sources into the generation process. Semantic search can help identify conceptually relevant passages, while document summarization can reduce the effort required to review lengthy materials. Separately, natural-language-to-SQL techniques can translate user questions into database queries, and analytical workflows can calculate key performance indicators (KPIs) from structured data.

However, these capabilities address different information-processing requirements. Document question answering depends on source extraction, retrieval relevance, evidence selection, and faithful response generation. Structured-data analytics depends on schema interpretation, query correctness, aggregation rules, authorization, and controlled execution. KPI generation additionally requires consistent metric definitions, appropriate reporting periods, and traceability to source data. General-purpose conversational assistance introduces further requirements for controlling access to privileged tools and maintaining appropriate boundaries between ordinary conversation and enterprise operations.

Integrating these capabilities into a single application therefore creates a broader systems-engineering problem. The platform must coordinate specialized workflows while maintaining consistent identity and authorization policies, validating inputs and intermediate outputs, managing external provider dependencies, and preserving the traceability of generated results. These requirements motivate an architecture that treats enterprise AI as a governed collection of specialized capabilities rather than as a single language-model interface.

➤ *Problem Statement*

The central problem addressed in this study is how to integrate heterogeneous enterprise AI capabilities within a common application architecture without treating model-generated outputs as inherently correct, authorized, or operationally reliable.

A document-intelligence workflow may successfully extract text and generate embeddings while still retrieving incomplete or inappropriate evidence. A generated answer may be fluent but insufficiently supported by the available documents. Similarly, an SQL query may be syntactically valid but semantically incorrect, computationally expensive, or outside the requesting user's permitted data scope. A KPI may be calculated consistently by the software but remain misleading if its definition, reporting period, or aggregation logic is ambiguous.

These risks become more difficult to manage when multiple specialized agents share the same application. Each agent may require different data sources, tools, validation procedures, and permission boundaries. Applying isolated controls to individual components can produce inconsistent behavior, while relying exclusively on instructions in a model prompt does not establish effective access control or safe execution.

Operational dependencies create an additional challenge. Enterprise AI workflows may depend on language-model providers, embedding services, OCR libraries, databases, background-processing systems, and vector indexes. Failures or configuration changes can affect service availability and result consistency. In particular, changing an embedding model without an appropriate index migration and validation procedure can make existing indexed vectors incompatible with new query embeddings.

The engineering problem is consequently multidimensional: enterprise AI capabilities must be integrated in a manner that supports evidence traceability, controlled data access, analytical validation, dependency compatibility, operational observability, and systematic evaluation. These properties must be considered separately from the simple availability of application features or the number of automated tests executed.

This paper investigates that problem through PIU Smart Intelligence Agents, an enterprise AI platform organized around specialized agents and shared application-level governance services. The study describes the architecture, its principal workflows, and the engineering controls considered necessary for dependable operation. It also distinguishes the implementation evidence available from the additional experiments required to establish effectiveness under representative enterprise conditions.

➤ *Research Gap*

Existing approaches provide important building blocks for enterprise intelligence, including document search, semantic retrieval, RAG, conversational interfaces, natural-language database access, and specialized analytical applications. However, the availability of these individual techniques does not, by itself, resolve the engineering challenges involved in integrating them into a common enterprise environment.

For the purposes of this study, the research gap is framed as a system-level integration and evaluation problem, rather than as an assertion that previous research has not addressed these technologies.

Five related challenges motivate the proposed architecture.

First, end-to-end information reliability requires more than retrieval or generation in isolation. The quality of a document-based answer depends on the entire processing chain, including extraction, normalization, chunking, indexing, retrieval, context construction, and response

generation. Evaluating only whether the pipeline executes successfully is insufficient to establish that its answers are relevant, factually supported, and traceable to authoritative sources.

Second, heterogeneous agents require consistent governance with task-specific controls. Document retrieval, SQL execution, KPI calculation, and conversational assistance have different security and correctness requirements. A shared interface or orchestration mechanism does not, by itself, guarantee consistent authorization, restricted tool access, or validation of intermediate results. The architectural challenge is to provide common governance while preserving the distinct responsibilities of individual agents.

Third, provider resilience must preserve operational and semantic compatibility. Enterprise AI applications depend on services and components with different capabilities, configuration requirements, and failure behavior. Fallback mechanisms must account for these differences rather than assuming that every provider or model can be substituted transparently. Compatibility, timeout handling, bounded retries, failure reporting, and index migration therefore require explicit design consideration.

Fourth, structured analytics requires controls beyond language-model output generation. Natural-language-to-SQL and KPI workflows must respect approved schemas, permitted database operations, metric definitions, reporting periods, and source-data constraints. Correct execution alone does not prove analytical correctness, and a successful query-generation test does not establish that the resulting business interpretation is accurate.

Fifth, implementation evidence must be distinguished from effectiveness evidence. Automated software tests can provide useful evidence about tested behavior, but their count alone does not establish retrieval quality, answer factuality, agent-specific correctness, security robustness, scalability, or recovery effectiveness. These properties require suitable test datasets, metrics, threat scenarios, controlled workloads, and reproducible evaluation procedures.

Taken together, these challenges lead to the following research question:

How can heterogeneous enterprise AI capabilities be integrated within a modular application architecture that applies consistent governance, preserves evidence and analytical traceability, manages provider and processing failures, and supports reproducible evaluation of individual components and end-to-end behavior?

PIU Smart Intelligence Agents addresses this question through an engineering-oriented architecture and implementation description. The study identifies the design mechanisms used to organize the platform and the evaluation requirements needed to assess its effectiveness. It does not claim that the proposed integration is universally superior to

existing systems or that every identified challenge has been experimentally resolved.

➤ *Research Objectives and Questions*

The objective of this study is to describe and critically assess an integrated enterprise AI architecture that combines document intelligence, structured-data analytics, and conversational capabilities with shared governance and operational controls.

The study has six specific objectives:

- Describe the document-ingestion and retrieval workflow, including processing stages, source traceability, authorization considerations, and recovery requirements.
- Explain how specialized AI agents can use common application services while maintaining distinct task responsibilities and controlled tool access.
- Examine provider integration and compatibility-aware resilience, particularly where model or embedding changes may affect existing indexes and generated results.
- Define the controls required for safe structured-data analytics, including schema-aware SQL generation, query validation, restricted execution, and explicit KPI definitions.
- Distinguish reported implementation evidence from independently measured quality, security, performance, and reliability outcomes.
- Identify a reproducible evaluation framework and prioritized validation activities for future assessment.

These objectives lead to the following research questions:

• *RQ1 — Document Processing and Retrieval:*

How can heterogeneous enterprise documents be processed and indexed through controlled workflows that preserve document identity, access permissions, and source traceability?

• *RQ2 — Multi-Agent Architecture and Governance:*

How can specialized enterprise AI agents share common platform services while maintaining task-specific responsibilities, controlled tool access, and consistent authorization policies?

• *RQ3 — Provider Compatibility and Resilience:*

What architectural considerations are required to manage provider failures and model changes without introducing incompatible embeddings, uncontrolled fallback behavior, or avoidable service disruption?

• *RQ4 — Security and Safe Execution:*

Which application-level and infrastructure-level controls should be considered to reduce unauthorized information access, prompt-injection risks, unsafe database operations, sensitive-data exposure, and misuse of agent tools?

- *RQ5 — Analytical Correctness:*

How can structured-data answers and generated KPIs be validated against approved schemas, metric definitions, reporting periods, filters, and source data?

- *RQ6 — Engineering Evaluation:*

What conclusions are supported by the available implementation evidence, and which independent benchmarks, adversarial security tests, performance measurements, and recovery experiments remain necessary?

The research questions are addressed primarily through architectural analysis and examination of the reported implementation evidence. Questions concerning measured effectiveness are treated as evaluation requirements where the available evidence is insufficient to support a definitive empirical conclusion.

➤ *Scope and Research Contributions*

This study focuses on the architecture and engineering considerations of PIU Smart Intelligence Agents as an integrated enterprise AI platform. Its scope includes document processing, RAG-based question answering, semantic retrieval, document summarization, natural-language-to-SQL analytics, KPI-oriented workflows, and general-purpose conversational assistance. It also considers shared governance, provider integration, operational observability, and the validation requirements associated with these capabilities.

The paper does not propose a new embedding algorithm, retrieval-ranking method, language-model architecture, or general-purpose multi-agent coordination algorithm. Instead, its contribution is the integration-oriented organization of heterogeneous enterprise AI workflows and the explicit treatment of governance, compatibility, validation, and evaluation as cross-cutting engineering concerns.

The principal contributions are as follows.

- *A Unified Enterprise AI Reference Architecture.*

The paper describes a layered architecture for organizing document intelligence, RAG, structured-data analytics, KPI generation, and conversational assistance within a common application and governance framework. It identifies the role of shared services and the separation of specialized agent responsibilities.

- *An Integrated Document-Processing and Retrieval Workflow.*

The paper organizes the document-intelligence process around file validation, text extraction and OCR, normalization, chunking, embedding generation, indexing, permission-aware retrieval, and evidence-grounded response generation. It highlights the need to preserve document identity and source traceability across processing stages.

- *A Guarded Approach to Structured-Data Analytics.*

The paper describes a separate SQL-agent workflow that connects schema-aware query generation with

validation, restricted database execution, and result interpretation. It also identifies requirements for KPI definitions, reporting-period consistency, aggregation rules, and source-data lineage.

- *A Shared Governance and Resilience Model.*

The paper examines how authentication, authorization, input validation, restricted tool access, provider abstraction, timeout handling, circuit-breaker mechanisms, and compatibility-aware fallback can be addressed across specialized workflows. These mechanisms are presented as architectural controls whose effectiveness must be established through appropriate testing.

- *An Evidence-Aware Engineering Assessment.*

The paper records the reported automated test results while distinguishing them from independent empirical validation. It identifies gaps in retrieval and answer-quality measurement, agent-specific correctness testing, adversarial security assessment, load testing, and recovery verification.

- *A Prioritized Evaluation and Improvement Framework.*

The paper outlines the evaluation activities needed to strengthen confidence in the architecture, including versioned test datasets, component-level metrics, security test suites, controlled performance experiments, provider-failure testing, and verification of data restoration and consistency.

The source implementation reports 1,242 automated tests passed and two integration tests skipped. These figures are reported implementation evidence and have not been independently reproduced in this study. They should not be interpreted as a direct measure of answer quality, security effectiveness, scalability, or production readiness. Likewise, the presence of a named agent or architectural control does not, on its own, establish that the corresponding capability has achieved a measured level of correctness or effectiveness.

Accordingly, the contribution of this paper is a structured engineering description of an integrated enterprise AI platform, together with an explicit account of the evidence available and the validation work still required. This framing provides a foundation for subsequent empirical assessment without overstating the novelty or effectiveness established by the current implementation evidence.

II. TECHNICAL CONTEXT AND DESIGN PRINCIPLES

PIU Smart Intelligence Agents is designed around the premise that dependable enterprise artificial intelligence (AI) requires coordinated information processing, specialized task execution, explicit governance, and operational controls. The platform combines document intelligence, retrieval-augmented generation (RAG), structured-data analytics, KPI-oriented workflows, and conversational assistance within a common application architecture. These capabilities depend on different data representations and execution requirements; consequently, their integration must address both individual component behavior and end-to-end system properties.

This section establishes the technical context and design principles that inform the proposed architecture. It considers enterprise search and document management, evidence-grounded generation, multi-agent coordination, structured-data analytics, security, provider resilience, and reproducible evaluation. The discussion identifies engineering requirements rather than assuming that the inclusion of a particular technology or control establishes its effectiveness.

➤ *Enterprise Search and Document Management*

Enterprise organizations maintain information in heterogeneous formats, including text-based and scanned PDF documents, word-processing files, spreadsheets, technical specifications, business reports, policies, standard operating procedures, and operational records. These resources may be distributed across departments, repositories, databases, and document management systems. Effective information access therefore requires more than file discovery: it also requires content extraction, metadata management, document identification, version awareness, and authorization.

Traditional lexical retrieval is useful when a query contains terminology that closely matches the source document. However, users may describe a concept differently from the language used by the document author. Semantic search addresses part of this limitation by representing queries and document passages as vectors and retrieving candidates according to a similarity measure. Combining lexical retrieval, semantic similarity, metadata constraints, and deterministic identifiers can support a broader range of information needs than relying exclusively on a single retrieval mechanism.

Semantic similarity does not guarantee that a retrieved passage is accurate, authoritative, current, or relevant to the user's precise question. Retrieval quality also depends on the quality of the underlying document corpus and its processing pipeline. Optical character recognition (OCR) may introduce errors when processing low-resolution scans, complex layouts, tables, unusual fonts, or handwritten content. These errors can propagate through normalization, chunking, embedding generation, retrieval, and response generation.

Document metadata introduces additional considerations. Duplicate files, outdated versions, missing ownership information, inconsistent categories, and ambiguous document status can make it difficult to identify the appropriate source. A document-intelligence system should therefore preserve document identity and relevant metadata throughout processing. Where supported by the implementation, document states, processing outcomes, version information, and deletion or reprocessing events should remain traceable.

Authorization must also influence search. Enterprise repositories may contain information with different access restrictions. Retrieval should be limited to resources the requesting user is permitted to access, rather than retrieving unrestricted content and attempting to remove unauthorized

information only after generation. This requirement applies to candidate selection, context construction, cached results, and any downstream component that can access retrieved content.

For PIU Smart Intelligence Agents, these considerations motivate a document-processing workflow that connects file validation, extraction, normalization, chunking, embedding generation, indexing, metadata management, and permission-aware retrieval. The architectural objective is to preserve traceability and controlled access across the information lifecycle. The resulting retrieval and answer quality must nevertheless be established through appropriate evaluation.

➤ *Retrieval-Augmented Generation and Evidence Grounding*

Retrieval-augmented generation combines information retrieval with language-model generation. Rather than relying exclusively on knowledge encoded in model parameters, a RAG workflow retrieves relevant material from an external knowledge source and supplies selected evidence as context for response generation. This approach can support document question answering, contextual summarization, and natural-language access to enterprise knowledge.

A typical RAG workflow consists of two related processes.

- *Index Construction:*

Source documents are validated and processed; their text is extracted, normalized, and divided into manageable passages. The passages are associated with metadata and transformed into embeddings using a selected embedding model. The resulting representations and associated records are stored in an index or retrieval system.

- *Query Processing:*

A user's question is interpreted and converted into a retrieval request. Candidate passages are retrieved, applicable authorization and metadata constraints are enforced, and selected evidence is assembled into a bounded context. A language model then generates a response from the available context, with source references where supported.

The quality of the final answer depends on the complete pipeline. Extraction errors may corrupt source text, inappropriate chunk boundaries may separate a statement from its qualifications, and retrieval may omit relevant evidence or include misleading passages. Even when the correct passage is retrieved, the generation model may misinterpret it, combine conflicting statements, or introduce unsupported claims.

Accordingly, the presence of retrieved context should not be treated as proof that an answer is grounded. Evaluation should distinguish retrieval performance from generation quality. Retrieval evaluation examines whether relevant evidence appears among the retrieved results, whereas answer evaluation examines correctness, completeness, factual support, and consistency with the supplied evidence.

Citation quality is another distinct property. A response may contain a syntactically valid source reference that does not support the associated claim. A rigorous assessment should therefore consider source-location accuracy, claim-to-evidence alignment, citation completeness, and unsupported statements. Where the system cannot retrieve sufficient evidence, its behavior should be evaluated for appropriate qualification, clarification, or abstention rather than confident speculation.

PIU Smart Intelligence Agents treats document ingestion, retrieval, evidence-context construction, and response generation as connected stages. Its design objective is to support traceable, permission-aware information access. Claims about the resulting retrieval effectiveness, factuality, or citation quality require a versioned evaluation corpus and reproducible measurements; they cannot be inferred solely from the existence of a RAG workflow.

➤ *Multi-Agent Systems and Shared Governance*

An enterprise AI platform may combine several specialized capabilities, including document summarization, document question answering, natural-language-to-SQL analytics, KPI generation, and general-purpose conversational assistance. Although these capabilities can share application infrastructure, they do not necessarily share the same data sources, permissions, validation rules, or definitions of a correct result.

A multi-agent architecture provides a way to separate these responsibilities. An agent can be associated with a specific task contract that defines its permitted inputs, available tools, expected outputs, and applicable validation requirements. Orchestration coordinates request handling and agent execution, while common services support cross-cutting responsibilities such as identity, authorization, request validation, provider integration, logging, and error handling.

This separation should not be interpreted as a security boundary by itself. An agent's name, prompt, or declared purpose does not prevent unauthorized actions. Access to documents, databases, external services, and other tools must be enforced by the relevant application and data-access mechanisms. The platform should validate the requesting user's permissions and the requested operation before privileged resources are accessed.

Agent-specific validation remains necessary even when common services are available. A document-summary agent must preserve the meaning of its source material; a SQL agent must generate and execute permitted queries; a KPI workflow must apply approved metric definitions; and a general conversational agent should not automatically inherit access to privileged enterprise tools. The result produced by one agent should not be assumed trustworthy merely because another component generated it.

The architecture should also define how failures are represented and propagated. A failed retrieval, rejected query, unavailable provider, or incomplete background task should produce an explicit and observable outcome rather than

silently returning a result that appears complete. Where intermediate results are passed between agents, their origin, relevant constraints, and validation status should be preserved when the implementation supports such tracking.

In PIU Smart Intelligence Agents, the multi-agent design is intended to organize specialized workflows within a common application and governance framework. Its principal engineering consideration is the balance between reusable shared services and explicit task-specific controls. The effectiveness of this separation should be assessed through tests of agent boundaries, permission enforcement, tool invocation, intermediate-result validation, and failure handling.

➤ *Natural-Language-to-SQL and Analytical Correctness*

Natural-language-to-SQL systems translate user questions into database queries. They can make structured business data more accessible to users who do not routinely write SQL, but translating natural language into executable statements introduces correctness and security risks.

A generated query can be syntactically valid while selecting the wrong table, applying an incorrect date range, using an inappropriate aggregation, omitting a required filter, or interpreting a business term incorrectly. The query may also access unauthorized records or consume excessive database resources. Consequently, successful query generation or execution does not independently establish that an analytical answer is correct.

A guarded analytical workflow should separate several responsibilities:

- *Request Interpretation:*

Identify the requested measure, dimensions, reporting period, filters, and comparison requirements.

- *Schema Grounding:*

Use the authorized schema and relevant metadata to identify available tables, columns, and relationships.

- *Query Generation:*

Produce SQL consistent with the interpreted request and available schema.

- *Query Validation:*

Check permitted statement types, table and column access, required restrictions, query structure, and applicable resource limits.

- *Controlled Execution:*

Execute only validated queries using appropriately restricted database privileges and bounded resource consumption.

- *Result Interpretation:*

Present the returned data with sufficient context to explain the measure, filters, reporting period, and limitations.

These stages address different failure modes and should be evaluated separately where practical. A syntactically valid query may still be semantically incorrect, and a safe query may still return an inaccurate business interpretation.

KPI generation introduces a related but distinct requirement: business measures need explicit definitions. A metric should specify its calculation, source data, aggregation rules, reporting period, filters, and applicable inclusion or exclusion criteria. Measures with similar labels may represent different business concepts, and inconsistent definitions can produce misleading comparisons even when the underlying SQL executes correctly.

Data lineage is important for analytical traceability. Where supported, the system should associate a reported result with the relevant source data, calculation logic, reporting period, and applied filters. Validation should compare generated results with independently calculated reference values or approved test cases. Testing should include edge cases such as empty result sets, missing values, duplicate records, boundary dates, and ambiguous metric definitions where those cases apply.

For PIU Smart Intelligence Agents, SQL analytics and KPI-oriented workflows should therefore be treated as constrained analytical operations rather than unrestricted text generation. The architecture describes the controls and workflow boundaries required for this purpose. Quantitative claims about SQL correctness, KPI accuracy, or operational safety require dedicated tests and reference results.

➤ *Security Principles for Enterprise AI*

Enterprise AI security requires controls across the request, retrieval, execution, and result-delivery lifecycle. Because language models process natural-language instructions and potentially untrusted content, security cannot depend exclusively on prompt wording or on the model correctly following instructions.

- *Authentication and Authorization.*

Authentication establishes the identity of a requesting user or service, while authorization determines which resources and operations that identity may access. Authorization should be enforced at the relevant data and tool boundaries. A user's permission to ask a question does not automatically imply permission to retrieve every document or execute every database operation.

- *Input and File Validation.*

User inputs and uploaded files should be checked against the application's accepted formats, size limits, and processing requirements. File validation should be combined with appropriate handling of malformed or potentially unsafe content. Successful validation of file structure does not establish that its content is trustworthy.

- *Prompt-Injection Risk Management.*

Retrieved documents and other external content may contain instructions that conflict with application policies. Such content should be treated as untrusted input rather than

as an authority capable of overriding system controls. Prompt-level defenses should be complemented by explicit tool permissions, server-side policy enforcement, and tests of adversarial scenarios.

- *Restricted Execution.*

Database queries and other privileged operations should be constrained by application policy and appropriate infrastructure permissions. Validation should occur before execution, and resource limits should reduce the risk of unexpectedly expensive operations. Model-generated content should not directly confer privileges or bypass authorization.

- *Sensitive-Data Protection.*

Credentials, confidential source content, and sensitive outputs require appropriate handling across configuration, storage, logging, and provider interactions. Data minimization should limit the information supplied to components to what is required for the requested operation. Logs and traces should support investigation without unnecessarily reproducing sensitive content.

- *Auditability and Traceability.*

Where supported by the implementation, the system should record relevant request identifiers, processing outcomes, authorization decisions, tool executions, provider failures, and configuration changes. Audit records should be designed to support investigation while respecting access restrictions and data-retention requirements.

These controls address different security objectives and should not be treated as interchangeable. For example, authentication does not prove that a requested operation is authorized, and a query-validation mechanism does not establish resistance to every prompt-injection technique. Security assessment should therefore map each control to explicit threat scenarios and corresponding test procedures.

PIU Smart Intelligence Agents considers security as a cross-cutting architectural responsibility. The presence of a security mechanism in the design is not equivalent to demonstrated security effectiveness. Stronger claims require evidence from authorization tests, adversarial testing, controlled security assessments, and verification of the deployed configuration.

➤ *Provider Resilience and Dependency Compatibility*

Enterprise AI workflows may depend on external or locally hosted language models, embedding services, OCR libraries, databases, vector indexes, and background-processing components. These dependencies may differ in availability, supported operations, configuration, response formats, resource requirements, and failure behavior.

Network interruptions, rate limits, provider outages, malformed responses, and timeouts can interrupt processing. Resilience mechanisms such as bounded retries, timeouts, circuit breakers, provider abstraction, and explicit failure reporting can help manage these conditions. Their configuration should account for the operation being

performed and the possibility that repeated attempts may increase latency or resource consumption.

Provider fallback requires particular care because availability does not guarantee compatibility. Two embedding models may produce vectors with the same dimensions but different semantic representations. Mixing vectors generated by incompatible models can make similarity comparisons unreliable. A model change may therefore require a controlled re-embedding and re-indexing process rather than an immediate substitution.

A safe migration procedure should identify the active model and index configuration, build or update the replacement index, validate the new index, and define how the previous index remains available during migration. The system should avoid exposing an unvalidated index as the active retrieval source. Configuration records should identify the embedding model, relevant index settings, and other parameters needed to reproduce retrieval behavior.

Generation-provider fallback presents a different problem. Alternative models may vary in supported features, context capacity, output format, latency, or response behavior. A substitute should be used only when it satisfies the requirements of the operation, and the system should make failure or degraded behavior observable where relevant. Fallback should not silently relax authorization, validation, or correctness requirements.

Background document processing also requires explicit state and failure handling. Upload acceptance, extraction completion, indexing completion, and failure should be distinguishable states where implemented. Recovery procedures should address incomplete processing, retries, duplicate work, and consistency between document metadata and indexed content. Backup creation alone does not demonstrate recoverability; restoration and post-recovery consistency require verification.

In PIU Smart Intelligence Agents, provider abstraction and operational resilience are treated as architectural concerns intended to support controlled dependency integration. Their effectiveness must be established through fault-injection tests, compatibility checks, migration verification, recovery exercises, and measurements under representative workloads.

➤ *Reproducibility and Evaluation Principles*

A dependable enterprise AI system requires evidence appropriate to each claim. Functional testing, retrieval evaluation, analytical correctness testing, security assessment, and performance measurement answer different questions and should not be collapsed into a single measure of system quality.

- *Functional Verification.*

Automated tests can verify expected behavior for defined inputs and conditions. Their interpretation depends on test coverage, assertion quality, execution environment, and the distinction between unit, integration, and end-to-end

tests. A test count is not a substitute for reporting what was tested and which outcomes were observed.

- *Retrieval and Answer Evaluation.*

Document retrieval should be evaluated using a versioned corpus and queries with known relevant evidence. Answer evaluation should separately assess factual correctness, evidence support, completeness, and citation quality. Where applicable, abstention behavior should be evaluated for questions that cannot be answered from the available sources.

- *Analytical Correctness.*

SQL and KPI workflows should be evaluated against known expected results and approved business definitions. Test cases should cover query-policy violations, schema errors, ambiguous questions, reporting-period boundaries, and relevant data-quality edge cases.

- *Security Evaluation.*

Security tests should be derived from defined threat scenarios and access policies. They should assess unauthorized retrieval, privilege boundaries, adversarial document content, unsafe query attempts, and sensitive-data handling as applicable. Passing a particular test suite provides evidence for the tested scenarios but does not prove the absence of all vulnerabilities.

- *Performance and Resilience.*

Controlled experiments should report workload characteristics, concurrency, dataset size, environment, failure conditions, and relevant measurements. Latency distributions, throughput, error rates, resource consumption, and recovery behavior should be measured rather than inferred from configuration or architectural descriptions.

- *Reproducible Configuration.*

Evaluation results depend on source data, extraction configuration, chunking strategy, embedding model, index version, retrieval parameters, generation model, prompts, and evaluation datasets. These settings should be recorded sufficiently to explain and reproduce the tested configuration. Changes that may affect behavior should trigger appropriate regression testing.

The source implementation reports 1,242 automated tests passed and two integration tests skipped. These results should be presented as reported implementation evidence, not as independently reproduced experimental findings. Without the corresponding test definitions, coverage information, execution records, and independent evaluation, the count alone does not establish comprehensive correctness, security effectiveness, or production-scale reliability.

The evaluation approach in this paper consequently distinguishes three categories of evidence: implemented functionality, reported test outcomes, and independently measured system properties. This distinction provides a basis for interpreting the available evidence and prioritizing the additional experiments required to assess the platform rigorously.

➤ *Design Principles Derived from the Technical Context*

The preceding discussion yields eight design principles for PIU Smart Intelligence Agents.

- *DP1 — Separation of Responsibilities.*

Organize specialized agents around explicit task responsibilities, interfaces, and expected outputs, while keeping shared application services distinct from agent-specific execution logic.

- *DP2 — Evidence Traceability.*

Preserve document identity, relevant metadata, and source references through document processing and retrieval so that generated responses can be evaluated against their supporting evidence.

- *DP3 — Authorization at Resource Boundaries.*

Enforce user permissions and operation-specific restrictions before protected data or privileged tools are accessed. Do not rely on model instructions as the sole security mechanism.

- *DP4 — Validation Before Execution and Delivery.*

Validate inputs, generated database operations, intermediate results, and output requirements according to the task being performed. Treat syntactic validity and successful execution as insufficient evidence of semantic correctness.

- *DP5 — Explicit Analytical Semantics.*

Define KPI calculations, filters, reporting periods, and aggregation rules so that business results can be interpreted consistently and compared with reference values.

- *DP6 — Compatibility-Aware Dependency Management.*

Treat provider substitution, embedding-model changes, index migration, and configuration updates as controlled operations requiring compatibility checks and appropriate validation.

- *DP7 — Observable and Recoverable Processing.*

Represent meaningful processing states and failures explicitly, and verify recovery and data consistency through controlled tests rather than assuming that retries or backups guarantee successful recovery.

- *DP8 — Evidence-Based Assurance.*

Match each claim to suitable evidence, distinguish reported tests from independent measurements, and use reproducible component-level and system-level evaluation to guide production-readiness decisions.

These principles establish the conceptual basis for the system architecture described in the next section. They define the engineering concerns that the architecture is intended to address, without assuming that every principle has been fully implemented or independently validated in the current system.

III. PLATFORM OVERVIEW AND REQUIREMENTS

➤ *High-Level User Journey*

PIU Smart Intelligence Agents is designed as a unified enterprise AI platform through which authorized users can interact with structured business data, enterprise documents, and conversational AI capabilities using a common application interface. The platform organizes these interactions into a controlled workflow that includes user authentication, request submission, task validation, agent selection, processing, result generation, and final presentation. Depending on the requested operation, the workflow may also include document upload, asynchronous processing, source verification, data export, or administrative review.

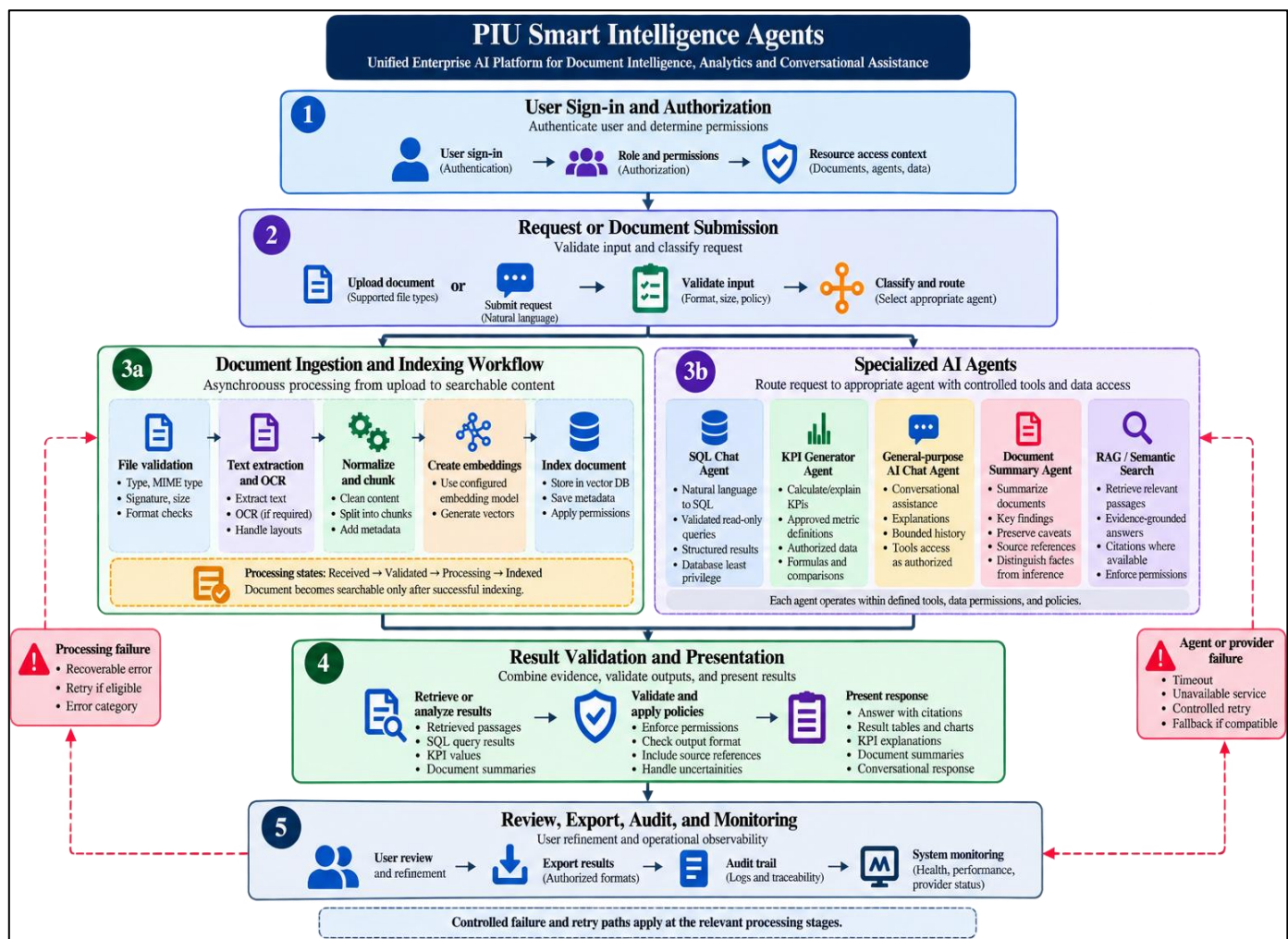


Fig 1 End-to-End User Journey through PIU Smart Intelligence Agents.

The user journey begins when a user signs in to the platform. The application establishes the user's authenticated identity and retrieves the relevant role, permissions, and resource-access context. These attributes determine which agents, documents, database resources, and administrative operations are available to the user. Authentication confirms the identity associated with a session, whereas authorization determines which operations that identity is permitted to perform. The two functions should remain separate so that successful sign-in does not automatically grant access to every platform capability.

After authentication, the user selects an available capability or submits a natural-language request. The available choices may include the SQL Chat Agent, KPI Generator Agent, General-purpose AI Chat Agent, Document Summary Agent, and RAG-based enterprise search. The application validates the request before processing it. Validation may include checking required fields, confirming supported input formats, resolving the requested operation, applying access restrictions, and determining whether the request contains sufficient information to proceed. Ambiguous requests may require clarification before execution.

For document-oriented tasks, the user may upload a file or select an existing authorized document. The platform should validate the uploaded file's type, size, structure, and other applicable constraints before accepting it for processing. Once the upload has been accepted, the document enters a processing workflow that may include text extraction, OCR, normalization, metadata assignment, chunk generation, embedding creation, and indexing. These operations can take longer than the initial upload request and may therefore execute as background jobs.

A critical design principle is the distinction between upload acknowledgement and processing completion. An acknowledgement indicates that the application has received or accepted the upload according to its intake policy; it does not establish that the document has been successfully parsed, embedded, or indexed. The document should become eligible for semantic search only after the required processing stages have completed successfully and the corresponding index records are available. If processing fails, the platform should retain an appropriate status and error category so that the failure can be investigated or retried under controlled conditions.

For conversational and analytical tasks, the platform routes the validated request to an appropriate agent. The SQL

Chat Agent processes supported questions over authorized relational data, while the KPI Generator Agent applies approved metric definitions and calculation rules. The General-purpose AI Chat Agent handles conversational assistance within its permitted scope, and the Document Summary Agent produces summaries of authorized source documents. RAG and semantic search retrieve relevant enterprise passages when a task requires document-based evidence. The precise routing strategy may depend on the request type, the available agents, and the application's policy configuration.

During execution, the selected agent may invoke retrieval services, language models, databases, or other approved tools. These operations should be subject to explicit resource limits, timeouts, authorization checks, and error-handling policies. If a dependency is unavailable or a request cannot be completed safely, the platform should report a controlled failure or a clearly identified partial result rather than presenting an unsuccessful operation as complete.

The final response is presented in a form appropriate to the requested task. Examples include a conversational answer, a document summary, a table of SQL query results, KPI values with formulas, or a ranked list of retrieved passages. Where supporting evidence is available, the response should retain source references or other traceability information. Analytical results should also identify relevant filters, time periods, and metric definitions where these affect interpretation.

The user may subsequently review the response, refine the request, export supported results, or initiate another operation. Export functionality should apply the same authorization and data-handling requirements as the underlying operation. Sensitive information should not become accessible merely because it is included in an exportable result.

The figure should illustrate authentication, capability selection, request validation, document or analytical processing, agent execution, result validation, response presentation, and optional export. For document ingestion, it should explicitly distinguish the upload acknowledgement state from the successful extraction and indexing state. Error handling and retry paths should be shown as controlled branches rather than as implicit transitions.

➤ *Agent Catalogue*

PIU Smart Intelligence Agents adopts a specialized-agent design in which individual capabilities address different enterprise information-processing requirements. Each agent has a defined purpose, expected inputs, output format, and applicable controls. This separation supports task-specific validation and testing while allowing the platform to share common services for authentication, authorization, logging, configuration, and error handling.

The catalogue below describes the intended logical responsibilities of the platform. Inclusion of a capability in the catalogue does not, by itself, establish that the capability has been fully implemented or independently evaluated.

The SQL Chat Agent addresses the gap between natural-language business questions and relational database queries. A user may ask for transaction volume over a reporting period or request a breakdown by a supported business dimension. The agent should map the request to approved tables and columns, identify the appropriate aggregation, construct the query, and submit it for deterministic validation before execution. The application should use restricted database credentials and should not rely solely on the language model to enforce read-only behavior. After execution, the displayed explanation must correspond to the query and results actually returned.

The KPI Generator Agent focuses on consistent business measurement. KPI definitions should be represented through explicit metadata or approved business rules wherever possible. Each definition may include the metric name, formula, numerator and denominator where applicable, aggregation method, unit, dimensional breakdown, reporting period, comparison baseline, and treatment of missing data. If the user requests a KPI whose definition is unavailable or ambiguous, the agent should ask for clarification or report the limitation instead of presenting an invented formula as an approved organizational metric.

The General-purpose AI Chat Agent provides a flexible conversational interface for questions that do not necessarily require database execution or document retrieval. Its responsibilities may include explanations, reasoning, and assistance with user-provided context. However, general conversational ability must not imply unrestricted access to enterprise records, administrative settings, or execution tools. When a question requires authoritative enterprise information, the agent should use an approved retrieval or analytical capability where available, or clearly communicate the limits of its answer.

The Document Summary Agent transforms authorized source content into a more concise and accessible representation. Depending on the application requirements, its output may include an executive summary, key findings, important dates, decisions, risks, action items, or section-level summaries. Summarization should preserve the meaning and limitations of the source. In particular, a summary must not convert tentative statements into confirmed facts or omit exceptions that materially affect a policy, requirement, or decision.

Table 1 Agent and Capability Catalogue

Agent / Capability	Purpose / Description	Applicable Context & Metadata	Controls & Enforcement
SQL Chat Agent	Handles questions over structured enterprise data.	Approved database schema, identity, and role context with applicable filters.	Returns an explanation grounded in the executed query. Enforces database least-privilege, query validation, execution limits, and result-size restrictions.
KPI Generator Agent	Calculates or explains business performance indicators.	Datasets, dimensions, filters, comparison periods, and reporting windows.	Uses explicit units and source lineage. Does not invent business definitions or silently change calculation rules.
General-purpose AI Chat Agent	Provides conversational reasoning, explanations, and general assistance.	Conversation history and explicitly authorized context or tools.	Distinguishes available evidence from unsupported assumptions. Enterprise data access and tool invocation require separate authorization.
Document Summary Agent	Transforms authorized enterprise text or retrieved passages into summaries.	Documents or selected passages with available metadata.	Provides source references where available. Preserves important qualifications, exceptions, warnings, and distinctions between source facts and inference.
RAG & Semantic Search	Retrieves enterprise passages and supports evidence-grounded answers.	Retrieval configuration and indexed content.	Provides answers grounded in retrieved evidence where generation is requested. Enforces authorization and preserves source traceability.
Orchestrator	Coordinates requests, routing to additional datasets, validated inputs, and explicit tool contracts.	Workflow tracking, execution data, and policy context.	Enforces policy, manages multi-step execution, and applies defined evaluation criteria.

RAG and semantic search provide reusable information-retrieval capabilities for the other agents. A retrieved passage may support a document summary, help establish an approved KPI definition, or provide contextual evidence for a conversational response. These services should return source identifiers and relevant metadata together with the retrieved text. The calling agent should receive only evidence that the requesting user is authorized to access.

A shared orchestration layer may classify a request, select a capability, validate the required inputs, and coordinate multi-step execution. For example, an analytical workflow may retrieve an approved metric definition, execute a permitted SQL query, and then explain the resulting value. Each step should have a documented input and output contract. The orchestrator should validate intermediate results and should not grant additional privileges merely because an agent requests them.

This catalogue supports modular extension, but adding an agent requires more than registering a new model prompt. Each new capability should define its business purpose, authorized resources, tool permissions, input and output schemas, timeout and failure behavior, audit requirements, and acceptance tests. These requirements help ensure that platform expansion does not undermine the security and consistency of existing capabilities.

➤ Functional Requirements

Functional requirements specify the operations the platform is expected to perform and the conditions under

which those operations are permitted. For PIU Smart Intelligence Agents, the requirements cover identity management, document ingestion, retrieval, agent routing, analytical execution, output generation, administration, and operational traceability.

• Authentication and Authorization

The platform shall authenticate users before permitting access to protected functionality. It shall associate requests with an authenticated identity and apply role- and resource-aware authorization checks to operations involving documents, database records, agents, exports, and administrative configuration.

Authorization shall be enforced at the application or service boundary responsible for the requested operation. A user interface restriction alone shall not be considered sufficient protection. The system shall also prevent a model-generated instruction or agent-generated plan from overriding the application's authorization policy.

• Document Intake and Validation

The platform shall accept supported document formats and validate uploaded files before processing. Validation should consider the declared extension, MIME type, file signature, file size, and format-specific structural constraints. Where applicable, the system should also apply controls for encrypted files, malformed documents, excessive page counts, embedded content, and potentially unsafe file structures.

The intake workflow shall distinguish rejected uploads from accepted uploads that are awaiting processing. A file that fails validation shall not proceed to ordinary document indexing. The system should record a suitable status and error category without unnecessarily exposing sensitive file content in operational logs.

- *Document Lifecycle and Background Processing*

The platform shall track the lifecycle of a document from initial submission through processing, indexing, and availability for retrieval. The lifecycle model should include meaningful states such as received, validation failed, queued, processing, indexed, and processing failed, as appropriate to the implementation.

State transitions shall reflect actual processing outcomes. A document shall not be marked searchable solely because its file upload succeeded or its extraction task was scheduled. The system should distinguish a recoverable processing failure from a permanent validation failure and provide controlled retry behavior for eligible jobs.

Background processing shall be designed to handle interrupted tasks and partial failures. Where retries are supported, duplicate execution should not produce uncontrolled duplicate index records. Reprocessing a document should also follow defined versioning or replacement rules so that stale content is not accidentally presented as current.

- *Text Extraction, OCR, and Indexing*

The platform shall extract text from supported digital documents and invoke OCR where the source requires image-based text recognition. Extracted content shall be normalized according to defined rules before being divided into retrieval chunks. The indexing pipeline shall preserve the metadata required for source identification, retrieval filtering, and authorization.

The platform shall generate embeddings using a configured embedding model and store the resulting vectors with their corresponding content identifiers and metadata. Index records shall be associated with a compatible model and vector configuration. Changes to extraction, chunking, or embedding configuration should be managed through a documented reprocessing procedure where necessary.

The system shall report processing outcomes at a level sufficient to identify failures in extraction, OCR, chunking, embedding generation, and index storage. The exact granularity of these states should be defined by the implementation and its operational requirements.

- *Semantic Search and Evidence Retrieval*

The platform shall retrieve relevant passages from indexed content using the configured retrieval strategy. Retrieval may incorporate semantic similarity, lexical matching, metadata constraints, and other approved ranking rules. The system shall apply the requesting user's document permissions and applicable resource filters before retrieved

content is supplied to a generation model or displayed to the user.

Where a response relies on retrieved content, the platform should preserve the available source identifiers and locators. If no sufficiently relevant evidence is found, the system should communicate that limitation rather than imply that the repository contains a verified answer.

- *Agent Routing and Execution*

The platform shall route validated requests to an appropriate available agent or capability. Routing decisions shall follow explicit task definitions and should be observable through operational metadata sufficient for debugging and evaluation.

Each agent shall operate within its documented tool and data permissions. Inputs received from users, retrieved documents, and other agents shall be treated according to their trust level and validated before use in sensitive operations. Agent execution shall also be subject to configured limits on processing time, output length, retries, and external resource consumption.

When a task cannot be completed, the platform shall return a controlled failure or clearly identified partial result. It should not represent a failed database query, incomplete document-processing job, or unavailable model response as a successful result.

- *SQL Generation and Execution*

The SQL Chat Agent shall generate queries only for supported analytical operations over approved database resources. Generated statements shall be validated against the permitted schema and execution policy before reaching the database. The execution environment shall use least-privilege database permissions and enforce applicable limits on execution time, returned rows, and resource consumption.

The system shall distinguish query-generation errors, policy-validation failures, database execution errors, and empty result sets. These outcomes have different meanings and should be presented accordingly. Where the query is displayed to the user, it should correspond to the statement that was actually validated and executed.

The agent shall not infer missing database columns or assume that a business term maps to a particular field without sufficient schema or semantic context. Where the requested analysis cannot be expressed using the approved schema, the system should explain the limitation or request clarification.

- *KPI Calculation and Explanation*

The KPI Generator Agent shall calculate or explain indicators using approved metric definitions and authorized data. A KPI result should include the applicable calculation rule and the reporting context needed to interpret it, such as time window, dimensions, filters, and comparison baseline.

The platform shall distinguish measured values from derived values and explanatory narrative. Growth rates,

period-over-period comparisons, and other derived indicators shall use explicitly defined calculation rules. Where a denominator is zero, the baseline is missing, or the requested metric is ambiguous, the system shall apply an approved handling rule or report that a reliable result cannot be produced.

- *Conversation, Summary, and Export*

The platform shall support conversational interaction within configured history limits. Conversation context shall be restricted to the information needed for the task and shall remain subject to applicable data-access and retention policies.

The platform shall generate document summaries from authorized source material and preserve important qualifications and available references. It shall also support export of applicable results using configured output formats. Export operations shall be subject to the same authorization and data-protection requirements as the underlying information.

- *Administration and Auditability*

The platform shall provide authorized administrative operations for supported configuration and operational management tasks. Depending on the implementation, these operations may include provider settings, model selection, processing controls, and service health inspection. Administrative functions shall be separated from ordinary user capabilities and protected by explicit permissions.

The system shall record sufficient audit and operational information to reconstruct significant processing events, identify failures, and support security review. Audit records should capture relevant identities, operation types, timestamps, resource identifiers, outcomes, and correlation identifiers where available. Sensitive prompts, documents, credentials, and result content should not be recorded unnecessarily.

➤ *Nonfunctional Requirements*

Nonfunctional requirements define the quality attributes and operational constraints that determine whether PIU Smart Intelligence Agents can function reliably in an enterprise environment. These requirements should be translated into measurable acceptance criteria for the target deployment. Unless supported by benchmark results, they represent engineering objectives rather than verified performance claims.

- *Security*

Security shall be treated as a cross-cutting property rather than a feature confined to the authentication layer. The platform should validate user identity, enforce resource-level permissions, protect credentials, restrict database access, and limit the information included in logs and model contexts. File processing and retrieval should account for untrusted content, while model-generated SQL and tool arguments should be validated independently of model instructions.

Prompt-injection resistance should be evaluated across direct user inputs, retrieved documents, and intermediate agent outputs. The design should assume that untrusted text may contain instructions attempting to alter the task or bypass controls. Such text must not override system policies, and access decisions must remain under application control.

- *Reliability and Fault Tolerance*

Reliability requires controlled behavior when a database, embedding endpoint, language model, OCR component, or background-processing service becomes unavailable. Network operations should use appropriate timeouts and bounded retry policies. Circuit breakers or equivalent controls may be used where repeated failures would otherwise overload a dependency or delay unrelated requests.

Provider fallback requires particular care. A substitute language model may be acceptable for some generation tasks if its behavior and output contract are compatible with the operation. An embedding provider cannot automatically replace another provider when the vector spaces are incompatible. In such cases, the system may need to use a matching index, perform re-embedding, or report that the operation is temporarily unavailable.

Background jobs should preserve meaningful processing states and support controlled recovery after interruption. Retry policies should avoid repeated execution of non-idempotent operations and should not create inconsistent document or index records. Reliability testing should include dependency timeouts, partial failures, interrupted processing, and recovery after services become available again.

- *Performance*

Performance should be measured at both component and end-to-end levels. A single overall response time does not identify whether delays originate from file extraction, OCR, embedding generation, vector retrieval, language-model inference, database execution, or export generation.

For interactive operations, useful measurements include median latency and higher-percentile latency, together with throughput and failure rates. For asynchronous document processing, queue delay and processing duration should be measured separately. Measurements should be collected under documented workload conditions, including input size, concurrency, model configuration, and dependency environment.

Performance targets should be established for the intended deployment and verified through reproducible tests. Without such results, the paper should describe the measurement methodology and design objectives rather than claim that the platform meets a particular latency or throughput threshold.

- *Scalability and Resource Management*

Scalability depends on controlling the amount of work that each request can initiate. The platform should impose

configurable limits on file size, page count, extracted text length, number of chunks, concurrent processing jobs, context size, output tokens, database query duration, and returned rows.

These controls help prevent unusually large or malformed inputs from exhausting application memory, queue capacity, model context windows, or database resources. Where workload isolation is required, document processing, interactive chat, and analytical execution may use separate queues or resource pools. The appropriate arrangement depends on expected usage and the deployment architecture.

Scaling should be evaluated through controlled load and stress tests. Relevant observations include throughput, queue growth, resource utilization, error rates, recovery behavior, and tail latency. The ability to start additional application instances does not by itself establish scalability if the database, model provider, or processing queue remains a bottleneck.

- *Maintainability*

The platform should use modular services and explicit interfaces to separate request handling, business logic, retrieval, database execution, model-provider integration, and presentation. Shared concerns such as authorization, configuration, error handling, and observability should be implemented consistently rather than duplicated in every agent.

Database schema changes should be managed through versioned migrations. Provider configuration and retrieval

settings should be documented and versioned so that behavior can be reproduced after deployment changes. Automated tests should cover component logic, interface contracts, database behavior, authorization boundaries, and important end-to-end workflows.

A modular architecture is valuable only when component boundaries are supported by stable contracts and tests. Consequently, maintainability should be assessed through code quality, change isolation, migration safety, and regression behavior rather than inferred from directory structure alone.

- *Observability*

Observability is required to understand how a request moves through authentication, routing, agent execution, retrieval, external dependencies, and result presentation. The platform should associate related events with a correlation identifier where possible and record meaningful lifecycle transitions, error categories, execution durations, and dependency status.

For document processing, operational metrics should distinguish queued, active, successful, and failed jobs. For AI operations, useful measurements include selected agent, provider response status, retrieval duration, generation duration, token usage where available, and validation outcomes. SQL operations should capture execution metadata without unnecessarily exposing sensitive query results or business data.

Table 2 Non-functional Requirements

Quality Attribute	Requirement Intent	Verification Approach
Security	Enforce authentication, authorization, least privilege, secret protection, data minimization, and defenses against malicious input and prompt injection.	Access-control tests, adversarial input tests, secret-handling checks, dependency review, and authorized security assessment.
Reliability	Apply timeouts, bounded retries, provider routing, circuit breakers where appropriate, recoverable background jobs, and consistent processing states.	Fault-injection tests, dependency-failure tests, retry verification, and lifecycle-state consistency checks.
Performance	Measure upload, extraction, OCR, embedding, retrieval, generation, SQL execution, and export separately.	Stage-level timing, end-to-end latency distributions, throughput tests, and resource measurements under defined workloads.
Scalability	Bound file size, page count, extracted characters, chunk count, concurrent jobs, output tokens, and database query cost.	Load and stress testing, resource monitoring, queue-depth analysis, and limit-enforcement tests.
Maintainability	Use layered services, explicit interfaces, versioned configuration, database migrations, and automated tests.	Code review, migration testing, interface-contract tests, regression suites, and reproducible deployment checks.
Observability	Provide correlation identifiers, lifecycle events, error categories, queue metrics, provider status, and latency measurements.	Trace-completeness checks, log review, metrics validation, and controlled incident simulations.
Governance	Apply retention rules, ownership checks, auditable configuration changes, and documented provider and data-residency decisions.	Policy review, audit-log inspection, access-control tests, retention verification, and configuration-change audits.

Logging and metrics should follow data-minimization principles. Operational visibility must not become a secondary mechanism for collecting confidential documents, credentials, or unrestricted conversation content. Retention periods and access to logs should be defined according to the deployment's governance requirements.

- *Governance and Data Management*

Enterprise governance requires clear rules for data ownership, retention, source versioning, provider selection, and the handling of sensitive information. The platform should define which data may be sent to external model providers, what retention conditions apply, and which deployment or data-residency constraints must be satisfied.

Document ownership and authorization metadata should remain associated with indexed content so that access restrictions can be applied during retrieval. When documents are updated or withdrawn, the system should have a defined procedure for updating or removing their searchable representations. The procedure should also account for cached content and derived records where applicable.

Administrative configuration changes, including changes to providers, models, retrieval settings, and access policies, should be auditable. Governance decisions should be documented and verified through configuration review and appropriate tests. Specific legal or regulatory compliance should be claimed only when the relevant requirements and evidence have been established.

- *Roles and Assumptions*

PIU Smart Intelligence Agents distinguishes ordinary users from administrators because these groups perform different operations and require different permissions. Ordinary users interact with the platform's approved information and analytical capabilities, whereas administrators manage authorized operational and configuration functions. This separation reduces the risk that routine interaction with an AI agent can alter sensitive platform settings or grant access to restricted resources.

- *Ordinary User Role*

Ordinary users may be permitted to upload supported documents, search authorized content, submit conversational requests, request permitted database analyses, generate approved KPIs, summarize accessible documents, and export eligible results. Each operation remains subject to resource-level authorization and applicable processing restrictions.

A user's ability to access one capability does not imply permission to access all underlying data. For example, permission to use the conversational interface should not automatically permit access to every indexed document or database table. The platform should evaluate the requested resource and operation using the application's authorization policy.

- *Administrator Role*

Administrators may be authorized to manage supported operational settings, inspect system health, configure model-

provider endpoints, review processing failures, and perform other approved administrative tasks. The exact administrative scope depends on the implemented role model and deployment configuration.

Administrative access should be limited to the functions required for the administrator's responsibilities. Changes to provider settings, model selection, retrieval configuration, and access policies should be validated and recorded. Credentials and secrets should be stored through an appropriate secret-management mechanism rather than exposed in user-facing interfaces, ordinary application logs, or generated responses.

- *Deployment Assumptions*

The documented architecture assumes a PostgreSQL database deployment, an accessible embedding endpoint, and one or more configured language-model endpoints. It also assumes that the application can access its configured storage, document-processing components, and other required dependencies.

These assumptions define the intended operating environment; they do not guarantee that every dependency is available in every deployment. The PostgreSQL version, enabled extensions, database schema, vector-index configuration, embedding model, language-model provider, and provider-specific API behavior must be confirmed for the actual environment. Where vector functionality depends on a database extension, deployment verification should establish that the extension is installed, enabled, and compatible with the configured index and query operations.

External provider behavior remains dependent on the provider's service availability, rate limits, authentication requirements, API contract, model lifecycle, and applicable data-handling terms. A provider integration should therefore define configuration validation, timeouts, error mapping, and any compatible fallback behavior. If a dependency cannot be replaced safely, the platform should report a controlled service limitation instead of silently substituting an incompatible component.

- *Assumptions and Limitations*

The platform assumes that users submit requests within the supported application scope and that authorized enterprise data sources provide sufficiently accurate and current information. However, uploaded documents may be incomplete or corrupted, metadata may be inconsistent, database records may contain errors, and language models may generate incorrect interpretations. These conditions require validation, source attribution, and explicit handling of uncertainty.

The platform also assumes that approved business definitions are available for KPIs and that the database schema contains the fields required for supported analytical queries. Where such definitions or fields are missing, the system should not fabricate them. Instead, it should request clarification, identify the limitation, or decline the unsupported operation.

Finally, the catalogue, requirements, and deployment assumptions in this section define the intended platform behavior. Claims about production readiness, security effectiveness, agent accuracy, retrieval quality, or performance must be supported by implementation evidence and appropriately designed experiments. These requirements provide the basis for the architecture, implementation, and evaluation discussed in the following sections.

IV. SYSTEM ARCHITECTURE

➤ *Layered Reference Architecture*

PIU Smart Intelligence Agents adopts a layered reference architecture to organize enterprise AI functionality into components with clearly defined responsibilities. The architecture separates user interaction, request handling, API contracts, orchestration, business logic, external provider integration, and data persistence. This separation is intended to improve maintainability, testability, security enforcement, and the ability to extend the platform with additional AI capabilities.

The presentation layer provides the user-facing interface through which users authenticate, upload documents, submit analytical questions, interact with specialized agents, review results, and export supported outputs. It is responsible for collecting user input, displaying processing states, presenting errors, and rendering responses in an appropriate format. The presentation layer should not independently determine whether a user is authorized to access a document, execute a database operation, or change administrative configuration. Those decisions belong to trusted backend services.

The edge and middleware layer handles cross-cutting request concerns, including request parsing, input validation, authentication integration, request-size restrictions, correlation identifiers, and consistent error handling. Depending on the deployment, this layer may also implement rate limiting, cross-origin controls, request logging, and other perimeter protections. Security-sensitive decisions should remain enforced at the appropriate backend service and data-access boundaries, even when similar checks are performed earlier in the request path.

The versioned API layer exposes application capabilities through stable interfaces. API versioning allows the platform to evolve request and response contracts without requiring every client to change simultaneously. Each endpoint should define its accepted inputs, authentication requirements, authorization conditions, response schema, and expected failure modes. Changes to an API should be managed so that clients can distinguish compatible improvements from breaking changes.

The orchestration and business-services layer coordinates application workflows. It may classify incoming requests, select a suitable agent, retrieve authorized context, invoke approved tools, and validate results before they are returned to the user. Business services implement domain-specific rules such as document lifecycle management,

metric resolution, SQL validation, retrieval policy, and export eligibility. Separating orchestration from business logic helps prevent agent prompts from becoming the sole location of critical business rules.

The provider and processing adapter layer encapsulates interactions with external or locally hosted model endpoints, embedding services, OCR components, file parsers, and other processing dependencies. Adapters normalize provider-specific request formats, responses, exceptions, and configuration requirements into application-level contracts. This boundary allows the platform to change provider configuration without unnecessarily rewriting its core workflows. However, an adapter should not conceal incompatibilities between models or treat different provider outputs as interchangeable without validation.

The persistence and indexing layer maintains application records, document metadata, source references, conversation records where configured, and the structures needed for retrieval and analytical execution. It may include a relational database, controlled file or object storage, and a vector-capable retrieval index. The architecture should preserve the relationship between source documents, extracted text, chunks, embeddings, and retrieved results so that information can be traced back to its origin.

These layers communicate through explicit interfaces rather than unrestricted access to internal implementation details. Controllers should remain relatively thin: they validate the request at the API boundary, invoke the appropriate service, and serialize the response. Business rules, authorization decisions, provider-specific behavior, and persistence operations should reside in the components responsible for those concerns.

The layered design also establishes clear testing boundaries. Presentation behavior can be tested independently of provider integration; orchestration can be evaluated using controlled agent responses; database operations can be tested against a defined schema; and provider adapters can be exercised with simulated success, timeout, and failure conditions. End-to-end tests are still necessary because correct individual components do not automatically guarantee a correct overall workflow.

The figure should depict the presentation layer, edge and middleware, versioned APIs, orchestration and business services, provider and processing adapters, and persistence and indexing as distinct layers. Arrows should represent controlled request and response paths, while shared security and observability controls should span the relevant layers.

➤ *Multi-Agent Orchestration*

The multi-agent orchestration layer coordinates specialized capabilities within a common application workflow. Its purpose is to interpret the user's requested task, select an appropriate capability, provide the required context, invoke approved services, and return a validated result. The orchestrator serves as a workflow coordinator; it should not

be treated as an unrestricted autonomous authority over enterprise data or system resources.

The workflow begins with request classification. The platform determines whether the request concerns structured-data analytics, KPI calculation, document summarization, semantic retrieval, general conversational assistance, or another supported operation. Classification may use deterministic routing rules, model-assisted intent detection, or a combination of both. Regardless of the routing mechanism, the resulting decision should be constrained by the set of capabilities that are actually available and permitted for the authenticated user.

After classification, the orchestration layer selects a suitable agent and prepares its execution context. That context should contain only the information required for the task, such as an approved schema description, authorized document passages, metric definitions, validated filters, or bounded conversational history. Sensitive data should not be included merely because it is available to the orchestrator. This follows the principle of data minimization and reduces the risk of accidental disclosure through model inputs or generated responses.

The selected agent then performs its assigned operation using an explicitly defined set of tools. For example, the SQL Chat Agent may request a read-only database operation, while the Document Summary Agent may process authorized extracted text. Tool access should be mediated by trusted application services that validate arguments, check permissions, enforce resource limits, and normalize errors. A model-generated plan must not be allowed to invoke arbitrary functions, change authorization policy, or bypass the approved execution boundary.

When an operation completes, the orchestrator or a designated validation service checks the returned result. Validation may include schema conformance, completeness, execution status, evidence availability, policy compliance, and consistency with the original request. For SQL

operations, validation should establish that the executed statement was permitted and that the displayed results correspond to the actual database response. For RAG responses, validation should preserve source references and identify whether the available evidence supports the requested answer.

The orchestrator should define controlled behavior for failure conditions. These may include unavailable providers, timeouts, invalid tool arguments, empty retrieval results, database errors, and malformed agent outputs. Depending on the operation, the system may retry within configured limits, choose a compatible alternative, request clarification, return a partial result, or report that the task could not be completed. Retries should not be used to conceal persistent failures or repeatedly execute operations that are not safe to repeat.

For traceability, each workflow should carry a correlation identifier across the relevant API request, orchestration steps, tool calls, and response generation. Operational records should capture the selected agent, task category, relevant processing stages, outcome, and failure classification. Where useful, the system may record a routing reason code, but it should avoid retaining unnecessary sensitive prompt content or complete document passages merely to explain the routing decision.

The agent architecture is intended to support modular extension. A new agent should define its purpose, input and output schemas, accessible resources, approved tools, authorization requirements, timeout policy, error behavior, and evaluation criteria before being introduced into the orchestration workflow. The platform should also verify that the new capability does not expand the permissions of existing agents or introduce unintended paths between restricted data sources.

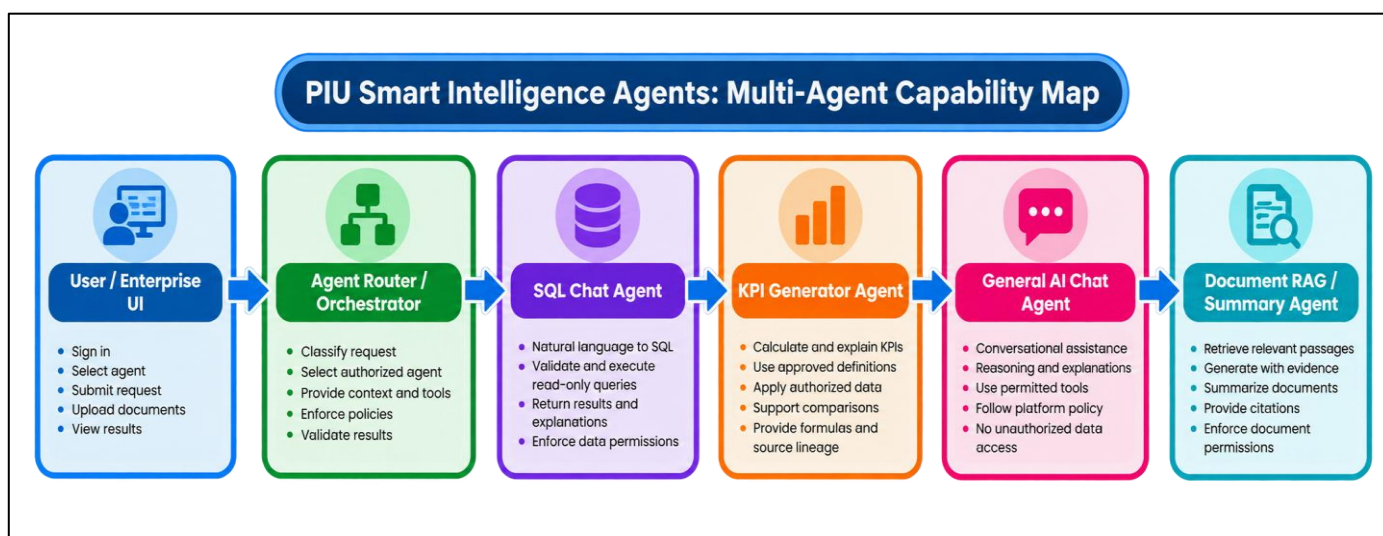


Fig 2 Conceptual Agent Capability Map.

The figure should show the orchestrator connected to the SQL Chat Agent, KPI Generator Agent, General-purpose AI Chat Agent, Document Summary Agent, and RAG/Semantic Search capability. Shared services for authentication, authorization, tool validation, logging, and output validation should be shown as governance components. This is a logical architectural view; the actual routing rules, interfaces, and execution paths must be confirmed against the deployed code.

➤ *API Architecture*

The API architecture provides a controlled interface between the presentation layer and backend services. Versioned API contracts allow clients to interact with platform capabilities without depending on internal service implementation details. Each endpoint should define the request schema, response schema, authentication requirements, authorization conditions, and possible error outcomes.

A request typically passes through boundary validation before being dispatched to the corresponding application service. Validation checks whether the request conforms to the expected schema, whether required fields are present, and whether input limits are satisfied. Authentication establishes the requesting identity, while authorization determines whether the identity may perform the requested operation on the specified resource. Resource-sensitive authorization should also be enforced inside the responsible service or data-access boundary rather than being assumed from a successful API-level check alone.

API contracts should distinguish synchronous operations from asynchronous workflows. Lightweight operations may return their results directly, whereas document ingestion and indexing may require a background-processing model. In the latter case, the API can acknowledge accepted work and return a job or document identifier, allowing the client to inspect processing status separately. The status endpoint should reflect the actual lifecycle state and should not report an indexed document as searchable before the required processing has completed.

For long-running operations, the API should define how timeouts, cancellation, duplicate submissions, and retries are handled. Where appropriate, idempotency mechanisms can help prevent repeated submissions from creating duplicate work. The exact mechanism depends on the operation and must be implemented consistently across request handling and background processing.

Responses should follow consistent schemas so that clients can interpret success and failure without relying on provider-specific error messages. Error categories should distinguish at least the following cases:

- *Validation Failure:*

The request or its arguments do not satisfy the defined contract.

- *Authorization Denial:*

The authenticated user is not permitted to perform the operation or access the resource.

- *Processing Failure:*

An internal task, such as parsing or indexing, did not complete successfully.

- *Provider Unavailability:*

A required external or internal dependency is unavailable or has rejected the operation.

- *Timeout:*

An operation did not complete within its configured execution window.

- *Empty Result:*

The operation completed successfully but returned no matching records or relevant passages.

- *Internal Failure:*

An unexpected application error occurred and requires controlled reporting and investigation.

These conditions should not be represented as equivalent outcomes. For example, an empty retrieval result is not necessarily a system failure, and a permission denial should not be retried as if it were a temporary network problem. API responses should expose enough information for the client to respond appropriately while avoiding disclosure of internal stack traces, credentials, sensitive queries, or restricted resource details.

API observability should include correlation identifiers, operation categories, execution duration, and outcome codes. These fields support tracing across the orchestration and provider layers without requiring every service to log complete request content. Sensitive values should be omitted or redacted according to the platform's logging policy.

The API architecture should be validated through contract tests, authorization tests, negative-input tests, timeout simulations, and end-to-end workflow tests. Version compatibility and migration behavior should also be evaluated before a new API version is promoted to a target deployment.

➤ *Data Stores*

The persistence architecture separates structured application records, original source files, extracted content, and vector-based retrieval representations according to their distinct operational requirements. This separation helps the platform manage document lifecycle state, preserve source traceability, support structured analytical execution, and update search indexes without treating all data as one undifferentiated store.

The relational database serves as the authoritative store for structured records managed by the application. Depending on the implemented schema, these records may

include user and role metadata, document registry entries, document categories, processing status, source references, conversation or message records, search history, and operational histories. Structured records also provide the identifiers and relationships required to associate processed content with its source document and current lifecycle state.

Relational constraints and transactions should be used where appropriate to maintain data consistency. For example, a document registry entry should have a stable identifier, and its processing status should be updated according to defined state transitions. Schema changes should be managed through versioned migrations so that deployments can establish which database structure is expected by a given application release.

Original files may be stored in a controlled file system or object-storage service. The storage layer should apply appropriate access restrictions, retention rules, and backup procedures. Where documents can be replaced or reprocessed, the system should maintain sufficient version and identity information to distinguish the source file from its derived text, chunks, and embeddings.

Extracted text and chunk records preserve the content used for retrieval. Each chunk should retain the identifiers and metadata required to locate its source document and, where available, its page or section. This relationship is important because a generated response may need to reference the source passage from which its evidence was retrieved. The system should also define how derived content is updated or removed when a source document is changed, withdrawn, or no longer authorized for retrieval.

The vector index stores embedding representations associated with chunk identifiers and relevant metadata. It supports similarity-based candidate retrieval, but it should not be treated as the authoritative record of document ownership or access policy. Retrieval must use trusted authorization information to ensure that only permitted content is returned to the caller. Depending on the deployment, the vector index may reside within the relational database through a compatible extension or in a separate retrieval service.

The source design describes a PostgreSQL-oriented persistence model and a 768-dimensional embedding configuration using cosine-distance retrieval. These are documented design details, not proof that every deployment contains a functioning vector index. The target environment must verify the PostgreSQL version, availability and activation of any required extension, the actual embedding model, vector dimensions, collection or table definitions, distance operator, and index configuration. A PostgreSQL installation does not automatically provide every vector-search capability.

If an extension or vector index is unavailable, deployment should fail validation for the affected retrieval feature or use an explicitly configured alternative that satisfies the same interface and security requirements. The system should not silently claim that semantic retrieval is operational when the required storage or indexing component has not been configured.

The data architecture should also address backups and recovery. Relational metadata, source files, and vector indexes may have different backup and reconstruction procedures. The deployment plan should establish which data is authoritative, which structures can be rebuilt from source records, how restoration is validated, and how consistency is re-established after recovery. These procedures should be tested rather than assumed from the presence of backup configuration.

➤ *Provider Management and Resilience*

PIU Smart Intelligence Agents may depend on one or more language-model providers, embedding services, OCR components, and other processing dependencies. Provider management isolates these integrations behind configured adapters and a common application-facing interface. The objective is to support controlled provider selection, consistent error handling, observable dependency status, and safe recovery from temporary failures.

The provider manager receives an application operation and resolves the configured provider and model appropriate to that operation. Configuration may identify the endpoint, model, authentication mechanism, timeout, retry policy, and supported request or response format. Sensitive credentials should be supplied through an appropriate secret-management mechanism and must not be included in ordinary logs, user-visible errors, or generated prompts.

Each provider adapter translates application-level requests into the provider's required format and converts the response into a defined internal representation. The adapter should classify failures such as authentication errors, rate limiting, service unavailability, malformed responses, and timeouts. This allows the application to apply a suitable recovery strategy without making every agent depend on provider-specific exception handling.

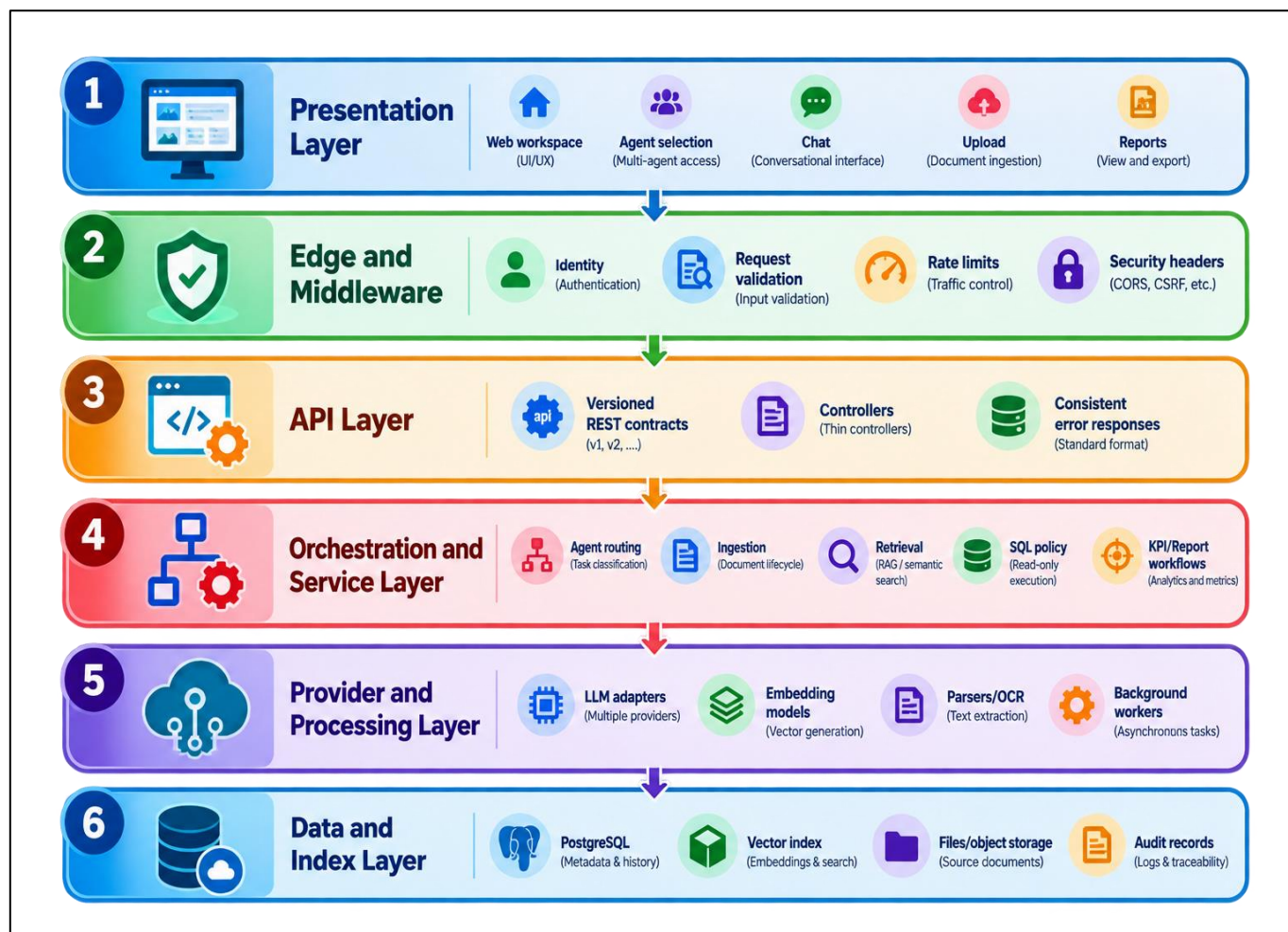


Fig 3 Layered Reference Architecture and Separation of Responsibilities

Timeouts and retries should be bounded. A timeout limits how long the application waits for an operation, while a retry policy determines whether another attempt is justified. Retrying a request that failed because of a temporary network problem may be appropriate; repeatedly retrying an invalid request or an authorization denial generally is not. Retry policies should also account for rate limits and avoid amplifying an outage through excessive concurrent attempts.

A circuit breaker or equivalent health-control mechanism can temporarily stop calls to a dependency after repeated failures. When the dependency becomes eligible for recovery testing, the application may allow controlled attempts before restoring normal routing. Health state should be based on defined operational signals rather than on the success or failure of a single unrelated request. The implementation should also define how circuit-breaker state is shared or isolated across application instances.

Fallback behavior must be specific to the operation being performed. A generation model may be replaced when the alternative supports the required task and its request and response contracts are compatible with the application. However, compatibility should not be inferred solely from the fact that both providers accept text input.

Embedding substitution requires stricter treatment because embedding models can represent text in different vector spaces. A replacement model is safe for an existing index only when compatibility has been established. If the replacement produces incompatible embeddings, the system must use a matching index, rebuild the index using the new model, or report that the operation cannot currently be completed. Equal vector dimensions alone do not establish semantic compatibility.

Provider routing and fallback decisions should be observable. The system should record the selected provider identifier, operation category, outcome, latency, and failure classification where appropriate. It should not log API keys, authorization headers, or unnecessary sensitive prompt and document content. Monitoring should also distinguish a provider outage from an application-level validation error or an empty search result.

Resilience testing should cover timeouts, rate limits, malformed responses, authentication failures, unavailable endpoints, circuit-breaker transitions, and recovery behavior. For embedding services, testing should additionally confirm that provider changes cannot silently mix incompatible vectors. The presence of retry or fallback code alone is insufficient evidence that the platform remains correct under dependency failure.

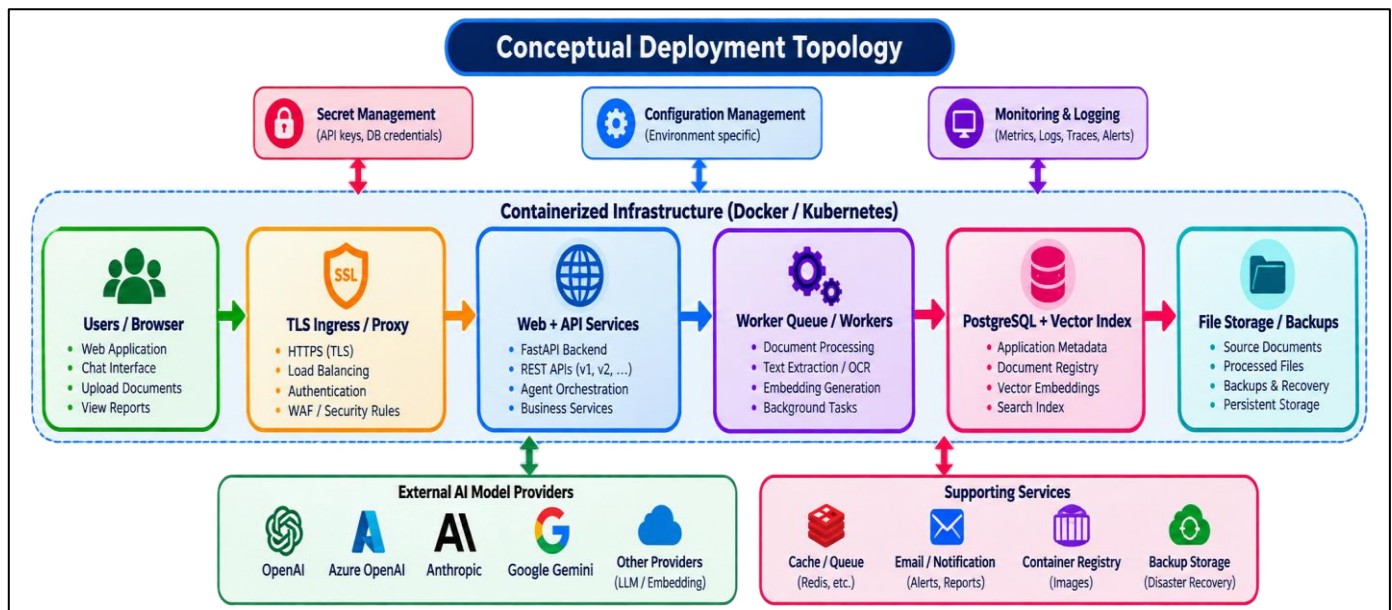


Fig 4 Provider Routing, Circuit-Breaker Behavior, and Fallback Paths.

The figure should show application requests entering a provider manager, passing through the selected adapter, and reaching the configured service. Failure paths should include timeout handling, bounded retry, health-state evaluation, and compatible fallback. Embedding-provider changes should be shown with an explicit compatibility check and an index-rebuild path when required.

➤ Deployment Architecture

The deployment architecture defines how the application services, database, source-file storage, retrieval index, and external model providers are configured and operated in the target environment. The intended deployment model uses containerized services, configuration-driven environment differences, health checks, controlled network communication, and persistent storage for stateful components. The exact topology must be verified against the deployment artifacts and infrastructure configuration.

A typical logical deployment may include a presentation service, backend API service, background document-processing worker, relational database, file or object-storage service, vector retrieval component, and one or more external or locally hosted AI providers. These components represent architectural responsibilities; they do not necessarily require a separate container for every component. The deployment should select service boundaries according to operational needs, dependency management, scaling requirements, and failure isolation.

Container images should include only the runtime dependencies needed by the service. Where feasible, application processes should run as non-root users and use restricted filesystem permissions. Configuration values should be supplied through environment-specific configuration or a secret-management mechanism rather than embedded in source code or committed to version control. Production credentials must be distinct from development

credentials, and secrets should be rotated according to the deployment's security requirements.

Health checks should distinguish process availability from dependency readiness. A process may be running even though it cannot access the database or a required model endpoint. The deployment should therefore define suitable liveness and readiness checks for the relevant services. Checks should be designed carefully so that a temporary dependency issue does not cause unnecessary restart loops or hide the underlying failure.

Network configuration should restrict communication to the services that require it. Database ports and administrative interfaces should not be exposed publicly unless there is an explicit, reviewed requirement and suitable protection. External traffic should pass through an appropriately configured TLS termination point, while internal service communication should follow the security requirements of the deployment environment. Network restrictions should complement, not replace, application authentication and authorization.

Persistent data requires explicit volume or storage configuration. Database files, source documents, indexes, and other stateful records should not depend on ephemeral container filesystems. Backup schedules, retention periods, restoration procedures, and disaster-recovery responsibilities should be documented. Recovery procedures should account for the relationships between relational metadata, source files, and derived indexes so that the restored system does not present inconsistent or stale search results.

Deployment configuration should also define how schema migrations, index creation, and background workers are initialized. Startup order alone should not be treated as proof that dependencies are ready. The application should handle dependency unavailability gracefully and should

report a controlled failure when a required component is missing or incompatible.

Environment-specific configuration should separate development, testing, staging, and production settings. This includes database endpoints, model-provider settings, credentials, logging levels, resource limits, and allowed origins. Production changes should be reviewed and traceable, and rollback procedures should be defined for application releases and incompatible schema changes.

Finally, the deployment should be evaluated through repeatable installation, health-check, backup-restoration, security-configuration, and failure-recovery tests. TLS termination, network policy, database exposure, secret injection, persistent storage, monitoring, and disaster recovery must be confirmed in the target environment. Containerization and a successful application startup do not, by themselves, establish that the system is secure, resilient, or production-ready.

V. DOCUMENT INGESTION AND PROCESSING

➤ *Asynchronous Ingestion Pipeline*

Document ingestion is a foundational component of PIU Smart Intelligence Agents because the accuracy, accessibility, and traceability of downstream retrieval and AI-generated responses depend on the quality of the indexed source material. The ingestion subsystem is designed to transform heterogeneous enterprise documents into validated, structured, and searchable representations while preserving document identity, ownership, processing status, and source metadata.

Enterprise documents may vary considerably in format, size, layout, content quality, and extraction complexity. Some documents contain directly extractable digital text, whereas others consist of scanned pages requiring optical character recognition (OCR). Large documents may require multiple processing stages, and embedding generation may depend on external or locally hosted model services. Consequently, the ingestion workflow is designed around asynchronous execution to prevent lengthy processing operations from blocking interactive application requests.

The proposed workflow separates document submission from document preparation and indexing. The initial API request is responsible for validating the upload, establishing a durable document record, and scheduling eligible processing work. A background-processing component then performs extraction, normalization, chunking, embedding generation, and index persistence.

This separation enables the application to acknowledge an accepted upload without incorrectly indicating that the document is already searchable.

• *Document Submission and Preliminary Validation*

The ingestion process begins when an authenticated user uploads a supported document through the platform

interface or an authorized API endpoint. The request includes the file and, where applicable, associated metadata such as document category, department, title, description, or other classification attributes.

Before accepting the document for processing, the application should verify the user's upload permissions and evaluate the submitted file against configured intake restrictions.

Validation should include the following controls:

- ✓ Verification of the permitted file extension and detected MIME type.
- ✓ Inspection of file signatures and relevant format structures.
- ✓ Enforcement of maximum file size and applicable upload quotas.
- ✓ Detection of malformed, encrypted, or unsupported document structures.
- ✓ Validation of supplied metadata and resource ownership.
- ✓ Rejection of unsafe file paths or filenames used in storage operations.
- ✓ Application of format-specific resource limits.

File extensions alone should not determine the processing method because a file may be incorrectly named or deliberately disguised. Where technically feasible, the platform should inspect the file's actual structure before selecting the appropriate parser.

A rejected file should receive a controlled validation response and should not enter the ordinary indexing workflow.

• *Document Registration and Job Scheduling*

After the file passes preliminary validation, the application creates a durable record in the document registry. This record establishes the document's identity and provides the basis for tracking its processing lifecycle.

The registry may contain a document identifier, owner or authorized resource scope, original filename, detected content type, file size, storage reference, processing status, creation timestamp, and relevant classification metadata.

The application then schedules a background-processing job associated with the registered document.

The transition between document registration and job scheduling requires careful handling. If the registry update succeeds but the job cannot be scheduled, the system must not leave the document indefinitely marked as processing without a recovery mechanism. A durable job record, transactional outbox, reconciliation process, or equivalent design can be used to identify and recover such inconsistencies.

The API acknowledgement should communicate that the upload has been accepted or queued. It must not imply

that parsing, embedding generation, or indexing has already completed.

• Background Document Processing

A background worker retrieves an eligible document-processing job and verifies that the associated document is in a state that permits processing.

The worker performs the following logical operations:

- ✓ Load the authorized source file from controlled storage.
- ✓ Select the parser according to the validated document format.
- ✓ Extract embedded text or perform OCR where necessary.
- ✓ Normalize the extracted content.
- ✓ Divide the content into retrievable text chunks.
- ✓ Associate each chunk with source and processing metadata.
- ✓ Generate embeddings using the configured compatible model.
- ✓ Persist chunk records and vector representations.
- ✓ Verify that the required indexing operations completed successfully.
- ✓ Transition the document to its searchable state.

Each stage should produce an explicit outcome so that failures can be associated with the operation that caused them.

For example, a parsing failure should be distinguishable from an embedding-provider timeout or a database persistence error. This distinction improves operational diagnosis and supports more appropriate retry policies.

• Processing State Management

The ingestion lifecycle should use a clearly defined state model rather than relying on loosely related Boolean flags.

A representative lifecycle is:

**RECEIVED → VALIDATED → QUEUED →
PROCESSING → INDEXED**

Alternative terminal or recovery states may include **VALIDATION_FAILED**, **PROCESSING_FAILED**, **RETRY_PENDING**, and **DELETED**, according to the implemented workflow.

The state model should enforce valid transitions. A document that has not completed extraction and indexing should not move directly from RECEIVED to INDEXED. Similarly, a failed document should not become searchable merely because some chunks were written before the failure.

The INDEXED state should indicate that the document's required searchable representation is complete and available through the approved retrieval path.

Where partial indexing is intentionally supported, the platform should use a separate and explicitly defined availability state. Partial results should not be silently presented as a complete document index.

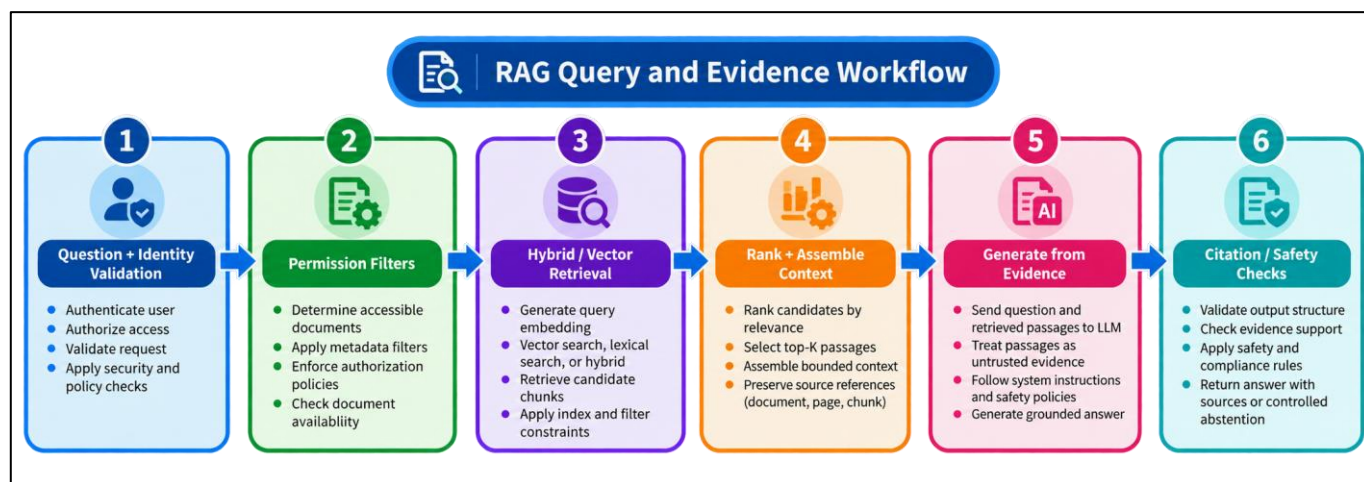


Fig 5 RAG Query Pipeline from Identity Validation to Citation and Safety Checks

• Idempotency and Retry Management

Asynchronous processing introduces the possibility of duplicate job delivery, interrupted execution, and repeated attempts after temporary failures.

An ingestion operation should therefore be idempotent where practical: processing the same document version and job more than once should not create uncontrolled duplicate records or inconsistent document states.

A practical design may associate each processing attempt with a document identifier, source version, processing configuration version, and unique job identifier.

Before writing derived records, the worker should determine whether the corresponding document version has already been processed successfully. Database uniqueness constraints, controlled upsert operations, version-specific

staging records, or equivalent mechanisms can help prevent duplication.

Retries should be permitted only when the failure is considered recoverable. Temporary provider unavailability or network timeouts may justify another attempt, whereas an unsupported file format or irrecoverably corrupted source should normally result in a terminal failure.

Retry attempts should be bounded and should record the failure category and attempt count. A failed attempt must not overwrite a previously valid searchable version unless the replacement has been successfully prepared and committed.

- *Completion and Search Availability*

The final stage verifies the relationship between the document registry, extracted content, chunk metadata, and stored embeddings.

A document should be marked INDEXED only when the required index records are present and consistent with the document version and configured embedding model.

The retrieval subsystem should additionally enforce an availability check so that documents in processing, failed, deleted, or superseded states are not returned as active searchable content.

This separation between index persistence and retrieval visibility is important because the relational registry and vector index may not participate in one atomic transaction.

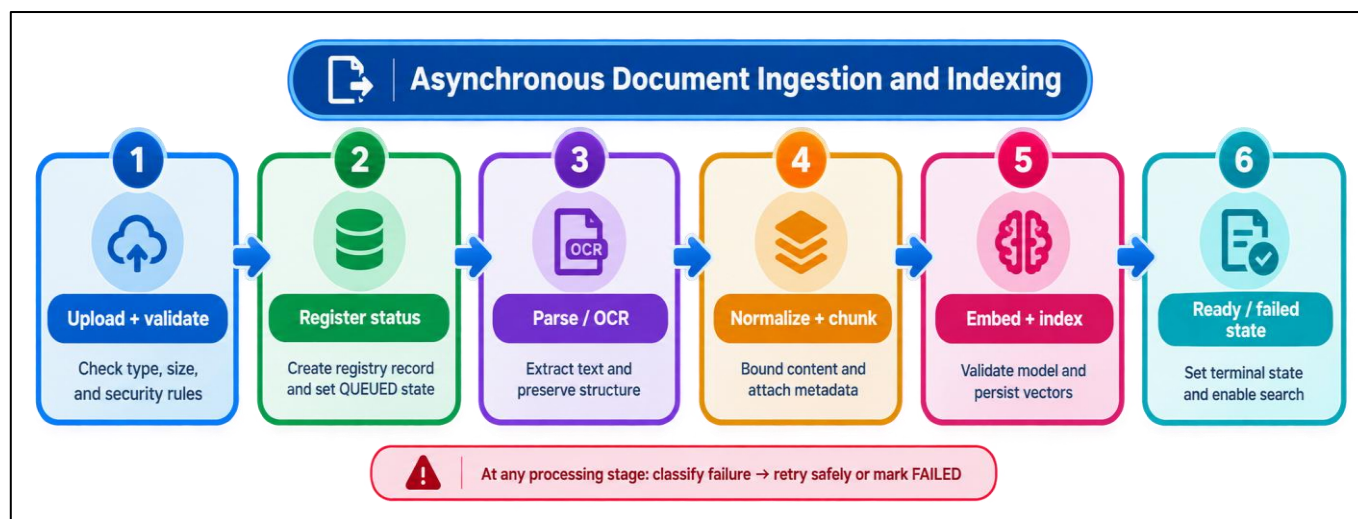


Fig 6 Asynchronous Document Ingestion and Indexing Pipeline.

The figure should show upload validation, durable registration, job scheduling, background extraction/OCR, normalization, chunking, embedding generation, index persistence, completion verification, and searchable-state activation. Controlled failure and retry paths should be represented separately from the successful processing path.

➤ *Parsing and Optical Character Recognition*

Document parsing converts heterogeneous source files into textual content that can be normalized, indexed, and used by downstream AI agents. Because enterprise repositories contain both digitally generated documents and scanned materials, a reliable ingestion subsystem must support multiple extraction strategies and select them according to the validated characteristics of each file.

The documented platform scope includes common office document formats, PDFs, and image-based documents. The exact supported extensions, parser libraries, and OCR engines should be confirmed against the deployed implementation.

- *Format-Aware Parsing*

Different document formats require different extraction methods.

Digitally generated PDF files may contain selectable text that can be extracted directly. Scanned PDF pages may contain only images and therefore require OCR. Word-processing files may contain paragraphs, tables, headings, and other structured elements. Spreadsheets may contain multiple sheets, formulas, values, and tabular relationships that require format-aware interpretation.

A format-aware parser should preserve useful document structure where feasible. Headings, section boundaries, page identifiers, table associations, and ordered content can improve downstream chunking and source attribution.

The extraction process should not assume that all documents can be accurately converted into a single uninterrupted text string. For example, flattening a multi-column document without respecting reading order may combine unrelated statements. Similarly, extracting table

cells without preserving row and column relationships may change the meaning of the information.

Where the parser cannot preserve important structure, the system should identify the limitation rather than silently treat the extracted text as an exact representation of the source.

- *OCR Processing*

OCR is required when relevant document content is embedded in images rather than represented as directly extractable text.

The OCR workflow may include page rendering, image preprocessing, text recognition, and reconstruction of the recognized text in a suitable reading order.

Image preprocessing can improve recognition under some conditions by correcting orientation, reducing noise, or

enhancing contrast. However, preprocessing choices must be evaluated because aggressive transformations can also remove characters or distort symbols.

OCR quality depends on factors such as scan resolution, font size, page orientation, image compression, language, background interference, and document layout.

Low-quality scans may produce incorrect numbers, missing punctuation, merged words, or misrecognized identifiers. These errors are particularly important in financial, regulatory, and technical documents, where small textual differences can materially change meaning.

When supported by the OCR engine, confidence values can be retained as diagnostic metadata. Such values should be interpreted according to the engine's documented semantics and should not be treated as a universal measure of factual correctness.

Table 3 Document Ingestion Operations and Failure Controls

Operation	Expected Behavior	Failure Handling and Control
Receive upload	Accept a file from an authenticated and authorized user.	Reject unsupported or oversized files; enforce quotas and access permissions.
Validate type and signature	Confirm the detected file format and structural validity.	Reject mismatched extensions, unsafe structures, and unsupported formats.
Register document	Create a durable document identity and initial lifecycle record.	Prevent orphaned processing jobs and recover incomplete registration/scheduling transitions.
Schedule processing	Queue an eligible background job.	Record scheduling failures and support controlled recovery.
Extract text or OCR	Obtain usable textual content from the source document.	Record parser failures, OCR timeouts, encrypted files, and extraction errors.
Normalize and chunk	Produce bounded, ordered text segments with source metadata.	Enforce limits on pages, characters, and chunks; reject invalid output.
Generate embeddings	Convert chunks into vectors using the configured model.	Verify provider identity, dimensions, response completeness, and timeout behavior.
Persist chunks and vectors	Store searchable records associated with the correct document version.	Use transactions, staging, uniqueness rules, or reconciliation to prevent partial and duplicate records.
Verify completion	Confirm required indexing records and metadata consistency.	Keep incomplete or inconsistent versions unavailable for retrieval.
Complete or fail	Expose an accurate terminal or recoverable processing state.	Record error category, attempt count, and authorized retry path.

If confidence information is unavailable, the platform should acknowledge that OCR quality cannot be quantified directly from the extraction output.

- *Hybrid Extraction Strategy*

A document may contain a mixture of digitally extractable text and scanned content. For example, a PDF may include ordinary text pages, scanned annexures, and image-based tables.

A hybrid strategy can select direct text extraction for suitable pages and OCR for pages that require it.

This approach may reduce unnecessary OCR processing while retaining access to image-based content. However, the decision to invoke OCR should consider more than whether a page contains any text at all. A page may

contain a small amount of selectable text while its main content remains embedded in an image.

The implementation should therefore define how it detects pages requiring OCR and how it handles conflicting or duplicate text produced by multiple extraction methods.

- *Secure Parsing and Resource Limits*

Document parsers operate on externally supplied files and should be treated as security-sensitive components.

The ingestion service should account for malformed files, password-protected documents, unsafe embedded content, decompression bombs, oversized page dimensions, deeply nested archive structures, and attempts to exploit parser vulnerabilities.

Where a document format contains embedded archives or file references, processing should prevent path traversal and unauthorized filesystem access.

Parsing should operate within configured resource limits, including maximum execution time, memory consumption where enforceable, extracted character count, and page count.

For higher-risk processing environments, isolation through restricted worker processes or containers can reduce the consequences of a parser failure or exploit. Such isolation must be verified through deployment configuration and testing.

- *Empty and Incomplete Extraction*

Successful parser execution does not necessarily mean that useful content was extracted.

A document may produce no text because it contains unsupported image content, uses an unrecognized encoding, is encrypted, or has an unusual structure.

The pipeline should distinguish an empty document, an extraction failure, and a partially extracted document. These conditions may require different responses.

For example, a document containing no extractable text should not be indexed as though its semantic representation were complete. Where partial extraction is accepted, the platform should record that limitation and define whether the document is eligible for search.

- *Chunking and Metadata Management*

Chunking transforms extracted document content into smaller units suitable for semantic retrieval and language-model context construction.

The purpose of chunking is to balance contextual completeness, retrieval precision, storage requirements, and model-processing limits.

A chunk that is too small may omit the qualifications needed to interpret a statement. A chunk that is too large may contain unrelated material and reduce the usefulness of similarity-based ranking.

- *Chunk Formation*

After extraction, the platform should normalize text using a documented and versioned procedure.

Normalization may include removal of extraction artifacts, correction of excessive whitespace, preservation of meaningful paragraph boundaries, and handling of repeated headers or footers.

Care is required because excessive normalization can change the meaning of the source. For example, removing line breaks from a table or combining unrelated sections may destroy structural relationships.

The normalized text is then divided into chunks according to a configured strategy.

Possible strategies include fixed-length segmentation, token-aware segmentation, paragraph-based segmentation, section-aware segmentation, and controlled overlap between adjacent chunks.

The choice of strategy should depend on the document types and retrieval tasks supported by the application.

For enterprise policy documents, section-aware chunking may help preserve definitions and exceptions. For technical manuals, maintaining relationships between headings, procedures, and explanatory text may be important. For tabular records, ordinary paragraph chunking may be insufficient.

The source material does not establish that one chunking strategy is universally superior. Chunking parameters should therefore be selected and evaluated using representative enterprise documents and questions.

- *Chunk Overlap and Context Preservation*

Chunk overlap allows adjacent chunks to share a limited amount of content.

This can help preserve statements that span chunk boundaries and reduce the chance that a relevant passage is separated from its immediate context.

However, overlap increases storage requirements and may introduce redundant retrieval results. Excessive overlap can cause the same passage to appear repeatedly in the generation context, consuming available context capacity without adding independent evidence.

The overlap configuration should therefore be bounded and evaluated rather than selected arbitrarily.

- *Metadata Requirements*

Each chunk should be associated with metadata sufficient to identify its source, determine its processing configuration, and enforce applicable access restrictions.

Recommended metadata includes:

- ✓ Stable document identifier.
- ✓ Document version or revision identifier.
- ✓ Chunk identifier and sequence number.
- ✓ Page number or section locator where available.
- ✓ Content hash or other integrity reference.
- ✓ Document category and relevant classification attributes.
- ✓ Owner or authorization scope.
- ✓ Extraction and chunking configuration versions.
- ✓ Embedding-provider and model identity.
- ✓ Embedding configuration or index version.
- ✓ Creation and update timestamps.
- ✓ Processing or index availability status where required.

The exact schema should be adapted to the implementation, but the metadata must preserve the relationships needed for source attribution, deletion, reprocessing, and retrieval filtering.

- *Stable Identity and Deduplication*

Chunk identifiers should remain unambiguous within the document and processing version to which they belong.

A content hash can help identify repeated or unchanged text, but identical text does not necessarily represent the same logical source location. For example, a standard disclaimer may appear in multiple documents, each with different ownership and retention requirements.

Consequently, content deduplication should not remove the ability to identify the original document and its access-control context.

The platform should distinguish physical content reuse from logical source identity. If storage optimization allows multiple documents to reference identical content, authorization and deletion rules must still apply independently to each source.

- *Versioned Chunking Configuration*

Chunking configuration influences the text supplied to the embedding model and therefore affects retrieval behavior.

Changes to chunk size, overlap, normalization, parser behavior, or section-boundary detection can produce different chunks even when the source document remains unchanged.

The platform should record the configuration used to generate each indexed version.

When a configuration change makes existing chunks incompatible with the intended retrieval behavior, the system should perform a controlled reprocessing or re-indexing operation.

Versioning supports reproducibility and helps explain why search results may differ between application releases.

➤ *Embedding Generation and Vector Indexing*

Embedding generation converts document chunks into numerical representations that can be compared during semantic retrieval.

Each chunk is submitted to a configured embedding model, which returns a vector representation. The vector is then stored in a compatible index together with the chunk identifier and metadata required for retrieval.

The documented architecture describes a 768-dimensional embedding configuration and cosine-distance retrieval. These are configuration details that must be verified against the actual embedding endpoint, database extension, and index implementation before deployment.

- *Embedding Model Configuration*

An embedding model should be identified by more than its output dimension.

The configuration should record the provider, model name, version where available, vector dimension, preprocessing behavior, and intended similarity metric.

These values collectively define the embedding contract.

Two models can produce vectors with the same number of dimensions while representing semantic relationships differently. Their outputs should not be mixed in a common retrieval space unless compatibility has been independently established.

For this reason, the system should associate each index or collection with the embedding configuration used to populate it.

- *Embedding Generation Workflow*

For each eligible chunk, the embedding service submits the normalized chunk text to the configured model.

The returned representation should be validated before persistence.

Validation should confirm that the response contains the expected number of vector elements, that the elements are valid finite numeric values, and that the response corresponds to the intended model configuration.

The service should also reject malformed or incomplete provider responses.

Batch processing may improve throughput where supported by the provider. However, batch size should respect provider limits, available memory, execution time, and error-handling requirements.

A failed batch should not automatically cause previously completed chunks to be duplicated. The system should maintain sufficient processing state to retry incomplete work safely.

- *Vector Storage and Retrieval*

After successful generation, the vector is stored in a compatible index and associated with its source chunk.

The index must support the configured distance operation and preserve the relationship between vectors, source identifiers, and access-control metadata.

Cosine-based retrieval ranks candidate vectors according to their angular relationship. The application must verify the database operator's exact distance semantics and ordering requirements.

A smaller cosine-distance value commonly indicates a closer match, but the implementation should not assume that every vector service uses the same scoring convention.

The retrieval service should apply the required document permissions and availability checks before content is exposed to an agent or generation model.

- *Index Compatibility*

The system should validate embedding compatibility at both write time and query time.

At write time, vectors must match the configuration associated with the target index.

At query time, the user query must be embedded using a model compatible with the stored vectors.

If the configured embedding provider is unavailable, the platform should not automatically substitute an unrelated model and continue searching the existing index.

Instead, the application should use an explicitly compatible provider, select an index created for the alternative model, or return a controlled unavailability response.

This requirement prevents apparently valid but semantically unreliable similarity comparisons.

- *Embedding Model Migration*

Embedding-model upgrades may require a complete or selective re-indexing process.

A controlled migration can create a new index or collection associated with the replacement model, regenerate embeddings from the authoritative source chunks, validate the new index, and then switch retrieval traffic to the new configuration.

The previous index should remain available until the new index satisfies the defined readiness conditions, subject to retention and operational policies.

The cutover process should avoid mixing vectors from incompatible models and should preserve the ability to identify which index version produced a particular retrieval result.

Where a rollback is required, the application should have a documented procedure for restoring the previous compatible retrieval configuration.

- *Index Verification and Evaluation*

Successful vector insertion does not establish that semantic retrieval is accurate.

Index verification should include model and dimension checks, representative similarity queries, metadata filtering, source attribution, and authorization behavior.

Retrieval effectiveness should be evaluated using a reviewed dataset of enterprise questions and relevant source passages.

Depending on the evaluation design, metrics may include precision at (k), recall at (k), mean reciprocal rank, and ranking-quality measures.

Latency and resource utilization should be measured separately from relevance quality.

The target deployment must also verify that the required vector-search extension or service is actually installed and configured. PostgreSQL availability alone does not guarantee that vector indexing is operational.

➤ *Document Lifecycle, Deletion, and Reconciliation*

Document lifecycle management ensures that the platform maintains a consistent relationship between original source files, registry records, extracted content, chunks, and vector representations.

Because these records may reside in different storage systems, lifecycle operations require explicit coordination.

A successful relational database transaction does not automatically guarantee that corresponding vector-index or file-storage operations have completed.

- *Lifecycle Consistency*

The document registry should provide an authoritative record of the document's logical state and active version.

The retrieval service should use this state, together with authorization metadata, to determine whether indexed content is eligible for search.

Documents that are pending processing, failed, deleted, or superseded should not be returned as active content unless the application explicitly defines and authorizes a different behavior.

A document should not be marked INDEXED until the required records have been verified.

Where a new version is being prepared, the previous valid version may remain searchable until the replacement is ready, provided that the application's retention and access policies permit this behavior.

- *Document Deletion*

Deletion may involve several distinct operations:

- ✓ Marking the document unavailable for retrieval.
- ✓ Removing or invalidating associated chunk records.
- ✓ Removing or invalidating vector representations.
- ✓ Deleting the original source file where required.
- ✓ Removing or retaining operational metadata according to policy.
- ✓ Recording the deletion outcome in an appropriate audit trail.

The sequence should prioritize preventing deleted or unauthorized content from being returned.

For systems in which physical deletion across multiple stores cannot occur atomically, a logical deletion or tombstone state can immediately prevent retrieval while cleanup proceeds asynchronously.

The system should not claim that deletion is complete until the relevant deletion requirements have been satisfied.

Retention obligations, legal holds, and backup policies may affect when physical removal is permitted or possible. These conditions should be documented for the target deployment.

- *Document Reprocessing*

Reprocessing may be required when a source document changes, extraction quality is improved, chunking configuration changes, or an embedding model is replaced.

The reprocessing workflow should identify the source version and processing configuration used to create the new derived records.

New chunks and embeddings should preferably be prepared without prematurely invalidating the currently active version.

After successful verification, the platform can activate the replacement version and retire the previous index records according to policy.

If reprocessing fails, the system should preserve a consistent active state rather than exposing a mixture of old and new chunks as one document version.

- *Reconciliation Across Storage Systems*

Reconciliation detects and resolves inconsistencies between the document registry, source storage, extracted-content records, and vector index.

Potential inconsistencies include:

- ✓ A registry record marked INDEXED without the expected vector records.
- ✓ Vector records referencing a deleted or nonexistent document.
- ✓ Duplicate chunks produced by repeated processing.
- ✓ Missing source files for registered documents.
- ✓ Chunks associated with an obsolete processing version.
- ✓ Embeddings stored under an incorrect model configuration.
- ✓ Incomplete cleanup after a deletion or migration.
- ✓ Documents remaining in a processing state after an interrupted job.

A reconciliation process should compare authoritative identifiers and expected record relationships across the relevant stores.

Detected inconsistencies should be classified before repair. For example, missing vectors may require re-indexing, while orphaned vectors may require removal after verifying that they are not referenced by an active document version.

Automated repair should operate within defined safety constraints and should avoid deleting data solely because a transient service failure made a record temporarily inaccessible.

- *Auditability and Recovery*

Lifecycle operations should generate sufficient operational records to support troubleshooting and audit review.

Useful fields include the document identifier, source version, operation type, processing attempt, timestamp, outcome, error category, and correlation identifier.

Logs should avoid unnecessarily retaining document content or sensitive metadata.

Recovery procedures should distinguish transient processing failures from persistent data-integrity problems.

For example, a temporary embedding-provider outage may be resolved through a bounded retry, whereas an inconsistent index may require reconciliation or rebuilding from authoritative source content.

Production readiness should be assessed through controlled tests of deletion, replacement, interrupted processing, duplicate job delivery, index failure, and recovery after partial completion.

- *Engineering Evaluation Considerations*

The ingestion subsystem should be evaluated using both functional and operational criteria.

Functional evaluation should verify that supported documents can be validated, parsed, chunked, embedded, indexed, retrieved, reprocessed, and deleted according to the defined lifecycle.

Operational evaluation should measure processing duration, throughput, queue delay, failure rates, retry behavior, resource consumption, and recovery effectiveness under representative workloads.

The evaluation should include difficult inputs such as scanned documents, malformed files, large documents within configured limits, empty files, mixed-content PDFs, and interrupted provider operations.

Importantly, successful automated tests demonstrate only the behaviors covered by those tests. They do not independently establish OCR accuracy, retrieval relevance, scalability, or resistance to every malicious file type.

The effectiveness of the ingestion design must therefore be reported using the actual implementation evidence, test coverage, and measured results available for the platform.

➤ *Summary of Ingestion Design Principles*

The document-ingestion subsystem of PIU Smart Intelligence Agents is intended to provide a controlled transformation from enterprise source files into authorized, traceable, and semantically searchable content.

Its principal design requirements are the separation of upload acknowledgement from indexing completion, format-aware extraction, bounded resource consumption, explicit processing states, idempotent retries, versioned chunk and embedding configurations, compatible vector indexing, and lifecycle consistency across storage systems.

These requirements establish the engineering foundation for downstream RAG and document-oriented agents. However, the reliability of the overall system depends on verifying the implemented processing paths and evaluating the quality of the resulting searchable content.

A document should be considered ready for enterprise retrieval only when its required processing stages have completed, its index records are consistent, and its access restrictions can be enforced at retrieval time.

VI. RETRIEVAL-AUGMENTED GENERATION AND DOCUMENT INTELLIGENCE

➤ *RAG Query Processing Pipeline*

Retrieval-Augmented Generation (RAG) is a core information-access capability of PIU Smart Intelligence Agents. It combines semantic retrieval with large language model (LLM)-based generation to answer questions using relevant information extracted from authorized enterprise documents. Instead of relying exclusively on knowledge encoded in model parameters, the RAG workflow retrieves potentially relevant passages from an indexed document collection and supplies them as contextual evidence to the generation model.

The query-processing pipeline begins when an authenticated user submits a natural-language question. The application validates the request and determines the user's authorized access scope. These checks establish which documents and resources may be used to answer the question. Authentication and authorization must remain application-enforced controls; they should not depend on instructions given to the language model.

The validated question is transformed into a vector representation using an embedding model compatible with the target document index. The retrieval service uses this representation to identify candidate passages from the indexed corpus. Depending on the configured retrieval strategy, candidate selection may use vector similarity, lexical matching, metadata filters, or a combination of these techniques.

The candidate passages are then checked against document availability, ownership, access permissions, and other applicable metadata constraints. This stage prevents unauthorized or inactive content from being supplied to the generation model. The remaining candidates are ranked and selected according to the retrieval configuration and the requirements of the question.

The context-assembly component combines the selected passages with the user's question and the instructions required for response generation. Context construction should preserve the relationship between each passage and its source document while respecting the model's context-window and resource limits. Relevant page numbers, section identifiers, chunk identifiers, and document versions should be retained where available.

The generation model receives the question and selected evidence and produces a response according to the application's response policy. The application should then validate the response structure, preserve supporting source references, and apply any required output or disclosure checks before presenting the answer to the user.

If relevant evidence cannot be found, the system should distinguish this outcome from a retrieval timeout, database failure, or generation-provider error. An empty search result indicates that the configured retrieval process did not identify suitable evidence; it does not necessarily prove that the information is absent from the entire enterprise repository.

The overall pipeline therefore consists of identity and request validation, query embedding, candidate retrieval, authorization and metadata filtering, ranking, bounded context assembly, answer generation, response validation, and citation presentation. Each stage should expose sufficient operational information to support troubleshooting and evaluation without unnecessarily recording sensitive document content.

The figure should illustrate the complete sequence from user question submission through authorization, embedding, retrieval, filtering, ranking, context assembly, generation, output validation, and evidence-linked response presentation. It should also show controlled paths for insufficient evidence, retrieval failure, and provider unavailability.

➤ *Retrieval and Ranking*

Retrieval determines which document passages are available to support the generated response. Its effectiveness depends on several interacting factors, including query representation, document extraction quality, chunking strategy, embedding-model compatibility, index configuration, metadata filtering, candidate-pool size, and ranking behavior.

Semantic retrieval compares the vector representation of a query with the stored vectors associated with document chunks. This approach can identify conceptually related passages even when the user's wording differs from the wording in the source document. However, semantic

similarity is not equivalent to factual relevance. A passage may be topically related to a question while failing to provide the specific information required to answer it.

Lexical retrieval can complement vector search when exact identifiers, policy numbers, abbreviations, technical terms, or other precise expressions are important. A hybrid strategy may combine lexical and semantic candidates before applying a ranking procedure. Whether this approach improves results depends on the corpus, query types, ranking configuration, and evaluation data; improvement should not be assumed without measurement.

The candidate-pool size influences the balance between retrieval coverage and processing cost. A small candidate pool may omit relevant passages, while a larger pool may introduce redundant or weakly related content. The ranking stage should prioritize evidence that directly addresses the question and preserve sufficient context for correct interpretation.

Relevance thresholds can be used to exclude weak matches or trigger an insufficient-evidence response. However, an overly restrictive threshold may suppress useful passages, particularly for short queries, uncommon terminology, abbreviations, scanned documents, or specialized enterprise vocabulary. Threshold values should therefore be treated as configurable parameters and evaluated against representative questions with reviewed relevance labels.

Metadata filtering is also important. Document category, department, version, ownership, and other attributes can improve retrieval precision when they are correctly populated and applicable to the request. Nevertheless, metadata filters should not be confused with authorization controls. A search preference may narrow the candidate set, but permission enforcement must independently prevent access to restricted content.

Retrieval quality should be evaluated using a labeled dataset that identifies relevant source documents or passages for representative queries. Measures such as Recall@ (k) , Precision@ (k) , Mean Reciprocal Rank (MRR), and normalized Discounted Cumulative Gain (nDCG) can provide complementary views of retrieval coverage and ranking quality.

Failure analysis should include queries that involve short or ambiguous wording, exact identifiers, table-based information, scanned records, conflicting document versions, and terminology that differs from the original source. The purpose is to identify whether retrieval errors originate in extraction, query representation, candidate generation, metadata filtering, or ranking.

For PIU Smart Intelligence Agents, the retrieval configuration should be versioned and its evaluation conditions documented. This makes it possible to compare changes to chunking, embedding models, candidate limits, ranking strategies, and relevance thresholds without

confusing configuration changes with improvements in the generation model.

➤ *Context Assembly and Evidence-Grounded Answers*

After retrieval and ranking, the context-assembly stage determines which passages are supplied to the generation model. This stage is important because retrieved evidence may be relevant but incomplete, redundant, outdated, contradictory, or insufficient to answer every part of a question.

The context should include the user's validated question, selected evidence passages, source identifiers, and instructions that define the expected response behavior. Only the information necessary for the task should be included. Context size should be bounded to control latency, token consumption, and resource use.

Retrieved passages must be treated as untrusted content. Enterprise documents may contain instructions, examples, or malicious text that attempts to influence the model's behavior. The generation model should be instructed to use retrieved passages as evidence rather than as instructions that override system policies, access restrictions, or the authorized task. More importantly, security enforcement must remain outside the model's discretion: the application must validate tool calls, access permissions, and sensitive operations independently.

Context assembly should preserve the relationship between the text and its source. Useful source metadata may include document title, document identifier, version, page number, section heading, and chunk identifier. This metadata supports traceability and allows a reviewer to locate the underlying content.

The generation model should produce answers that are supported by the available evidence. Where appropriate, the response should distinguish directly stated facts from interpretation or inference. If retrieved passages do not adequately support the requested conclusion, the system should identify the evidence gap rather than invent missing details.

Conflicting evidence requires particular care. If two authorized documents contain different values, policies, or dates, the system should not silently merge them into a single apparently consistent statement. Instead, it should preserve the relevant source references and identify the conflict where it materially affects the answer. Document version, effective date, and authority may help explain the discrepancy when that information is available.

Citation generation should associate relevant answer claims with the passages that support them. A citation may refer to a source document, page, section, or chunk, depending on the available metadata. If a source locator is unavailable, the system should not invent one.

The presence of a citation marker does not prove that the associated claim is correct. Citation validation should

examine whether the cited passage actually supports the statement and whether the reference points to the intended source. Answer quality should therefore be evaluated separately from citation formatting.

The application should also distinguish between several different outcomes:

- Evidence found: relevant passages are available to support an answer.
- Insufficient evidence: retrieval completed, but the selected passages do not adequately support the requested conclusion.
- No matching passages: the retrieval process returned no suitable candidates under the configured search conditions.
- Retrieval failure: the search service or underlying index could not complete the operation.
- Generation failure: the evidence was retrieved, but the configured model could not produce the response.

These outcomes require different user messages and recovery actions. For example, a retrieval timeout may justify a controlled retry, whereas insufficient evidence may require a narrower question or an additional authorized source.

Finally, the generated answer should pass any applicable output validation before being presented. This may include checking response structure, ensuring that source references remain attached, enforcing disclosure policies, and preventing unauthorized information from being included in the final response.

➤ *Document Summary Agent*

The Document Summary Agent provides a document-oriented capability within PIU Smart Intelligence Agents. Its purpose is to transform authorized source content into a concise and organized representation while preserving the information necessary for correct interpretation and subsequent review.

The summarization workflow begins by identifying the source document and verifying that the requesting user is authorized to access it. The system then obtains the document's extracted text or relevant passages, together with available metadata such as title, version, page references, and extraction status.

Before summarization, the application should determine whether the source content is sufficiently complete and usable. A document with failed OCR, missing pages, incomplete extraction, or unreadable sections may not support a reliable comprehensive summary. Such limitations should be communicated when they materially affect the result.

The summarization model receives the authorized content and instructions defining the intended output. Depending on the user's request, the result may be an executive summary, section-by-section overview, key findings, important dates, obligations, risks, decisions, or

action items. The output structure should reflect the purpose of the summary rather than applying the same format to every document type.

A reliable summary should preserve the source's central claims, qualifications, conditions, exceptions, limitations, and material dates. It should not convert uncertain statements into established facts, omit restrictions that change a policy's meaning, or combine conflicting statements without explanation.

Where the source contains contradictory information, the summary should identify the conflict and preserve the relevant source locations when available. The agent should not silently decide which statement is authoritative unless an approved rule or reliable source metadata supports that decision.

Long documents may exceed the context capacity of a generation model. A hierarchical summarization strategy can address this limitation by summarizing individual sections or groups of passages and then combining the intermediate summaries into a document-level overview.

However, intermediate summaries can omit details or introduce errors. Consequently, the final summary should remain traceable to the original document, and important claims should be checked against the source text where practical. Intermediate summaries should not become the sole evidence for high-impact statements when the original passages remain available.

The agent should also distinguish source facts from its own explanatory interpretation. Where a requested conclusion cannot be supported by the available document, the response should identify the limitation instead of filling the gap with invented information.

The output should identify the source document and preserve available page or section references. If extraction quality is uncertain or only part of the document was processed, the summary should indicate that it is based on incomplete or potentially unreliable content.

Document summaries must also respect authorization throughout processing. A user should not be able to summarize a restricted document merely by supplying its identifier or asking another agent to retrieve its contents. Access checks must apply to source selection, retrieval, context assembly, and result presentation.

The quality of the Document Summary Agent should be assessed through factual consistency, coverage of important information, preservation of caveats, source traceability, and compliance with access restrictions. Readability and conciseness are useful qualities, but they are not sufficient evidence of summary correctness.

➤ *Conversation State and Context Management*

Conversation state enables users to ask follow-up questions without repeating all previously supplied

information. Within PIU Smart Intelligence Agents, conversation history may preserve relevant questions, responses, task context, and source references needed to maintain continuity across a sequence of interactions.

Conversation history must be bounded because retaining every previous message can increase token consumption, latency, and storage requirements. The platform should define limits for message count, context size, retention period, and other applicable resource constraints.

When the conversation exceeds its configured context budget, the system may retain recent messages, summarize older exchanges, or preserve selected task information. These strategies should be applied carefully because history compression can omit important qualifications or make an earlier assumption appear to be an established fact.

A compressed conversation summary should not override the original source evidence, the current user's permissions, or the system's security policies. Where a follow-up question depends on a precise document statement or analytical result, the platform should retrieve or verify the relevant source rather than relying solely on a conversational recollection.

Session identity and ownership are also essential. A conversation identifier must be associated with an authorized user or explicitly authorized shared resource. Knowing or guessing a session identifier must not grant access to another user's messages or derived summaries.

The platform should distinguish conversational context from authorization context. Previous discussion of a document does not automatically grant continuing access to that document if permissions change or the document becomes unavailable. Authorization should be checked at the time the protected resource is accessed.

Retention and deletion policies should apply to stored messages, conversation summaries, derived artifacts, and any other retained conversational records within the defined policy scope. Sensitive information should not be retained indefinitely merely because it may improve future contextual continuity.

Conversation-state handling should also support failure recovery. If a request fails during retrieval or generation, the platform should avoid storing an incomplete response as if it were a completed answer. Where the application records failed attempts, their status should be distinguishable from successful interactions.

Evaluation should examine whether follow-up questions preserve the intended context, whether compressed history retains material constraints, and whether separate users and sessions remain isolated. It should also verify that a change in access permissions is respected even when the conversation previously contained information from the affected resource.

➤ *RAG Evaluation and Reproducibility*

A rigorous evaluation framework is required to determine whether the RAG pipeline retrieves relevant evidence, generates supported answers, preserves source traceability, and behaves reliably under realistic enterprise conditions.

The evaluation set should contain representative user questions, expected source documents, relevant passages, and reference answers where appropriate. The set should cover different document types, query lengths, terminology variations, metadata filters, and retrieval conditions.

Where feasible, relevance labels should be prepared or reviewed by qualified evaluators. The evaluation process should document the source corpus, dataset construction, annotation rules, retrieval configuration, embedding model, chunking parameters, ranking settings, generation model, and test environment.

• *Retrieval Quality*

Retrieval quality measures whether relevant source passages are returned and ranked appropriately.

Recall@ (k) measures the proportion of known relevant items retrieved within the top (k) results, while Precision@ (k) measures the proportion of the top (k) results that are relevant. Mean Reciprocal Rank evaluates the position of the first relevant result, and nDCG evaluates ranking quality while accounting for graded relevance where such labels are available.

These measures provide complementary information. A retrieval system may achieve high recall by returning many passages but still provide poor precision. Conversely, a highly restrictive search may return a small number of relevant passages while missing other necessary evidence.

The selected metrics and evaluation procedure should reflect the retrieval task and the availability of reliable relevance labels.

• *Answer Quality and Evidence Support*

Retrieval metrics alone do not establish that the generated answer is correct. The model may misinterpret retrieved passages, combine statements incorrectly, or introduce unsupported claims.

Answer evaluation should therefore examine factual correctness, relevance to the question, consistency with retrieved evidence, coverage of important conditions, and the handling of insufficient evidence.

Where reference answers are available, responses can be compared against those references using a documented evaluation procedure. Human review may also be necessary for complex policy or business questions where correctness depends on context and interpretation.

- *Citation Quality*

Citation evaluation should determine whether references point to the intended source and whether the cited passages support the claims associated with them.

Useful criteria include source-location accuracy, claim-to-evidence alignment, citation completeness, and the rate of unsupported claims presented with citations.

A correctly formatted citation is not sufficient if the referenced passage does not support the answer. Citation quality should therefore be measured independently of the presence of citation markers.

- *Abstention and Failure Handling*

The system should be evaluated on whether it declines to provide unsupported conclusions when evidence is missing or inadequate.

Abstention quality includes identifying insufficient evidence, distinguishing missing content from retrieval failure, and avoiding confident answers that rely on unrelated passages.

The evaluation should also cover provider timeouts, index unavailability, extraction failures, and malformed outputs. These tests determine whether the application reports the correct failure category rather than silently converting operational errors into apparently valid answers.

- *Performance and Resource Consumption*

End-to-end latency should be measured alongside the durations of query embedding, candidate retrieval, filtering and ranking, context assembly, generation, and response validation.

Measurements should document the query dataset, model configuration, context size, concurrency, hardware or hosting environment, and dependency conditions.

Where appropriate, the evaluation should report median and higher-percentile latency, throughput, error rates, and resource consumption. Results should distinguish model inference time from the total time experienced by the user.

- *Security and Authorization*

Security evaluation should verify that retrieval and generation operate only on information the requesting user is authorized to access.

Tests should include unauthorized document requests, metadata-filter bypass attempts, changes in permissions during a conversation, and retrieved passages containing malicious instructions.

The evaluation should also confirm that provider errors, audit logs, citations, and generated responses do not expose restricted content through unintended paths.

- *Reproducibility and Evidence Limitations*

Reproducible evaluation requires recording the versions of the source corpus, parser configuration, chunking strategy, embedding model, index configuration, retrieval parameters, generation model, and evaluation dataset.

Configuration changes should be documented because differences in chunk boundaries, embedding representations, ranking rules, and generation settings can affect the results.

The available source material does not provide a complete external RAG benchmark or sufficient independently validated measurements to support a claim that PIU Smart Intelligence Agents outperforms alternative retrieval systems. Accordingly, this work does not claim superior retrieval quality based solely on the described architecture.

The contribution is instead the design of an evidence-oriented RAG workflow that integrates authorized retrieval, bounded context construction, source traceability, document summarization, conversation management, and explicit evaluation requirements. Quantitative performance and comparative effectiveness must be established through reproducible experiments using a documented evaluation dataset.

In summary, the RAG and document-intelligence subsystem is intended to connect enterprise information retrieval with controlled language-model generation while preserving authorization and source traceability. Its effectiveness depends on the quality of document extraction, retrieval and ranking, context construction, generation, citation validation, and operational controls. Each component must therefore be evaluated both independently and as part of the complete user-facing workflow.

VII. SPECIALIZED AI AGENTS

PIU Smart Intelligence Agents is designed to provide multiple AI capabilities through a common application platform. Each capability addresses a specific category of enterprise tasks while sharing foundational services such as authentication, authorization, request validation, observability, and controlled access to data sources.

The specialized-agent architecture separates natural-language interpretation from the execution of sensitive operations. A language model may interpret a request or propose an execution plan, but the final operation must be validated by trusted application services before it accesses enterprise data or invokes external tools.

Each agent should have a clearly defined purpose, an explicit set of permitted inputs and tools, a documented output contract, and a controlled failure-handling strategy. This separation supports independent evaluation and reduces the likelihood that an error in one capability will automatically grant access to unrelated platform functions.

➤ *SQL Chat Agent*

The SQL Chat Agent enables authorized users to retrieve and analyze structured enterprise data through natural-language questions. Its primary objective is to translate a business question into a valid, constrained database operation and present the returned results in an understandable format.

Unlike the RAG subsystem, which retrieves passages from document collections, the SQL Chat Agent operates on structured relational data. Its correctness therefore depends on schema interpretation, query construction, database permissions, execution controls, and the accurate explanation of returned records.

• *Request Interpretation and Intent Classification*

The workflow begins when a user submits a natural-language analytical request. The agent identifies the requested metric, dimensions, filters, time interval, comparison basis, and expected result format.

For example, a user may request the total transaction value for a specified reporting period, the number of successful transactions by payment channel, or a comparison of monthly transaction volumes.

The agent should determine whether the request requires aggregation, grouping, filtering, sorting, or a join between authorized tables. If the question is ambiguous—for example, when the meaning of a business term or reporting period is unclear—the application should request clarification rather than silently select an arbitrary interpretation.

• *Schema-Grounded Query Planning*

After interpreting the request, the agent obtains the approved schema context relevant to the operation. This context may contain permitted table names, column names, data types, relationships, and documented business definitions.

The query planner maps the user's requested concepts to fields and operations supported by that schema. It must not invent columns, assume undocumented relationships, or construct joins based solely on similar names.

A query plan may identify the target tables, selected fields, aggregation functions, filters, grouping dimensions, ordering requirements, and result limits. The plan should be validated before it is translated into executable SQL.

Where the requested analysis cannot be expressed using the available schema or approved business definitions, the system should communicate the limitation or request additional information.

• *SQL Generation and Validation*

The SQL-generation component converts the validated plan into a database statement. The generated statement must conform to the application's read-only execution policy and the database schema.

Before execution, the platform should inspect the SQL using a parser or equivalent structural validation mechanism. Validation should confirm that the statement uses permitted tables and columns, contains only allowed operations, and satisfies the configured restrictions on joins, filters, aggregation, sorting, and result size.

Simple keyword checks alone are insufficient because SQL syntax can contain nested expressions, comments, common table expressions, subqueries, and other structures that make textual validation unreliable.

The system should reject multiple statements and disallowed operations, including data modification and schema-changing statements. It should also verify that the final executable statement is the same statement that passed validation.

• *Restricted Database Execution*

Validated SQL should execute using a dedicated database role with only the permissions necessary for the supported analytical tasks. Where practical, the role should have read access only to approved views or tables.

Application-level validation is an additional safeguard; it does not replace database permissions.

Execution should be subject to statement timeouts, maximum row counts, result-size restrictions, and applicable query-cost controls. These limits reduce the risk of expensive or unexpectedly broad queries consuming database resources.

User-provided values should be parameterized rather than concatenated directly into SQL. Dynamic identifiers, such as table and column names, require separate allowlist-based validation because ordinary SQL parameters do not substitute for identifiers in every database interface.

If the query fails, the application should distinguish validation rejection, authorization denial, database execution failure, timeout, and successful execution with no matching rows.

• *Result Processing and Explanation*

After successful execution, the application returns the result set to the presentation layer. The agent may then generate a concise explanation of the returned values, identify relevant trends, or describe the filters and aggregation used.

The explanation must be grounded in the actual database response. It should not invent records, infer unavailable values, or describe a failed query as a successful analysis.

Where appropriate, the response should include the reporting period, metric definition, filters, aggregation method, and other details required to interpret the result.

The platform should preserve sufficient query metadata for audit and debugging purposes while avoiding

unnecessary exposure of sensitive records, credentials, or unrestricted database output.

The SQL Chat Agent should be evaluated for query validity, schema adherence, execution correctness, analytical accuracy, permission enforcement, and safe behavior under adversarial inputs. Generating syntactically valid SQL alone does not establish that the result answers the user's question correctly.

➤ *SQL Defense in Depth*

The SQL Chat Agent introduces risks because natural-language input can influence statements that access enterprise databases. PIU Smart Intelligence Agents should therefore implement multiple independent controls rather than relying on a single prompt, model, or validation function.

- *Read-Only Query Policy*

The application shall permit only explicitly authorized read-only query operations. It should reject data modification, schema changes, privilege changes, and other disallowed statements.

The policy should account for nested queries, multiple statements, database-specific syntax, and indirect execution mechanisms. Validation should be performed against the actual database dialect and the operations supported by the deployment.

A read-only statement may still expose sensitive data or consume excessive resources. Consequently, the read-only policy must be combined with access controls and resource limits.

- *Least-Privilege Database Access*

Database permissions should restrict the SQL execution role to the minimum required resources. Where feasible, the application should expose approved views that omit sensitive columns or enforce business-level restrictions.

The database must independently enforce these permissions. If a generated query contains an unauthorized table reference, the database should deny access even if an application-level check fails.

Separate database credentials should be used for distinct responsibilities when appropriate. Administrative credentials must not be made available to the SQL-generation model or ordinary query-execution path.

- *Schema and Query Validation*

The application should validate all referenced tables, columns, joins, filters, grouping expressions, ordering rules, and result limits against the approved schema and query policy.

The validator should operate on the parsed SQL structure where feasible rather than relying only on regular expressions or keyword matching.

Validation should also enforce business-specific requirements when applicable. For example, a transaction analysis may require a defined success-status filter, while a financial metric may require an approved aggregation rule.

Such rules should be defined by the actual data model and business requirements. They should not be assumed to apply universally to every query.

- *Query Resource Limits*

The execution layer should impose limits on query duration, returned rows, result size, and other relevant resource dimensions.

Where supported, database-level controls may restrict execution time, statement behavior, or access to expensive operations. The application should also prevent repeated retries from amplifying an overloaded database.

The chosen limits should be configurable and validated under representative workloads. A maximum row count alone does not guarantee that a query is inexpensive because the database may scan or aggregate a large number of records before returning a small result.

- *Parameterization and Input Handling*

Values originating from user input should be passed through parameterized database interfaces wherever possible. This reduces the risk of user-controlled values being interpreted as SQL syntax.

Dynamic identifiers and query structures should be selected from approved schemas or allowlists. The application should not permit the language model to insert arbitrary identifiers, functions, or database commands without validation.

Prompts, retrieved documents, and previous agent outputs should be treated as untrusted input. Text that instructs the model to bypass query restrictions must not change the application's execution policy.

- *Auditability and Testing*

The platform should record relevant query-execution metadata, including the operation category, validation outcome, execution status, duration, and correlation identifier. Logging should avoid exposing secrets or unnecessary sensitive values.

Adversarial testing should cover attempts to generate multiple statements, access unauthorized tables, bypass filters, exploit nested SQL structures, request excessive results, and use prompt injection to override the intended policy.

The effectiveness of these controls must be assessed through implementation review and testing. The existence of SQL validation logic does not, by itself, prove that the system is secure against all malicious queries.

➤ *KPI Generator Agent*

The KPI Generator Agent converts business-performance questions into defined, reproducible measurements using authorized enterprise datasets. Its purpose is to calculate or explain key performance indicators while preserving the business rules and reporting context needed to interpret them.

Unlike unrestricted conversational generation, KPI generation should be governed by an approved metric catalogue or equivalent set of explicit business definitions. This reduces ambiguity and prevents the model from inventing formulas or silently changing the meaning of a business indicator.

• *KPI Definition and Resolution*

The workflow begins by identifying the requested KPI and mapping it to an approved metric definition.

A metric definition may specify:

- ✓ Metric name and business purpose.
- ✓ Formula or aggregation rule.
- ✓ Source dataset, table, or approved view.
- ✓ Numerator and denominator, where applicable.
- ✓ Permitted dimensions and filters.
- ✓ Reporting period and time-zone interpretation.
- ✓ Comparison baseline and calculation method.
- ✓ Unit, currency, and rounding rules.
- ✓ Treatment of missing, invalid, or incomplete data.

For example, transaction volume and transaction value are different measures. Volume generally requires a count of qualifying records, whereas value generally requires an appropriate aggregation of a monetary field. The agent must select the approved definition rather than infer the metric solely from the user's wording.

If the requested KPI is not defined or has multiple possible interpretations, the agent should request clarification or flag the metric for business approval.

• *Data Authorization and Calculation Planning*

After resolving the metric, the agent verifies that the user is permitted to access the required dataset and dimensions.

The calculation plan identifies the relevant data source, aggregation, filters, time window, grouping dimensions, and comparison baseline. If the requested calculation requires data that is unavailable or unauthorized, the agent must not substitute an unrelated source without explicit approval.

The plan should be validated against the approved metric definition before execution.

• *KPI Execution and Validation*

The calculation may be performed through a validated SQL query or another approved computation service. Execution should use the same access restrictions, query

limits, and error-handling principles that apply to other analytical operations.

After execution, the system should validate the returned values and relevant calculation conditions. Examples include checking for missing periods, unexpected null values, invalid denominators, incomplete datasets, and differences between the requested and actual reporting windows.

Comparative indicators require explicit baseline handling. Month-over-month, week-over-week, year-over-year, and other comparisons should use defined period boundaries and calculation formulas.

If the baseline is unavailable, the denominator is zero, or the source data is incomplete, the agent should apply an approved handling rule or explain why a reliable value cannot be produced.

• *KPI Presentation and Lineage*

A KPI response should identify the metric, calculated value, unit, reporting period, applied filters, formula or aggregation rule, and source dataset. Where relevant, the response should also include the comparison baseline and assumptions.

The natural-language explanation must reflect the actual calculation results. An estimated value must be clearly distinguished from an observed or directly calculated result.

The agent should preserve sufficient lineage to identify how the reported value was obtained. This may include the approved metric-definition identifier, source query identifier, calculation timestamp, dataset version where available, and relevant filter configuration.

• *KPI Evaluation*

The KPI Generator Agent should be evaluated for metric-definition accuracy, query correctness, arithmetic correctness, temporal consistency, filter application, lineage completeness, and authorization enforcement.

Tests should include undefined metrics, ambiguous reporting periods, missing data, zero denominators, incomplete time series, unauthorized datasets, and conflicting business definitions.

A KPI response should be considered correct only when its value and explanation follow the approved business definition and the actual source data.

➤ *General-Purpose AI Chat Agent*

The General-purpose AI Chat Agent provides a conversational interface for explanations, reasoning, and general assistance within the policies and capabilities configured for PIU Smart Intelligence Agents.

Its primary purpose is to respond to natural-language requests that do not necessarily require direct database execution or document retrieval. Depending on the task, the

agent may explain concepts, help users interpret information, organize ideas, or support multi-turn conversations.

The agent's capabilities depend on the configured model, available context, system instructions, and permitted tools. Its responses should not be represented as authoritative enterprise records unless they are grounded in an approved source.

- *Request Processing and Conversation Context*

The workflow begins by validating the request and obtaining the authorized conversation context. Conversation history should be bounded to control processing cost and reduce the risk of carrying irrelevant or sensitive information into later requests.

The system should distinguish user-provided information from verified enterprise data. If a question depends on an authoritative business value, document clause, or database result, the agent should use an appropriate authorized capability when available or communicate the limitation.

- *Controlled Tool Access*

The General-purpose AI Chat Agent should not automatically inherit the permissions of every other platform agent.

Access to enterprise documents, SQL execution, KPI calculation, or administrative operations should occur through explicitly authorized tools or specialized agents. Each tool call should be validated by application services before execution.

The application should verify the user's permissions, the requested resource, and the arguments supplied to the tool. The model must not be able to grant itself access or modify platform policy by generating a suitable instruction.

- *Response Quality and Limitations*

The agent should produce responses that are relevant to the user's request and consistent with platform policies. It should distinguish established facts from assumptions and acknowledge uncertainty when the available context is insufficient.

The agent may still generate incorrect or unsupported statements. For enterprise-critical questions, the platform should encourage source-backed answers and direct users toward the relevant authorized data or document evidence.

- *Evaluation*

Evaluation should examine response relevance, factual consistency, instruction adherence, context handling, appropriate abstention, and resistance to attempts to invoke unauthorized tools.

Tests should also examine whether the agent incorrectly claims to have executed a database operation, retrieved a document, or verified a business value when no corresponding operation occurred.

➤ *Agent Routing, Contracts, and Result Validation*

The orchestration layer coordinates the specialized agents and determines how a validated user request is processed. Its routing decisions should be based on the task's intent, required resources, user permissions, and the capabilities available in the deployed platform.

Routing may use deterministic rules, model-assisted classification, or a combination of both. Regardless of the selection mechanism, the resulting workflow must remain within the application's approved execution policy.

- *Explicit Agent Contracts*

Each agent should define a documented contract covering its purpose, inputs, outputs, accessible resources, permitted tools, authorization requirements, and failure behavior.

Input contracts should specify required fields, accepted data types, constraints, and handling of ambiguous requests. Output contracts should specify the expected response structure and the conditions under which a result may be considered complete.

These contracts enable the orchestration layer to validate an agent's inputs and outputs consistently and help prevent incompatible components from exchanging malformed or unauthorized data.

- *Authorization-Aware Routing*

The orchestrator should select only agents that are available and authorized for the requested task. The fact that an agent is technically callable does not mean that every user may invoke it or access its underlying data.

Authorization should be enforced both when selecting a capability and when the capability accesses protected resources. This is particularly important when one agent requests another agent's services or when an agent attempts to invoke a database or document-retrieval tool.

An agent must not gain additional privileges simply because another agent or the orchestration model requests them.

- *Output Validation*

Agent outputs should be validated before they are presented to the user or consumed by another agent.

Validation may include schema conformance, execution status, source-reference integrity, data completeness, policy compliance, and consistency with the original request.

For example, a downstream explanation agent should not treat a failed SQL operation as a valid analytical result. Similarly, a summary agent should not claim to have processed a complete document if only a subset of the source was extracted.

- *Multi-Agent Workflow Integrity*

When a task involves multiple agents, the workflow should preserve the provenance of intermediate results. The system should be able to identify which agent performed each operation, which approved tool was invoked, and which data or evidence supported the final response.

Intermediate outputs should be treated according to their trust level and validated before reuse. A model-generated explanation should not automatically become authoritative evidence for another agent merely because it was produced earlier in the same workflow.

The orchestration layer should also define limits on the number of steps, execution duration, retries, and resource consumption. These limits reduce the risk of uncontrolled loops and excessive provider or database usage.

- *Observability and Evaluation*

Each workflow should use a correlation identifier to connect relevant routing decisions, agent executions, tool calls, validation outcomes, and final responses.

Operational records should capture useful metadata without unnecessarily storing complete prompts, sensitive documents, or confidential query results.

Agent routing should be evaluated for correct capability selection, unauthorized-route rejection, failure handling, and preservation of source lineage. Output validation should be tested using malformed responses, incomplete results, conflicting intermediate outputs, and adversarial instructions.

➤ *Result Handling and Residual Risk*

The result-handling layer converts internal agent outcomes into consistent user-facing responses. Its purpose is to ensure that the platform communicates accurately whether an operation succeeded, returned no matching data, failed validation, was denied, or could not complete because of an unavailable dependency.

At a minimum, the application should distinguish the following outcomes:

- Invalid request: required fields or request parameters do not satisfy the expected contract.
- Authorization denial: the user is not permitted to perform the requested operation or access the resource.
- Validation rejection: the generated query, tool arguments, or intermediate output violates an application policy.
- Empty result: the operation completed successfully but returned no matching records or relevant passages.
- Timeout: the operation did not complete within its configured execution window.
- Provider failure: a required model or processing dependency was unavailable or returned an unusable response.
- Execution failure: a database, parsing, indexing, or other internal operation failed.

- Successful result: the operation completed and passed the required validation checks.

The natural-language explanation should be derived from the actual execution result. The platform must not report that a query was executed when it was only generated, or claim that a document was retrieved when the retrieval operation failed.

Partial results should be identified explicitly. For example, an analytical response based on incomplete data should not be presented as a complete reporting-period total without an appropriate qualification.

Prompting and static validation can reduce risk, but neither guarantees safe execution. SQL parsing, least-privilege database permissions, argument validation, execution isolation, resource limits, and adversarial testing remain necessary.

The residual risks include model hallucination, ambiguous business terminology, incomplete source data, incorrect metric definitions, retrieval errors, malicious document content, provider incompatibility, and failures in authorization or result validation. These risks should be documented and assessed through targeted testing rather than treated as fully resolved by the multi-agent architecture.

VIII. SECURITY ANALYSIS

Security is a cross-cutting requirement of PIU Smart Intelligence Agents because the platform processes enterprise documents, structured business data, conversational context, and model-generated operations. A weakness in any of these paths may affect confidentiality, integrity, availability, or auditability.

The security design therefore combines identity verification, resource-level authorization, safe document processing, controlled tool execution, protected configuration, API and infrastructure safeguards, and operational monitoring. No single control is sufficient to protect every workflow. The controls should be applied at the boundaries where data is received, processed, stored, retrieved, and returned to the user.

➤ *Authentication and Authorization*

Authentication establishes the identity associated with a user or service request. Authorization determines whether that identity is permitted to perform a specific operation on a particular resource.

PIU Smart Intelligence Agents should keep these responsibilities separate. Successful authentication must not automatically grant access to all documents, database tables, agents, conversation records, provider settings, or administrative functions.

- *Identity and Session Validation*

Protected API operations should require an authenticated identity or an explicitly authorized service

identity. The application should validate the authentication mechanism and apply the configured session or token rules.

Where sessions or tokens are used, the implementation should define expiration, revocation, renewal, and logout behavior as appropriate to the selected authentication architecture.

A request should not be trusted solely because it originates from the platform's user interface. Direct API calls must be subject to the same authentication and authorization requirements as requests submitted through the interface.

- *Role- and Resource-Aware Authorization*

Role-based permissions can determine which categories of operations a user may perform. Resource-level checks determine whether the user may access a particular document, conversation, dataset, or administrative object.

Both levels may be necessary. For example, a user may be permitted to use the Document Summary Agent but not be authorized to summarize every document in the repository.

Authorization checks should apply to document retrieval, SQL execution, KPI calculation, exports, conversation access, and administrative operations. The system should also validate the permissions required for any tool invoked by an agent.

- *Background-Job Authorization*

Authorization must extend beyond interactive API requests. A background worker should process only jobs created through an authorized workflow and should retain the relevant ownership and resource context.

The worker should not treat a document identifier or queued job message as sufficient proof of permission. It should validate the associated record and apply the appropriate policy before accessing the source content or changing its state.

- *Authorization Consistency*

Access policies should be enforced consistently across API routes, business services, agent tools, database access, vector retrieval, exports, and background processing.

When permissions change, the system should apply the updated policy to subsequent protected operations. Previously stored conversation history or cached retrieval results must not be used to bypass current authorization.

- *Verification*

Authorization testing should include direct endpoint access, attempts to access another user's resources, unauthorized agent invocation, document-ownership changes, export restrictions, and background-job processing.

The tests should verify both permitted and denied operations. A security design should not be considered verified merely because unauthorized requests are rejected in the user interface.

- *Upload and Content Security*

Document ingestion accepts external files and therefore introduces risks associated with malformed content, parser vulnerabilities, resource exhaustion, and unsafe embedded structures.

The platform should apply layered validation before a file is stored, parsed, or indexed.

- *File Validation*

Upload validation should consider the declared extension, detected MIME type, file signature, structural validity, and configured size limits. These checks should be combined because no single property reliably establishes that a file is safe.

The application should reject unsupported formats and files that violate processing limits. It should also enforce user quotas or equivalent controls to prevent uncontrolled upload volume.

Filenames should be treated as untrusted metadata. Storage paths should be generated or normalized safely rather than constructed directly from user-supplied filenames.

- *Archive and Parser Protection*

If the application processes archive-based document formats, it should account for path traversal, excessive decompression ratios, deeply nested archives, and resource-exhaustion attacks.

XML-based formats should be processed using safe parser settings that disable unnecessary external entity resolution and other unsafe features. The exact configuration should depend on the parser library and document format.

Document conversion, OCR, and parsing should operate within defined resource limits. Where feasible, processing workers should use restricted filesystem access, limited privileges, and isolation from unrelated application services.

- *OCR and Conversion Safety*

OCR and conversion operations may process large or malformed images and documents. The application should impose appropriate timeouts, page limits, image-dimension limits, and memory or resource constraints where supported.

A processing failure should not expose internal filesystem paths, stack traces, or sensitive configuration to the user.

The system should also ensure that an OCR or conversion error does not result in incomplete content being silently marked as a complete searchable representation.

- *Safe Storage and Lifecycle Management*

Original files and derived document content should be stored under controlled access policies. Uploaded files should not be placed in executable application paths or made publicly accessible by default.

Deletion and replacement workflows should prevent stale derived content from remaining available through semantic retrieval after the source is no longer authorized or active.

- *Verification*

Testing should include malformed files, unsupported formats, oversized uploads, archive-related attacks, password-protected documents, invalid XML structures, OCR timeouts, and attempts to access files through manipulated identifiers or paths.

The results should document the tested formats, configured limits, parser versions, and relevant deployment conditions.

➤ *Prompt-Injection Resistance*

Prompt injection is a risk in systems that pass user-controlled or retrieved text to language models. An attacker may place instructions in a user prompt, document, table, OCR output, or retrieved passage that attempt to override the intended task, reveal confidential information, or invoke unauthorized tools.

PIU Smart Intelligence Agents should treat external and retrieved content as untrusted input. The platform must distinguish between instructions established by trusted application policy and text that merely appears within a document or user-provided message.

- *Direct and Indirect Prompt Injection*

Direct prompt injection occurs when a user submits instructions intended to bypass restrictions. Indirect prompt injection occurs when malicious instructions are embedded in content that an agent later retrieves or processes.

Examples include a document that instructs a model to disclose other users' data, a table cell that requests an unauthorized tool invocation, or OCR text that attempts to change the model's assigned role.

The application should test these attack paths across the relevant agents rather than limiting testing to ordinary chat prompts.

- *Trust Boundaries and Context Delimitation*

Retrieved passages should be clearly separated from trusted system instructions in the generation context. The model should be instructed to use the passages as evidence rather than as instructions that redefine the task or security policy.

However, delimiters and model instructions are only supporting safeguards. They cannot guarantee that a model will ignore every malicious instruction.

Security-critical decisions must therefore remain outside the model. Access permissions, allowed tool calls, database restrictions, export permissions, and administrative actions should be enforced by application code and the relevant underlying services.

- *Tool Invocation Controls*

Each agent should have an explicit tool allowlist defining which operations it may request. Tool arguments must be validated before execution, and the application should verify the requesting user's permissions for the target resource.

The model should not be able to introduce arbitrary tool names, modify authorization rules, access unrestricted credentials, or execute administrative operations outside its approved scope.

Where a workflow involves multiple agents, intermediate outputs should be treated as potentially untrusted until they have passed the required validation checks.

- *Data Exfiltration and Cross-Resource Access*

Prompt-injection testing should examine attempts to retrieve unrelated documents, disclose restricted database results, expose secrets, or combine information from resources the user is not authorized to access.

The application should minimize the context supplied to each agent and should enforce authorization at the data-access boundary.

A model response should not be allowed to reveal confidential information merely because the information was accidentally included in a previous conversation or intermediate agent output.

- *Verification and Residual Risk*

Adversarial tests should include malicious instructions in document paragraphs, tables, OCR-derived text, retrieved snippets, and multi-agent handoffs.

The evaluation should verify that such content cannot bypass authorization, expand tool permissions, or cause restricted operations to execute.

Prompt-injection defenses reduce risk but cannot establish immunity to all future attack techniques. High-assurance deployments require continuing testing, monitoring, and review of changes to models, tools, retrieval strategies, and source-processing components.

➤ *Secrets, Privacy, and Data Minimization*

The platform may process confidential business documents, analytical results, conversational content, provider credentials, and database connection information. Security therefore requires careful control over secret storage, data transmission, logging, and retention.

- *Secret Management*

API keys, database passwords, authentication secrets, and other credentials should not be hard-coded into application source files or committed to version control.

Secrets should be supplied through protected configuration or a suitable secret-management mechanism.

Access should be restricted to the services that require each secret, and credentials should be rotated according to the deployment's requirements.

Secrets must not be returned in API responses, embedded in model prompts unnecessarily, or written to ordinary application logs.

- *Data Minimization*

The platform should send only the context required for the requested operation to a language-model provider or other external service.

For example, a document-summary request should not automatically include unrelated conversation records or other enterprise documents. A KPI calculation should use the required metric definition and authorized data rather than transferring unrestricted database contents into a generation prompt.

Minimization reduces unnecessary exposure and can also reduce processing cost and latency.

- *Logging and Retention*

Prompt, response, document, and query logging should be governed by an explicit retention policy. Operational monitoring should retain enough metadata to support troubleshooting and security review without unnecessarily storing full confidential content.

Where practical, logs should use identifiers, event categories, outcome codes, and correlation identifiers rather than unrestricted copies of sensitive inputs and outputs.

Access to audit and diagnostic records should itself be controlled and monitored. Retention periods should reflect operational needs, organizational policies, and applicable legal obligations.

- *External Provider Review*

If source text, prompts, database-derived information, or conversation context is transmitted to an external model gateway, the deployment must review the provider's applicable contractual terms, data-handling practices, retention settings, processing location, and relevant privacy requirements.

The use of an external provider should be an explicit deployment decision rather than an undocumented side effect of a default configuration.

Where data-residency or confidentiality requirements prohibit a particular transfer, the application should use an approved alternative or disable the affected operation.

- *Verification*

Secret-handling tests should examine source repositories, build artifacts, configuration files, logs, error responses, and provider requests for unintended credential exposure.

Privacy testing should assess whether unnecessary document content or conversation history is included in external requests and whether retained data follows the configured deletion and retention rules.

The actual privacy properties depend on the selected providers, deployment configuration, and operational procedures and should be documented accordingly.

- *API and Infrastructure Security*

API and infrastructure controls protect the components through which users, agents, databases, storage services, and external providers communicate.

These controls complement application-level authorization and safe tool execution. They do not replace them.

- *API Boundary Controls*

The API layer should validate request schemas, enforce request-size limits, and return consistent error responses. Rate limits or equivalent resource controls should be applied where necessary to prevent abuse and excessive workload.

Cross-Origin Resource Sharing (CORS) should allow only the origins required by the deployment. Security headers should be configured according to the application's browser-facing requirements.

Error responses should not disclose stack traces, internal paths, secret values, database connection strings, or sensitive provider details.

- *Transport and Network Security*

External traffic should use appropriately configured Transport Layer Security (TLS). Network policies should restrict communication between application services, database services, background workers, storage systems, and external providers to the required paths.

Database and administrative ports should not be publicly exposed without an explicit, reviewed requirement and suitable protective controls.

Network restrictions should be combined with identity verification and service-level authorization rather than treated as sufficient proof that an incoming request is trusted.

- *Container and Runtime Hardening*

Where containerized deployment is used, services should run with the minimum privileges necessary. Non-root execution, restricted filesystem permissions, controlled writable paths, and removal of unnecessary packages can reduce the attack surface.

Container images and application dependencies should be reviewed and scanned for known vulnerabilities. Updates should follow a controlled process that includes compatibility testing and rollback planning.

Builds should be reproducible where practical, and deployed artifacts should be traceable to the corresponding source revision and dependency configuration.

- *Database and Storage Isolation*

Database credentials should have narrowly defined permissions, and storage access should be restricted to authorized services.

Source files, relational metadata, and vector-index records should be protected against unauthorized access and modification. Backup storage should also follow appropriate access restrictions and retention requirements.

Database migrations, index changes, and document lifecycle operations should be tested to ensure that they do not inadvertently expose stale, deleted, or unauthorized content.

- *Health Monitoring and Incident Response*

Operational monitoring should identify repeated authentication failures, authorization denials, abnormal upload volume, unexpected provider usage, processing failures, and unusual resource consumption.

Alerts should be tied to defined operational thresholds or security events. Incident procedures should describe how to investigate failures, revoke credentials, disable affected capabilities, restore service, and preserve relevant audit evidence.

Monitoring should avoid collecting more sensitive content than necessary to detect and investigate these events.

- *Verification and Compliance Boundaries*

Infrastructure security should be evaluated through configuration review, dependency scanning, access-control tests, deployment validation, and authorized security testing.

Mapping platform controls to a security framework or listing recognized security practices does not establish that the platform has passed a penetration test, received a formal audit, or achieved regulatory certification.

Any claim of compliance must identify the applicable requirements, assessment scope, evidence, and assessment status.

➤ *Residual Security Risks and Required Verification*

Despite layered safeguards, PIU Smart Intelligence Agents retains residual risks that require continued assessment. These risks arise from model uncertainty, external dependencies, complex authorization paths, untrusted content, and operational configuration.

The following areas require explicit verification before the platform is described as secure for high-assurance enterprise use.

- *Authentication and Session Management*

Production validation should establish that token or session expiration, revocation, logout, and renewal behave according to the configured identity architecture. Rate limits and controls against repeated authentication attempts should also be tested where applicable.

- *Complete Authorization Coverage*

Every protected API route, agent tool, document-retrieval operation, database-access path, export function, and background task must enforce the applicable authorization policy.

Tests should include direct endpoint calls and cross-resource access attempts rather than relying exclusively on user-interface restrictions.

- *Dependency and Image Vulnerabilities*

Application dependencies, parser libraries, OCR components, model adapters, and container images should be scanned for known vulnerabilities. Remediation should follow a documented process that considers severity, exposure, compatibility, and deployment risk.

- *Provider Data Handling*

Each external provider should be reviewed for the data it receives, applicable retention behavior, processing location, contractual terms, and permitted usage.

These properties may differ between providers and configurations, so they should not be inferred from the platform's general architecture.

- *Adversarial Testing and Independent Review*

Security tests should cover SQL injection attempts, unauthorized tool invocation, direct and indirect prompt injection, malicious file structures, access-control bypass attempts, secret leakage, and resource-exhaustion scenarios.

Independent review should examine both application logic and deployment configuration. Passing a defined test suite provides evidence for the scenarios tested, but it does not prove the absence of all vulnerabilities.

- *Security Governance and Change Management*

Security controls must remain effective as models, providers, dependencies, database schemas, agent tools, and retrieval configurations change.

The platform should document security-relevant changes, validate them through regression testing, and preserve an auditable record of important configuration updates.

- *Evidence-Based Security Claims*

The security analysis distinguishes proposed controls from verified outcomes. A described safeguard should not be represented as implemented unless supported by the source code or deployment evidence. An implemented control

should not be described as effective against all attacks without appropriate testing.

Similarly, passing automated tests does not independently establish compliance certification, successful penetration testing, or complete protection against prompt injection.

The principal security contribution of PIU Smart Intelligence Agents is its layered control strategy: identity and resource authorization, restricted tool execution, safe document processing, protected configuration, data minimization, infrastructure hardening, and explicit verification requirements. The effectiveness of this strategy must be demonstrated through implementation review, adversarial testing, and deployment-specific security assessment.

IX. DATABASE DESIGN AND DATA LIFECYCLE

The database architecture of PIU Smart Intelligence Agents provides the persistent data foundation for document management, semantic retrieval, conversational interaction, and operational governance. It must maintain reliable relationships between original documents, extracted content, chunk records, embeddings, user identities, and processing events while supporting secure access and controlled data lifecycle operations.

The platform's data model combines structured relational records with document content and vector-based retrieval representations. Relational storage supports identity, metadata, lifecycle state, and transactional consistency, whereas the vector index supports similarity-based retrieval of document chunks. Original files may be maintained in controlled file or object storage, depending on the deployed configuration.

The logical design should preserve stable identifiers and explicit relationships across these components. This is particularly important because a single enterprise document may produce many chunks and embeddings, and the resulting representations must remain associated with the correct source document, document version, and authorization scope.

The conceptual entities described in this section represent a reference data model. Exact table names, column definitions, constraints, storage technologies, and relationships must be confirmed against the actual database schema and migration files before they are treated as implemented features.

➤ *Logical Entities and Relationships*

The logical data model organizes platform information into related entities according to their responsibilities. The document registry acts as the central reference for document identity, ownership, classification, and processing lifecycle. Related entities represent extracted chunks, vector

embeddings, conversational records, search activity, and audit events.

• *User and Role Entities*

User records represent identities permitted to interact with the platform. Depending on the authentication architecture, the application may maintain user identifiers, account status, role assignments, creation timestamps, and other required identity metadata.

Role records or equivalent authorization structures define the categories of operations that users may perform. These permissions may govern document uploads, retrieval, SQL analytics, KPI generation, exports, and administrative configuration.

The data model should distinguish user identity from resource authorization. A user may be permitted to use an agent without being authorized to access every document or dataset available to that agent.

Where role assignments and resource permissions are stored in the application database, the relationships should be designed to support consistent authorization checks and auditable changes. If identity or role information is managed by an external identity provider, the database model should retain only the application metadata necessary for the integration.

• *Document Registry*

The document registry is the principal relational entity for document management. It establishes the logical identity of each document and provides the metadata required to track processing, retrieval availability, ownership, and source provenance.

A registry record may contain the following attributes, depending on the implementation:

- ✓ Stable document identifier.
- ✓ Original filename and detected document type.
- ✓ Document category, department, or classification.
- ✓ Owner or authorized resource scope.
- ✓ Source-file storage reference.
- ✓ Document version or revision identifier.
- ✓ Processing status and relevant error category.
- ✓ Creation, update, and processing timestamps.
- ✓ Extraction and indexing configuration metadata.

The registry should serve as the authoritative source for the document's logical lifecycle state. A document that is queued, processing, failed, deleted, or superseded should not be treated as an active searchable document merely because associated vector records remain physically present.

The registry should also distinguish the source file from its derived representations. Reprocessing the same file under a new extraction or embedding configuration may produce different chunks and vectors without creating a completely unrelated logical document.

Where multiple versions are retained, the data model should identify which version is active and how previous versions are treated under the applicable retention and access policies.

- *Document Chunk Entities*

Chunk records represent the text segments produced during document processing. Each chunk is associated with a source document and, where versioning is supported, a specific document version.

A chunk record may include a stable chunk identifier, source document identifier, version identifier, chunk sequence number, extracted text, page or section locator, content hash, and creation timestamp.

The relationship between documents and chunks is generally one-to-many: a document version may produce multiple chunks, while each logical chunk should have an unambiguous source association.

Chunk metadata is essential for more than retrieval. It supports citation generation, document reprocessing, deduplication analysis, deletion, and investigation of retrieval failures.

The data model should preserve enough information to reconstruct the relationship between a retrieved passage and its original document location. If a page or section cannot be identified reliably, the application should not invent a source locator.

- *Embedding and Vector Index Entities*

Embedding records represent the numerical vectors generated from document chunks. Each vector must remain associated with the chunk from which it was generated and with the embedding configuration used to create it.

Depending on the implementation, vectors may be stored directly in a vector-capable relational database, in a dedicated vector table, or in an external vector index. The logical model should remain independent of the physical arrangement while preserving source traceability.

The embedding metadata should identify the applicable model or configuration version and vector dimension. Where multiple embedding models are supported, the model identity should be explicit so that incompatible vectors are not mixed unintentionally.

A chunk may have different embeddings across processing versions or model configurations. If such versions are retained, the data model should identify the active representation used by retrieval and define how obsolete representations are removed.

The vector index should not be treated as the sole authority for document ownership or lifecycle state. Retrieval should consult trusted metadata or an equivalent enforcement mechanism to confirm that the source document remains available and authorized.

- *Chat Session and Message Entities*

Chat-session records associate conversational activity with a user or another explicitly authorized scope. A session may include a stable session identifier, ownership information, creation and update timestamps, and lifecycle status.

Message records represent individual interactions within a session. Depending on the implemented design, they may contain message identifiers, session references, message roles, timestamps, and the information required to reconstruct the conversation.

Where the platform stores agent outputs or source references, those records should preserve the relationship between a response and the relevant execution or evidence metadata. However, storing complete prompts and responses is not always necessary and should be governed by data-minimization and retention requirements.

The database must prevent unauthorized users from reading or modifying another user's sessions. A session identifier alone should not grant access; the application must verify the requesting identity and applicable resource permissions.

Conversation records should also remain separate from authoritative business records. A statement stored in chat history does not become a verified KPI, database result, or approved policy definition simply because it appears in a previous assistant response.

- *Search History and Audit Entities*

Search-history records may capture information about retrieval activity, such as the requesting identity, query timestamp, selected filters, result status, and relevant execution metadata. The exact information retained should reflect operational needs and privacy requirements.

Audit records serve a different purpose: they provide a trace of significant security, administrative, and lifecycle events. Examples include document deletion, permission changes, provider configuration updates, failed authorization attempts, and important processing-state transitions.

Operational records may also track background jobs, retry attempts, provider failures, and index-reconciliation activities.

These record categories should not be combined indiscriminately. Search history may contain sensitive user intent, while audit records may need stronger integrity and retention guarantees. Their access controls and retention periods should therefore be defined according to their respective purposes.

The figure should illustrate users and roles, the document registry, document versions, chunks, embeddings or vector-index records, chat sessions, messages, search history, and audit/operational records. The relationships should indicate source-document traceability, ownership, and

lifecycle dependencies. The final diagram must be aligned with the actual database migrations before being presented as an implemented schema.

➤ *Primary Keys, Foreign Keys, Constraints, and Indexes*

Database integrity is essential to maintaining consistent relationships between application records and derived search representations. PIU Smart Intelligence Agents should use stable identifiers, appropriate relational constraints, and workload-driven indexes to reduce ambiguity and prevent invalid relationships.

• *Primary Keys and Stable Identifiers*

Each logical entity should have a primary key that uniquely identifies its records. The identifier strategy may use UUIDs, numeric identifiers, or another suitable format according to the entity's requirements.

Document identifiers should remain stable across operations that do not create a new logical document. Reprocessing a document should not accidentally cause its chunks to become associated with an unrelated registry entry.

Chunk and embedding identifiers should also be unambiguous within their relevant document and processing versions.

A stable identifier is not a substitute for authorization. Knowing a document ID, session ID, or chunk ID must not automatically permit access to the associated resource.

• *Foreign Keys and Referential Integrity*

Foreign keys should enforce the relationships between entities stored within the same relational database where appropriate.

Examples include references from chunks to their source documents, messages to their chat sessions, and resource-specific operational records to the relevant document or job.

Referential constraints help prevent records from referencing nonexistent parents. However, they do not automatically enforce consistency with records stored in a separate vector service or object store.

Cross-store relationships require additional application-level coordination and reconciliation because ordinary relational transactions cannot guarantee atomic updates across independently managed systems.

• *Uniqueness and State Constraints*

Uniqueness constraints should prevent duplicate logical identities and ambiguous records.

Depending on the schema, these constraints may apply to external document identifiers, version-specific chunk identifiers, job identifiers, or other fields that must remain unique within a defined scope.

State constraints should restrict processing records to recognized lifecycle values. Where practical, application logic or database mechanisms should also enforce valid transitions between processing states.

A database constraint that restricts status values to an approved set does not, by itself, guarantee that the transition sequence is valid. The transition logic must be enforced by the appropriate application or database mechanism.

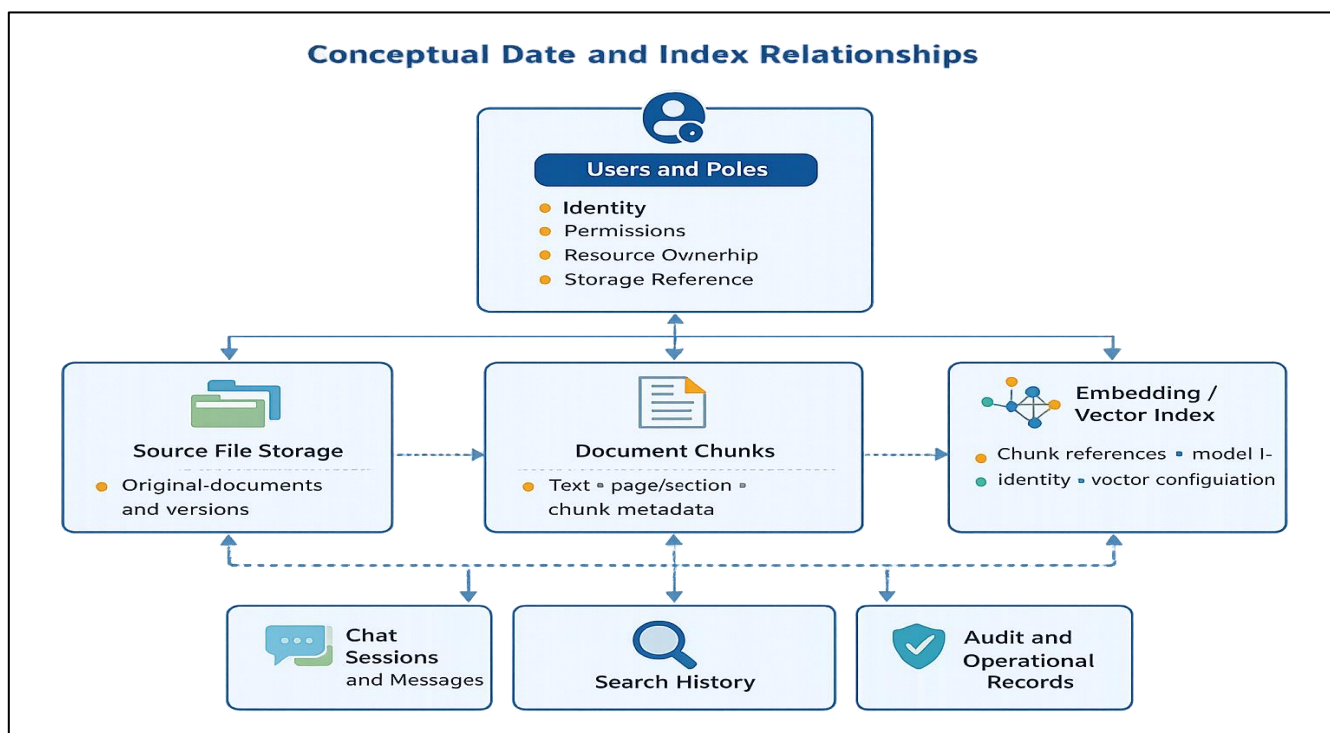


Fig 7 Conceptual Data and Index Relationships

- *Index Design*

Indexes should be selected according to measured access patterns rather than added indiscriminately.

Relational indexes may be useful for document ownership, processing status, document category, creation timestamps, active versions, session identifiers, and other frequently queried fields.

Composite indexes may be appropriate when application queries repeatedly filter by multiple attributes, but their order should reflect the actual query predicates and database behavior.

Vector indexes should be configured according to the selected vector technology, similarity operation, dimensionality, and expected workload.

The source architecture describes 768-dimensional embeddings and cosine-distance retrieval, but this does not establish that a particular vector index or indexing strategy is optimal. The actual extension, index type, query operators, and performance characteristics must be verified in the target environment.

- *Query and Index Validation*

Index effectiveness should be evaluated using representative queries and realistic data volumes.

Measurements may include query latency, execution plans, storage overhead, write performance, and retrieval quality. A faster vector search is not necessarily a better retrieval system if it returns less relevant evidence.

Database indexes should also be monitored as the corpus grows and the balance between insertion, deletion, and search operations changes.

Index maintenance, reindexing, and migration procedures should be documented for the chosen database and vector technology.

➤ *Data Retention, Deletion, and Lifecycle Consistency*

Data lifecycle management defines how long information is retained, when it becomes unavailable, and how it is removed from the platform. It must cover not only original files but also derived text, chunks, embeddings, cached results, conversation records, and operational metadata.

- *Retention Policy*

The platform should establish retention rules for each relevant data category. Source documents, extracted text, vector representations, search history, chat messages, audit records, and backups may have different operational and governance requirements.

The retention policy should identify the applicable retention period or deletion condition, the responsible component, and the expected treatment of derived data.

Retention decisions should account for the deployment's business requirements, privacy obligations, contractual restrictions, and any applicable legal holds. A single retention period should not be assumed to suit every data category.

- *Source Document Deletion*

When a document is deleted or withdrawn, the system should first prevent it from being returned through ordinary retrieval paths.

The deletion workflow may then remove or invalidate its associated chunks and vector records, delete the original file where required, and update relevant registry metadata.

If physical deletion across all stores cannot occur atomically, a logical deletion marker or equivalent control can prevent retrieval while cleanup proceeds.

The application should not report deletion as fully completed until the defined deletion requirements have been satisfied. Where some records must be retained for audit or legal reasons, the retained information should be limited to the approved purpose and should not preserve sensitive document content unnecessarily.

- *Derived Data and Cache Invalidation*

Deleting a source file is insufficient if derived content remains accessible through chunks, embeddings, cached retrieval results, conversation summaries, or other stored representations.

The lifecycle design should identify every component that may retain source-derived information and define how deletion or access revocation affects each one.

Where cached content is used, the system should apply appropriate invalidation rules. Previously generated summaries or conversational responses should not automatically be treated as current authoritative evidence after their source has been deleted or its permissions have changed.

The exact scope of deletion must be documented so that the system can distinguish ordinary index cleanup from requirements involving all retained copies, backups, and derived artifacts.

- *Reprocessing and Version Replacement*

A document may require reprocessing after a source update, parser change, chunking adjustment, or embedding-model migration.

The application should preserve the relationship between the source version and its derived representations. A new processing run should not create ambiguous mixtures of old and new chunks.

A controlled replacement procedure may prepare the new version, validate its extracted content and index records,

activate it, and then retire the previous version according to the retention policy.

If the new processing run fails, the platform should preserve a consistent state and should not silently activate incomplete derived content.

- *Reconciliation Across Stores*

The relational database, source-file storage, and vector index may become inconsistent after interrupted operations or partial failures.

A reconciliation process should detect conditions such as registry records without source files, indexed documents without expected vectors, orphaned vector records, duplicate chunks, and obsolete representations remaining after deletion.

Detected problems should be classified before repair. For example, missing embeddings may require reprocessing, whereas an orphaned vector record may require removal after confirming that no active document version references it.

Repairs should be auditable and should not delete records solely because a dependency was temporarily unavailable.

➤ *Database Migration, Backup, and Recovery*

Database migrations and recovery procedures help ensure that changes to the schema, indexing configuration, and deployment environment do not compromise data integrity or service continuity.

- *Versioned Schema Migrations*

Schema changes should be managed through a versioned migration process rather than relying on undocumented manual modifications.

Each migration should define the intended schema change, its prerequisites, the expected impact on existing records, and the recovery or rollback approach where feasible.

Changes affecting document identifiers, chunk relationships, lifecycle states, or embedding metadata require particular care because they may affect retrieval correctness and source traceability.

Migration testing should use a representative copy of the schema and data where practical. The process should verify that the application version and database schema remain compatible throughout the deployment.

- *Index and Embedding Migrations*

Vector-index changes may require additional steps beyond relational schema migration.

For example, changing the embedding model may require generating new vectors and creating a compatible index. A safe migration should verify the new representations before routing normal retrieval traffic to them.

The deployment should not assume that a relational migration automatically updates vector records or rebuilds incompatible indexes.

Migration procedures should specify the expected processing state, validation checks, activation point, and rollback behavior.

- *Backup Strategy*

The backup strategy should cover the authoritative relational records and all other persistent information required to restore the platform's defined service level.

Depending on the deployment, this may include original documents, durable processing-job records, configuration metadata, vector-index state, and relevant audit records.

If a vector index can be reconstructed from source files, extracted content, and embedding configuration, the recovery plan should document the reconstruction process and its resource requirements. Rebuildable data should not be assumed to be inexpensive or immediately recoverable.

Backups should be protected against unauthorized access and should follow the applicable retention and encryption policies.

- *Recovery and Consistency Verification*

Recovery procedures should verify more than whether a database backup can be restored.

The restored environment should be checked for schema compatibility, record integrity, source-file availability, index completeness, document lifecycle consistency, and correct authorization behavior.

Where vector indexes are restored separately from relational records, the system should validate the relationships between document versions, chunk identifiers, and vector records before enabling normal retrieval.

Recovery tests should also verify that deleted or superseded content does not become unintentionally searchable after restoration.

- *Recovery Objectives and Testing*

The deployment should define suitable recovery objectives according to business requirements. These may include a Recovery Point Objective (RPO), which specifies the acceptable amount of data loss measured in time, and a Recovery Time Objective (RTO), which specifies the target time for restoring a service.

No numerical RPO or RTO should be claimed unless it has been defined for the deployment and supported by testing.

Recovery exercises should include database restoration, source-file recovery, index reconstruction where required,

application restart, authorization checks, and representative retrieval queries.

The results should record the tested environment, recovery duration, data-integrity findings, and unresolved limitations.

➤ *Data Governance and Integrity Verification*

Database design must support not only application functionality but also traceability, controlled access, and accountable data management.

The platform should maintain sufficient metadata to identify the source and processing history of indexed content. It should also define how document ownership, version changes, retention decisions, and deletion operations affect the associated derived records.

Data-integrity verification should include referential checks, uniqueness checks, lifecycle-state consistency, and reconciliation across relational storage, file storage, and vector indexes.

Where a generated response depends on retrieved evidence or a database calculation, the system should preserve enough lineage to identify the source records or passages used in the operation.

Audit records should capture significant changes without unnecessarily retaining confidential content. Their access and retention policies should be defined separately from those of ordinary conversation or search history.

The database design should be evaluated using tests that cover duplicate job delivery, interrupted indexing, document replacement, deletion, migration failure, backup restoration, and recovery from partial cross-store updates.

These tests provide evidence about the consistency and recoverability of the deployed system. A conceptual entity model, a successful schema migration, or a completed database backup alone does not establish full-service data integrity or recoverability.

In summary, the database and data lifecycle architecture of PIU Smart Intelligence Agents is intended to preserve stable identities, valid relationships, authorized access, source traceability, and controlled retention across structured records and derived search representations. The final implementation should be verified against its migration files, index configuration, storage integrations, and recovery tests before the conceptual model is presented as the deployed production schema.

X. IMPLEMENTATION AND TECHNOLOGY STACK

The source describes a Python-oriented backend using FastAPI, relational persistence through PostgreSQL, ORM-based access, asynchronous/background processing, document parsers and OCR, embedding and LLM adapters, and a web front end. Containers package services for repeatable deployment. The stack is a source-derived implementation description, not a claim that every component is mandatory for every deployment.

Table 4 Implementation and Technology Stack

Concern	Role / Implementation Consideration
Backend API	FastAPI endpoints, request validation, consistent error contracts.
Relational persistence	PostgreSQL for registry metadata, users, sessions, histories, and audit records.
Vector retrieval	Vector extension or compatible vector service; verify availability and model/index compatibility.
Document processing	Format-aware parsers, OCR, normalization, chunking, asynchronous workers.
AI providers	Adapters for LLM and embedding endpoints, timeouts, compatibility checks, controlled fallback.
Frontend	Enterprise workspace for upload, agent selection, search, chat, result review, and exports.
Deployment	Containers, health checks, non-root execution, secret injection, persistent storage, backups.
Quality assurance	Automated tests, migrations, dependency scanning, security tests, load and recovery tests.

➤ *Configuration and Provider Abstraction*

Provider-specific settings should be centralized and validated at startup. Configuration includes endpoint, model identifier, timeouts, token budgets, retry policy, embedding dimensions, and fallback order. Secrets must remain separate from ordinary configuration. Provider changes should be controlled and tested against compatibility requirements.

➤ *Observability*

Telemetry should cover request counts, latency by stage, error categories, queue depth, ingestion outcomes, retrieval results, provider availability, circuit-breaker state, token usage, and database health. Logs should include correlation identifiers and state transitions while avoiding sensitive document text and credentials. Health endpoints should distinguish process liveness from readiness.

XI. ADVANTAGES AND COMPARATIVE ANALYSIS

➤ *Advantages*

- A unified workspace for document intelligence, conversational AI, structured analytics, and KPI-oriented workflows.
- Layered separation between controllers, services, agents, providers, and data access.
- Asynchronous ingestion separates upload responsiveness from expensive processing.
- Provider routing and bounded resource controls support resilient operation.
- Document identifiers, history, and citation-oriented answers improve traceability.

➤ *Limitations*

- Vector storage and embedding pipelines add operational and migration complexity.
- RAG quality depends on extraction, chunking, retrieval configuration, and source coverage.

- External model providers can introduce latency, cost, privacy, and availability dependencies.
- Agent orchestration increases the need for explicit tool permissions and result validation.
- Security controls cannot rely solely on prompt wording.
- Limited benchmark and scale evidence constrains production-readiness claims.

Table 5 Future Enhancements and Roadmap

Indicative Horizon	Focus	Exit Evidence
0–3 months	Authorization coverage, prompt-injection defenses, SQL guardrails, secret/dependency controls.	Passing access-control, adversarial prompt, and SQL-policy test suites.
3–6 months	Versioned evaluation corpus, chunking and ranking experiments, KPI definitions and lineage.	Published retrieval/answer metrics and KPI correctness tests.
6–12 months	Provider failover tests, durable queues, storage scaling, recovery drills.	Fault-injection results, restore evidence, measured availability and latency.
12+ months	Multilingual support, broader file coverage, workflow integration, advanced reporting and agent collaboration.	Per-agent evaluation, operational acceptance criteria, controlled rollout.

➤ *Recommended Validation Gates*

- Do not claim production readiness until access-control, adversarial prompt, and SQL-policy tests pass.
- Establish a versioned evaluation set and report retrieval, answer, summary, SQL, and KPI metrics.
- Run controlled load and fault-injection tests and publish reproducible p50/p95/p99 latency and error rates.
- Demonstrate backup restoration and cross-store reconciliation, not merely successful backup creation.
- Track provider quality and compatibility before enabling automatic fallback.
- Maintain an implementation-status register distinguishing complete, partial, planned, and externally dependent controls.

XII. CONCLUSION

This paper presents PIU Smart Intelligence Agents as a unified enterprise multi-agent AI platform whose capabilities include RAG-based document intelligence, semantic search, document summarization, natural-language-to-SQL, KPI generation, and general-purpose conversational assistance. Its principal architectural contribution is the integration of these capabilities with shared identity, authorization, provider abstraction, asynchronous processing, observability, and operational governance. The source implementation reports 1,242 automated tests passed and two integration tests skipped, but the evidence does not include independent reproduction, a complete external RAG benchmark, a full load-test profile, or independently validated security results. Accordingly, the platform should be regarded as an engineering reference baseline rather than proof that all production risks have been eliminated. The next steps are to validate each agent independently, strengthen retrieval and KPI evaluation, demonstrate authorization and injection defenses dynamically, test provider and data recovery behavior, and establish reproducible performance and scale measurements.

ACKNOWLEDGMENT

This manuscript is a source-grounded rewrite and consolidation of the supplied IEEE-style document, expanded to frame the application as a multi-agent enterprise intelligence platform. No external experimental results have been added, and reported implementation claims have not been independently reproduced. Agent-specific capabilities named in the expanded product framing should be reconciled with source code and test artifacts before submission.

REFERENCES

- [1]. P. Lewis, E. Perez, A. Piktus, F. Petroni, V. Karpukhin, N. Goyal, H. Küttler, M. Lewis, W.-t. Yih, T. Rocktäschel, S. Riedel, and D. Kiela, “Retrieval-augmented generation for knowledge-intensive NLP tasks,” in *Advances in Neural Information Processing Systems*, vol. 33, pp. 9459–9474, 2020.
- [2]. A. Vaswani *et al.*, “Attention is all you need,” in *Advances in Neural Information Processing Systems*, vol. 30, 2017.
- [3]. N. Reimers and I. Gurevych, “Sentence-BERT: Sentence embeddings using Siamese BERT-networks,” in *Proc. Conf. Empirical Methods in Natural Language Processing and 9th Int. Joint Conf. Natural Language Processing (EMNLP-IJCNLP)*, 2019, pp. 3982–3992, doi: 10.18653/v1/D19-1410.
- [4]. N. F. Liu, K. Lin, J. Hewitt, A. Paranjape, M. Bevilacqua, F. Petroni, and P. Liang, “Lost in the middle: How language models use long contexts,” *Transactions of the Association for Computational Linguistics*, vol. 12, pp. 157–173, 2024, doi: 10.1162/tacl_a_00638.
- [5]. T. Yu *et al.*, “Spider: A large-scale human-labeled dataset for complex and cross-domain semantic parsing and text-to-SQL task,” in *Proc. 2018 Conf. Empirical Methods in Natural Language Processing (EMNLP)*, 2018, pp. 3911–3921.
- [6]. S. Yao *et al.*, “ReAct: Synergizing reasoning and acting in language models,” in *Proc. Int. Conf. Learning Representations (ICLR)*, 2023.

- [7]. T. Schick *et al.*, “Toolformer: Language models can teach themselves to use tools,” in *Advances in Neural Information Processing Systems*, vol. 36, 2023.
- [8]. Y. Gao *et al.*, “Retrieval-augmented generation for large language models: A survey,” arXiv:2312.10997, 2023.
- [9]. S. Es, J. James, L. Espinosa-Anke, and S. Schockaert, “RAGAS: Automated evaluation of retrieval augmented generation,” in *Proc. 18th Conf. European Chapter of the Association for Computational Linguistics: System Demonstrations*, 2024, pp. 150–158, doi: 10.18653/v1/2024.eacl-demo.16.
- [10]. Sujit Shibaprasad Maity, “StockBuddy: An interpretable and lightweight web-based decision support system for short-term equity forecasting and recommendation,” *International Journal of Innovative Science and Research Technology*, vol. 11, no. 3, pp. 1759–1769, Mar. 2026, doi: 10.38124/ijisrt/26mar1068. [Online]. Available: <https://www.ijisrt.com/stockbuddy-an-interpretable-and-lightweight-webbased-decision-support-system-for-shortterm-equity-forecasting-and-recommendation>.
- [11]. Y. A. Malkov and D. A. Yashunin, “Efficient and robust approximate nearest neighbor search using hierarchical navigable small world graphs,” *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 42, no. 4, pp. 824–836, 2020.
- [12]. Y. Xu *et al.*, “LayoutLM: Pre-training of text and layout for document image understanding,” in *Proc. 26th ACM SIGKDD Int. Conf. Knowledge Discovery and Data Mining*, 2020, pp. 1192–1200.
- [13]. National Institute of Standards and Technology, *Artificial Intelligence Risk Management Framework (AI RMF 1.0)*, NIST AI 100-1, Jan. 2023, doi: 10.6028/NIST.AI.100-1.
- [14]. National Institute of Standards and Technology, *Artificial Intelligence Risk Management Framework: Generative Artificial Intelligence Profile*, NIST AI 600-1, July 2024, doi: 10.6028/NIST.AI.600-1.
- [15]. OWASP Foundation, “LLM01: Prompt injection,” *OWASP Top 10 for Large Language Model Applications*. [Online]. Available: <https://genai.owasp.org/llm-top-10/>. Accessed: Oct. 10, 2026.
- [16]. OWASP Foundation, “SQL injection prevention cheat sheet.” [Online]. Available: https://cheatsheetseries.owasp.org/cheatsheets/SQL_Injection_Prevention_Cheat_Sheet.html. Accessed: Oct. 10, 2026.
- [17]. OWASP Foundation, “File upload cheat sheet.” [Online]. Available: https://cheatsheetseries.owasp.org/cheatsheets/File_Upload_Cheat_Sheet.html. Accessed: Oct. 10, 2026.
- [18]. pgvector Contributors, “pgvector: Open-source vector similarity search for Postgres,” software repository. [Online]. Available: <https://github.com/pgvector/pgvector>. Accessed: Oct. 10, 2026.
- [19]. M. Kleppmann, *Designing Data-Intensive Applications: The Big Ideas Behind Reliable, Scalable, and Maintainable Systems*. Sebastopol, CA, USA: O’Reilly Media, 2017.
- [20]. National Institute of Standards and Technology, *The NIST Cybersecurity Framework (CSF) 2.0*, NIST CSWP 29, Feb. 2024, doi: 10.6028/NIST.CSWP.29.
- [21]. Sujit Shibaprasad Maity, “An intelligent heart disease identification framework using supervised machine learning for clinical decision support,” *International Journal for Multidisciplinary Research*, vol. 8, no. 1, Jan.–Feb. 2026, doi: 10.36948/ijfmr.2026.v08i01.69115. [Online]. Available: <https://www.ijfmr.com/research-paper.php?id=69115>.