# The Adaptive Hierarchical Reasoning (AHR) Engine: A Blueprint for a Novel, Self-Improving Document Intelligence System

## Part I: The Foundational Architecture - Realizing the Hierarchical Semantic Reasoning (HSR) System

The conceptual framework for a Hierarchical Semantic Reasoning (HSR) system provides a robust and sophisticated foundation for advanced document understanding.[1] To transform this vision into a tangible, high-performance, and scalable reality, a carefully selected stack of open-source technologies must be architected. This section outlines the blueprint for this foundational layer, translating the HSR concept into an implementable system that addresses the unique challenges of complex domains like insurance and legal documentation.

### 1.1 The Ingestion and Knowledge-Structuring Pipeline

The efficacy of any reasoning system is predicated on the quality and structure of its underlying knowledge base. The initial and most critical phase of the AHR engine is the ingestion pipeline, which is responsible for transforming raw, unstructured, and multi-modal documents into a coherent, interconnected, and machine-readable knowledge graph. This process goes beyond simple text extraction, employing advanced techniques to preserve context, identify domain-specific entities, and model the intricate relationships within the data.

#### 1.1.1 Hybrid Structural-Semantic Chunking

The initial step in processing documents is chunking—breaking down large texts into smaller, manageable segments for embedding and retrieval. A naive approach, such as fixed-size chunking, often leads to context fragmentation by splitting sentences or logical units of information across different chunks.[2] While the proposed HSR system correctly identifies the need for semantic chunking to group related sentences, this method alone can also be suboptimal. Semantic chunking, which relies on embedding similarity, can inadvertently group text from disparate structural sections of a document (e.g., a paragraph from the main body and a footnote) if their topics are similar, thereby losing the document's inherent logical flow.[3]

To overcome these limitations, the AHR engine will implement a more robust **Hybrid Structural-Semantic Chunking** strategy. This advanced methodology combines the strengths of both structural and semantic analysis, directly fulfilling the "Multi-Modal Context Preservation" goal outlined in the HSR design.[1] The process unfolds in two stages:

1. **Structural Segmentation:** The system first analyzes the document's layout and structure. This can be achieved by leveraging document-native formats like Markdown or by using layout-aware models to parse PDFs and identify structural elements such as headings, lists, tables, and figures.[5] This initial pass divides the document into large, structurally coherent blocks.
2. **Semantic Cohesion:** Within each of these structural blocks, the system then applies semantic chunking. Using a sentence transformer model, it calculates the embedding for each sentence and identifies breakpoints where the semantic meaning shifts significantly.[7] This ensures that chunks are not only semantically cohesive but also respect the logical boundaries established by the document's author.

This hybrid approach, inspired by emerging frameworks that integrate spatial and semantic analysis [8], ensures that a table and its caption are processed as a single contextual unit, a list of policy exclusions remains intact, and a section heading correctly frames the paragraphs that follow it. This preserves the rich, multi-modal context of the original document, creating a much higher-fidelity data source for the subsequent reasoning steps.

### 1.1.2 Domain-Adapted Entity and Relationship Extraction

Once documents are appropriately chunked, the system must extract the meaningful entities and the relationships between them. For domains as specialized as insurance and law, general-purpose Named Entity Recognition (NER) models are insufficient. Studies show that such models experience a significant degradation in accuracy—with F1-scores dropping by as much as 60%—when applied to legal texts without domain-specific training.[1] Therefore, developing a custom, domain-adapted NER model is a prerequisite for success.

The AHR engine will leverage **spaCy**, a highly efficient, production-ready open-source library, for this task.[11] The process for creating a custom NER model involves:

1. **Data Annotation:** Creating a high-quality training dataset is the most critical step. This involves annotating sample documents (e.g., insurance policies, legal contracts) with the desired entity labels (e.g., POLICY_HOLDER, COVERAGE_LIMIT, JURISDICTION, LEGAL_STATUTE). The data is typically formatted in JSON, specifying the text and the start/end character offsets for each entity.[12]
2. **Model Fine-Tuning:** Using spaCy's training pipeline, a pre-trained language model (e.g., en_core_web_lg) is fine-tuned on this custom annotated data.[13] This process adjusts the model's weights to become proficient at recognizing the specific entities relevant to the insurance and legal domains.

A subtle but critical challenge in this process is **catastrophic forgetting**. This phenomenon occurs when a model, fine-tuned intensively on a narrow domain, begins to lose its proficiency in general-purpose tasks it was originally trained on, such as recognizing common entities like dates, people, or organizations.[13] A system that can identify a "subrogation clause" but can no longer correctly parse a date is fundamentally flawed.

To proactively mitigate this risk, the AHR engine's training protocol will incorporate a dual strategy:

- **Balanced Re-training Data:** The dataset used for fine-tuning and subsequent updates (particularly within the Active Learning Loop detailed in Part II) will not consist solely of new domain-specific examples. It will be augmented with a representative sample of the general-purpose data the original spaCy model was trained on. This reminds the model of its foundational knowledge while it learns new concepts.
- **Regularization and Parameter Constraints:** Advanced techniques such as

regularization will be employed during training. These methods apply constraints that penalize large changes to the model's weights, particularly those identified as crucial for general knowledge.[15] This effectively encourages the model to learn new information by adjusting less critical parameters, thereby preserving its core capabilities.

By addressing catastrophic forgetting head-on, the AHR engine ensures the NER model is both a domain expert and a generalist, capable of understanding the full spectrum of information present in the documents.

### 1.1.3 Dynamic Knowledge Graph Construction

The final stage of the ingestion pipeline is to assemble the extracted information into a structured, queryable **Knowledge Graph (KG)**. The HSR system's choice of Neo4j is an excellent one, as it is a mature, high-performance, and scalable native graph database perfectly suited for this application.[1] The property graph model it employs is inherently flexible, allowing the schema to evolve over time—a critical feature for the "Self-Evolving KG" concept introduced in Part II.[20]

The KG will be constructed as follows:

- **Nodes:** Each distinct piece of information becomes a node with a specific label and properties.
  - Document nodes represent the source files (e.g., policy_terms.pdf).
  - Chunk nodes represent the structural-semantic segments of text, containing the text itself and its vector embedding.
  - **Entity Nodes:** Entities extracted by the NER model become their own nodes (e.g., a Person node with name: "John Doe", a Policy node with policy_number: "XYZ123", a Clause node with type: "Exclusion").
- **Relationships:** The connections between these nodes are represented by directed, typed relationships.
  - PART_OF: Connects a Chunk node to its parent Document node.[21]
  - NEXT_CHUNK: Creates a sequential link between chunks, preserving the document's narrative flow.[21]
  - CONTAINS_ENTITY: Links a Chunk node to the Entity nodes found within its text.
  - **Semantic Relationships:** The system will also extract relationships between

entities (e.g., (Person)-->(Policy), (Clause)-->(Document)).

This rich, interconnected graph structure moves beyond a simple vector database of chunks. It captures the complex web of relationships inherent in the source material, enabling the powerful multi-hop reasoning capabilities required by the HSR system's reasoning engine.[1]

### 1.2 The Query Deconstruction and Reasoning Core

With a well-structured knowledge graph in place, the system's focus shifts to understanding and answering user queries. This "brain" of the AHR engine deconstructs natural language questions into formal, machine-executable instructions and reasons over the KG to synthesize accurate, evidence-backed answers.

#### 1.2.1 Multi-faceted Query Analysis

A simple semantic search of a user's raw query is often insufficient for complex, multi-conditional questions. The AHR engine will implement the "Query Intelligence Module" from the HSR design, which performs a deep analysis of the user's intent before initiating retrieval.[1] This involves two key steps:

1. **Intent Classification:** A fine-tuned classifier maps the natural language query to a predefined, hierarchical decision tree of intents. For example, a query like *"Am I covered for knee surgery in Pune under my 3-month-old policy?"* would be mapped to a path such as eligibility_check -> coverage_verification -> geographic_restriction_check -> waiting_period_check. This transforms the query from a single string into a structured plan of action.
2. **Constraint Extraction and Fuzzy Matching:** The system uses the domain-adapted NER model to extract key parameters from the query (e.g., procedure: "knee surgery", location: "Pune", policy_duration: "3 months"). Crucially, it will also implement "Fuzzy Constraint Matching" to handle ambiguity. A term like "recent policy" would be translated into a formal constraint like policy_age < 90 days, and "major surgery" would be mapped to a predefined list

of covered high-cost procedures.

This structured approach to query understanding is a cornerstone of the system's explainability. It ensures that the reasoning process is deterministic and auditable, a non-negotiable requirement in legal and insurance contexts.[1]

### 1.2.2 The Hybrid Retrieval Engine

No single retrieval method is optimal for all types of information needs. The AHR engine will therefore implement the powerful **Hybrid Retrieval System** envisioned in the HSR document, which combines three complementary search modalities to maximize both relevance and precision.[1]

1. **Dense Retrieval (Semantic Search):** The system will use the user query's embedding to perform a vector similarity search against the Chunk node embeddings stored in a vector database. This method excels at finding conceptually related information, even if the keywords don't match. For this component, **Qdrant** is the recommended open-source vector database. It is built in Rust for high performance, offers advanced filtering capabilities that can be tied to the KG, and is specifically well-suited for RAG workflows.[23]
2. **Sparse Retrieval (Keyword Search):** To handle queries that require exact matches (e.g., specific policy numbers, names, or legal statute codes), the system will employ a traditional sparse retrieval algorithm like BM25. This ensures that critical, precise terms are not missed by the semantic search.
3. **Graph Traversal (Relational Search):** This is where the power of the knowledge graph is truly unleashed. Using the entities extracted from the query, the system will execute Cypher queries against the Neo4j database to find information based on explicit relationships. For example, it can find all Clauses related to a specific Policy that are also of type Exclusion.

The combination of vector-based semantic search and keyword-based search is a state-of-the-art technique for improving retrieval quality.[26] The addition of graph traversal creates a uniquely powerful engine capable of answering questions that require understanding semantic meaning, exact terms, and explicit relationships simultaneously. The results from these three retrievers will be combined using a fusion algorithm with learnable weights to produce a single, ranked list of the most relevant evidence.

### 1.2.3 Evidence-Based Chain-of-Reasoning (CoR)

The final step is to synthesize a coherent, trustworthy, and explainable answer from the retrieved evidence. The AHR engine will construct a **Chain-of-Reasoning (CoR)**, as illustrated in the HSR design, that explicitly shows how it arrived at its conclusion.[1]

For each step in the query plan (e.g., age_eligibility_check), the system will present the evidence it found, including:

- The specific clause or text snippet.
- A citation pointing to the source document and page number.
- A confidence score derived from the retrieval process.
- A sub-decision for that step (e.g., ELIGIBLE, COVERED, CONFLICT_FOUND).

These individual steps are then fused into a final decision, such as CONDITIONALLY_APPROVED. The output will not only provide the answer but also a full justification, listing the supporting and restricting clauses that led to the conclusion. This "Explainable AI" approach is the system's primary competitive advantage, providing the transparency and auditability required to build trust with users in high-stakes domains.[1]

## Part II: The Novelty Engine - The Adaptive Feedback Loop

A truly intelligent system does not remain static; it learns, adapts, and improves over time. The foundational architecture described in Part I provides a powerful reasoning engine, but its knowledge and capabilities are fixed at the time of deployment. This section introduces the novel core of the AHR engine: a multi-layered framework of interconnected feedback loops designed to make the system self-correcting, self-evolving, and continuously improving. This framework directly addresses the user's desire for a "self-improving" system in a way that is practical, phased, and built entirely on open-source principles.[1]

### 2.1 The Self-Correction Loop for Retrieval and Generation

The quality of a Retrieval-Augmented Generation (RAG) system is fundamentally limited by the quality of the information it retrieves. If the retrieval step returns irrelevant or low-quality document chunks, even the most advanced Large Language Model (LLM) will produce a poor or incorrect answer, a phenomenon known as "garbage in, garbage out." To address this, the AHR engine will incorporate a **Self-Correction Loop**, a form of "Corrective RAG" or "Self-Reflective RAG," which introduces an LLM-based "judge" to evaluate and refine the retrieval process in real-time.[27]

This approach provides a significant step towards self-improvement without the immense complexity of implementing a full Reinforcement Learning from Human Feedback (RLHF) pipeline from scratch.[29] A full RLHF implementation requires training a separate reward model and using complex optimization algorithms, which conflicts with the goal of a system that is "not very hard to implement".[1] The self-correction loop, however, can be implemented using the same core LLM (e.g., Llama 3.1) for both generation and grading, making it a pragmatic and powerful enhancement.

The workflow for this loop is as follows:

1. **Initial Retrieval:** The Hybrid Retrieval Engine retrieves a set of candidate document chunks based on the user's query.
2. **Relevance Grading:** Before these chunks are passed to the generator, they are first presented to the LLM configured in a "grader" mode. The grader's task is to evaluate the relevance of each chunk to the original query. It assigns a score or a categorical label (e.g., Relevant, Ambiguous, Irrelevant) to each chunk.[28]
3. **Conditional Routing and Query Refinement:** The system's next action is determined by the grader's output:
    - If a sufficient number of chunks are graded as Relevant, they are passed directly to the generation stage.
    - If the majority of chunks are graded as Irrelevant or Ambiguous, this signals a failure in the initial retrieval. The system then triggers a **Query Refinement** step. The LLM is prompted to rephrase the original query, explore alternative terminology, or break it down into more specific sub-questions. This refined query is then used to perform a new retrieval, effectively giving the system a "second chance" to find better information.[28]
4. **Evidence-Grounded Generation:** The final, validated set of relevant chunks is used as context for the LLM to generate the final, evidence-backed answer.

This loop enables the system to intelligently self-correct from poor retrieval results, significantly reducing the likelihood of hallucinations and improving the overall quality and reliability of its responses.

### 2.2 The Self-Evolving Knowledge Graph

The knowledge graph should not be a static archive but a living, dynamic representation of the system's understanding. The AHR engine is designed to facilitate the continuous growth and refinement of its KG, making it a **Self-Evolving Knowledge Graph**. This concept aligns with research indicating that the future of KGs lies in their ability to adapt dynamically to real-time changes, moving beyond manual, pre-defined schemas.[32] The synergy between LLMs and Neo4j is particularly powerful here, as LLMs can be used to infer schema and relationships on the fly, which can then be seamlessly integrated into Neo4j's flexible property graph structure.[21]

The self-evolution of the KG is driven by three primary mechanisms:

1. **Real-time Ingestion:** The most straightforward mechanism is the continuous ingestion of new documents. As new policies are written or new legal precedents are set, they are processed by the ingestion pipeline, and new Document, Chunk, and Entity nodes are added to the graph in real-time, keeping the knowledge base current.
2. **Insight-driven Relationship Creation:** The system will log successful query-answer pairs, especially those that receive positive user feedback (e.g., a thumbs-up rating). An offline agentic process will periodically analyze this data. By identifying entities that frequently co-occur in the context of highly-rated answers but are not yet directly linked in the graph, the agent can infer new, valuable relationships. For instance, if a specific legal precedent is consistently cited in successful answers about a particular type of insurance claim, the agent can propose adding a new IS_RELEVANT_TO relationship between the corresponding nodes. This strengthens the graph's connective tissue based on demonstrated utility, not just on what was explicitly stated in the source text.
3. **Automated Schema Consolidation:** As a knowledge graph grows organically, it can develop schema drift, where redundant or inconsistent labels and relationship types emerge (e.g., CITED_IN, MENTIONED_IN, REFERENCES). To maintain a clean and efficient structure, an LLM-powered agent will periodically

analyze the graph's schema. It will identify potential candidates for consolidation and propose these changes to a human operator for review and approval.[21] This semi-automated process ensures the graph remains logically coherent and optimized for querying as it scales.

### 2.3 The Active Learning Loop for Continuous NER Improvement

The performance of the domain-adapted NER model is a critical dependency for the entire system. A better NER model leads to a richer, more accurate knowledge graph. The **Active Learning (AL) Loop** is a human-in-the-loop process designed to improve the NER model's accuracy over time with minimal human annotation effort. AL is a proven technique for cost-effectively improving models in data-scarce domains by intelligently selecting the most informative examples for labeling.[36]

The AL workflow is seamlessly integrated into the ingestion pipeline:

1. **Identify Uncertainty:** When the spaCy NER model processes a new document chunk, it assigns a confidence score to each entity prediction. Any prediction that falls below a predefined confidence threshold is flagged as "uncertain."
2. **Query for Annotation:** These uncertain snippets are automatically added to a queue in a simple web-based annotation interface. A human domain expert can then quickly review these examples and either confirm or correct the predicted entity label. This focuses the expert's valuable time on the examples the model is most likely to learn from.
3. **Collect Verified Data:** Each human-verified annotation is added to a growing dataset of high-quality training examples.
4. **Periodic Re-training:** On a regular schedule (e.g., weekly or bi-weekly), the spaCy NER model is automatically re-trained. This training run uses the newly accumulated set of verified data, combined with a sample of the original training data to prevent the catastrophic forgetting discussed earlier. The newly improved and more accurate NER model is then automatically deployed back into the production ingestion pipeline.

This creates a virtuous cycle of improvement. However, the architecture of the AHR engine allows for an even more powerful, symbiotic relationship between its components. The NER model populates the KG, and the AL loop improves the NER model. Simultaneously, the Self-Evolving KG process analyzes user interactions to

identify new concepts and relationships. This connection can be leveraged to make the system learn not just new *instances* of entities, but entirely new *classes* of entities.

For example, if the KG evolution agent detects that the term "Form 10-K" is consistently appearing in important, user-validated contexts but is not a recognized entity type, it can flag this term as a candidate for a new entity label. This suggestion can be passed to the annotation team, prompting them to create a new LEGAL_FORM label in the NER schema. The system can then use the AL loop to start gathering and learning from annotated examples of this new entity type. This feedback mechanism, from the KG back to the NER model, transforms the system from one that simply gets better at what it already knows to one that can expand the very scope of its knowledge over time.

## Part III: Implementation Blueprint and Long-Term Trajectory

A sophisticated architecture is only valuable if it can be implemented effectively. This section provides a practical, actionable blueprint for building the Adaptive Hierarchical Reasoning (AHR) engine. It includes a phased roadmap to de-risk development, a detailed workflow diagram, a justified technology stack, and a vision for the system's future evolution.

### 3.1 A Phased Implementation Roadmap (MVP to Full System)

The AHR engine is a complex system with multiple interconnected components. Attempting to build the entire system in a single phase would be high-risk and capital-intensive. A phased approach is essential to manage complexity, deliver value iteratively, and align development with business goals. This roadmap breaks the project into three distinct, manageable phases, directly addressing the user's request for a system that is "not very hard to implement" by making the complexity incremental.[1]

**Table 1: Phased Implementation Roadmap**

| Phase | Key Objective | Core Components to Build | Success Metrics | Estimated Complexity |
|---|---|---|---|---|
| **Phase 1: Core HSR (MVP)** | Build a static but powerful RAG system based on the user's HSR design. | **Ingestion Pipeline:** Hybrid Structural-Semantic Chunking, Domain-Adapted NER (initial version). **Knowledge Base:** Neo4j Knowledge Graph Construction, Qdrant Vector Store. **Reasoning Core:** Hybrid Retrieval (Dense, Sparse, Graph), Evidence-Based CoR Generation. | Accuracy >90% on a benchmark dataset of representative legal/insurance queries. Average response time <3 seconds. 100% of decisions are traceable to source clauses. | Medium |
| **Phase 2: Self-Correction** | Introduce real-time response quality improvement and user feedback mechanisms. | **Adaptive Loop:** Implement the Self-Correction Loop (LLM-based Relevance Grader, Conditional Query Refinement). **Feedback UI:** Add simple user feedback mechanisms (e.g., thumbs up/down, rating). | Measurable reduction in "irrelevant context" errors during retrieval. User satisfaction scores (from feedback UI) improve by >10%. Demonstrated ability to self-correct on ambiguous queries. | Medium-High |
| **Phase 3: Full** | Enable | **Adaptive Loop:** | Demonstrable | High |

| | | | | |
|---|---|---|---|---|
| **Adaptivity** | long-term, continuous system evolution and learning. | Activate the Self-Evolving KG (insight-driven relationship updates, semi-automated schema consolidation). **Adaptive Loop:** Activate the Active Learning Loop for continuous NER model improvement. | growth in KG density (nodes and relationships) over time. NER model F1-score improves by >5% per quarter on a holdout evaluation set. System identifies and suggests new entity types for annotation. | |

This phased approach allows for the delivery of a valuable Minimum Viable Product (MVP) in Phase 1 that proves the core reasoning capabilities. Subsequent phases build upon this stable foundation, progressively adding the layers of intelligence and adaptivity that will define the system's long-term competitive advantage.

### 3.2 Detailed System Workflow and Architecture

To visualize the interplay of these components, a comprehensive workflow diagram is essential. This diagram maps the journey of data and queries through the fully realized AHR engine, integrating the foundational architecture with the adaptive feedback loops.

**(Note: A textual description of the flowchart is provided below, which would be rendered graphically in a formal report.)**

**Workflow Diagram: The Adaptive Hierarchical Reasoning Engine**

- **Part A: Ingestion & Knowledge Evolution**
  1. **Input:** A new document (PDF, DOCX, etc.) enters the system.
  2. **Hybrid Chunker:** The document is segmented first by structure (headers, tables), then by semantic content.
  3. **Domain-Adapted NER (spaCy):** The fine-tuned NER model processes each chunk, extracting entities and their relationships.

        - **Feedback Path (Active Learning):** If a prediction's confidence is low, the chunk and prediction are sent to the **Annotation UI Queue**.
            - A **Human Annotator** reviews the queue, confirms/corrects the label, and submits it.
            - The verified annotation is added to the **NER Training Dataset**.
            - Periodically, the **NER Model is Re-trained** and the updated model is deployed back to the NER component.
    4. **KG Population (Neo4j):** The chunks, entities, and relationships are used to create and link nodes in the Neo4j knowledge graph.
    5. **Vector Store Population (Qdrant):** The embeddings of the chunks are stored in the Qdrant vector database.
    6. **KG Evolution Agent (Offline Process):**
        - Analyzes stored user feedback and successful Q&A pairs.
        - Identifies new potential relationships and entity types.
        - Proposes updates to the KG and suggestions for the NER annotation team.
- **Part B: Query Processing & Self-Correction**
    1. **Input:** A user submits a natural language query via the **API (FastAPI)**.
    2. **Query Intelligence Module:** The query is deconstructed into a structured plan (intent hierarchy) and formal constraints.
    3. **Hybrid Retrieval Engine:** The system queries the knowledge base using all three methods:
        - **Dense Search:** Qdrant for semantic similarity.
        - **Sparse Search:** BM25 for keyword matching.
        - **Graph Search:** Neo4j for relational traversal.
    4. **Self-Correction Loop:**
        - The retrieved evidence chunks are passed to the **LLM Grader (Llama 3.1)**.
        - **Decision Node:** Are the chunks relevant?
            - **YES:** Proceed to the Generator.
            - **NO/AMBIGUOUS:** Trigger **Query Refinement**. The LLM rephrases the query, and the flow returns to the Hybrid Retrieval Engine.
    5. **LLM Generator (Llama 3.1):** The final, validated context is used to generate the answer.
    6. **Chain-of-Reasoning Formatter:** The generated answer is formatted with full evidence tracking and citations.
    7. **Output:** The final, explainable answer is returned to the user via the API. The user interaction and feedback are logged for the KG Evolution Agent.

### 3.3 The Open-Source Technology Stack and Rationale

The selection of each technology is a critical decision driven by performance, scalability, community support, and licensing. The AHR engine is designed to be built exclusively with powerful, commercially-permissive open-source tools, eliminating vendor lock-in and paid API dependencies.

**Table 2: Open-Source Technology Stack Comparison**

| Component | Recommended Tool | Rationale & Key Features | Viable Alternatives |
|---|---|---|---|
| **LLM** | **Llama 3.1 (8B/70B)** | State-of-the-art open-source model with a commercial-friendly license. Possesses strong reasoning capabilities, a large 128K token context window ideal for RAG, and is highly effective for fine-tuning tasks like the "grader" role in the self-correction loop.[39] | Mistral, Falcon. These are also powerful open-source models, but Llama 3.1 currently offers a leading combination of performance, context length, and community support.[40] |
| **Backend Framework** | **FastAPI** | A modern, high-performance Python framework built on Starlette (async speed) and Pydantic (automatic data validation and API documentation). Its API-first design is perfect for this system, promoting a clean separation of concerns and rapid | Django (a full-stack framework with more overhead, less ideal for a pure API backend), Flask (a microframework that is less performant for asynchronous tasks out-of-the-box).[45] |

| | | | |
|---|---|---|---|
| | | development.[42] | |
| **Vector Database** | **Qdrant** | An open-source vector database built in Rust, prized for its high performance and resource efficiency. It offers advanced filtering capabilities that can integrate with metadata from the KG and is explicitly designed for modern RAG workflows.[23] | Milvus (highly scalable, enterprise-grade), Weaviate (cloud-native, GraphQL interface). Both are strong contenders, but Qdrant's focus on performance and RAG-centric features gives it an edge for this use case.[47] |
| **Graph Database** | **Neo4j Community Edition** | The leading native graph database, offering exceptional performance for traversing relationships (index-free adjacency). Its mature ecosystem, extensive documentation, and the powerful Cypher query language make it ideal for building and evolving complex knowledge graphs.[17] | ArangoDB (multi-model, offering document and graph in one), Dgraph (distributed graph database). While capable, Neo4j's singular focus on the graph model provides deeper optimization and a larger community for this specific task.[17] |
| **NER & NLP** | **spaCy** | A production-grade NLP library known for its speed and efficiency. It provides an excellent, well-documented pipeline for training custom NER models and is robust enough for a high-throughput ingestion system. Its large community ensures extensive | NLTK (more academic/educational, less optimized for production speed), Hugging Face Transformers (extremely powerful but can be more complex to integrate into a custom pipeline compared to spaCy's streamlined workflow). |

| | | | |
|---|---|---|---|
| | | support.[11] | |

### 3.4 Future Horizons - Towards Full Autonomy

The three-phase implementation plan establishes a clear path to building a state-of-the-art adaptive intelligence system. However, the architecture is designed with future evolution in mind, paving the way for even more advanced, autonomous capabilities.

- **Implementing Full Reinforcement Learning from Human Feedback (RLHF):** The user feedback collected in Phase 2 creates a valuable dataset of human preferences. In a future phase, this data can be used to train a dedicated **Reward Model**. This model would learn to predict a human preference score for any given response. The AHR engine could then be optimized using RL algorithms (like PPO) against this reward model, transitioning from the rule-based self-correction of Phase 2 to a more nuanced, policy-based optimization. This would allow the system to learn complex behaviors beyond simple relevance checking, such as adjusting its tone, verbosity, or level of detail to match user preferences.[29]
- **Advanced Agentic Workflows:** The current design uses agents for offline analysis (KG evolution). Future iterations could deploy more sophisticated, online agents. These agents could be granted tools, allowing them to perform multi-step reasoning tasks that go beyond simple retrieval. For example, upon encountering a conflicting piece of information, an agent could be empowered to perform a web search to find a third source for verification before synthesizing its final answer, moving from a reasoning engine to a problem-solving one.[34]
- **True Multi-Modal Reasoning:** The initial hybrid chunking strategy respects the structure of tables and figures. The next evolutionary step would be to natively ingest and reason about the *content* of these elements. This would involve integrating computer vision models to interpret charts, extract data from tables directly into structured formats within the KG, and understand the semantic content of images, fully realizing the multi-modal potential of the HSR vision.[1]

By following this blueprint, the initial concept of a Hierarchical Semantic Reasoning system can be realized and then elevated into a truly novel and powerful Adaptive Hierarchical Reasoning engine. It provides a pragmatic, phased, and entirely open-source path to building a system that not only understands documents but

learns, adapts, and grows more intelligent with every interaction.

**Works cited**


1. HackerX 1[1].docx
2. 7 Chunking Strategies in RAG You Need To Know - F22 Labs, accessed on August 4, 2025, https://www.f22labs.com/blogs/7-chunking-strategies-in-rag-you-need-to-know/
3. Optimizing RAG with Advanced Chunking Techniques - Antematter, accessed on August 4, 2025, https://antematter.io/blogs/optimizing-rag-advanced-chunking-techniques-study
4. Semantic Chunking in Complex Documents | by Gal Lellouche - Medium, accessed on August 4, 2025, https://gal-lellouche.medium.com/semantic-chunking-in-complex-documents-cc49b0cde4ea
5. Chunking strategies for RAG tutorial using Granite - IBM, accessed on August 4, 2025, https://www.ibm.com/think/tutorials/chunking-strategies-for-rag-with-langchain-watsonx-ai
6. Chunk and vectorize by document layout - Azure AI Search - Microsoft Learn, accessed on August 4, 2025, https://learn.microsoft.com/en-us/azure/search/search-how-to-semantic-chunking
7. How to split text based on semantic similarity | 🦜 LangChain, accessed on August 4, 2025, https://python.langchain.com/docs/how_to/semantic-chunker/
8. Vprashant/s2-chunking-lib: A library for structural-semantic chunking of documents. - GitHub, accessed on August 4, 2025, https://github.com/Vprashant/s2-chunking-lib
9. E-NER — An Annotated Named Entity Recognition Corpus of Legal Text - ACL Anthology, accessed on August 4, 2025, https://aclanthology.org/2022.nllp-1.22.pdf
10. Training LayoutLM from Scratch for Efficient Named-Entity Recognition in the Insurance Domain - arXiv, accessed on August 4, 2025, https://arxiv.org/html/2412.09341v1
11. Named Entity Recognition: A Practical Guide - Label Your Data, accessed on August 4, 2025, https://labelyourdata.com/articles/data-annotation/named-entity-recognition
12. Fine-Tuning SpaCy Models: Customizing Named Entity Recognition for Domain-Specific Data | by Wiem Souai | UBIAI NLP | Medium, accessed on August 4, 2025, https://medium.com/ubiai-nlp/fine-tuning-spacy-models-customizing-named-entity-recognition-for-domain-specific-data-3d17c5fc72ae
13. Training a Custom Named-Entity-Recognition (NER) Model with spaCy | by Harisudhan.S, accessed on August 4, 2025, https://medium.com/@speaktoharisudhan/training-a-custom-named-entity-reco

gnition-ner-model-with-spacy-072ae59f9ed2
14. Train NER with Custom training data using spaCy. | Towards Data Science, accessed on August 4, 2025, https://towardsdatascience.com/train-ner-with-custom-training-data-using-spacy-525ce748fab7/
15. How to Alleviate Catastrophic Forgetting in LLMs Finetuning? Hierarchical Layer-Wise and Element-Wise Regularization - arXiv, accessed on August 4, 2025, https://arxiv.org/html/2501.13669v2
16. How to prevent catastrophic forgetting in fine tuned large language models?, accessed on August 4, 2025, https://discuss.huggingface.co/t/how-to-prevent-catastrophic-forgetting-in-fine-tuned-large-language-models/135153
17. Top 10 Open Source Graph Databases in 2025 - GeeksforGeeks, accessed on August 4, 2025, https://www.geeksforgeeks.org/blogs/open-source-graph-databases/
18. Neo4j Graph Database & Analytics | Graph Database Management System, accessed on August 4, 2025, https://neo4j.com/
19. 10 Best Open Source Graph Databases in 2025 - Index.dev, accessed on August 4, 2025, https://www.index.dev/blog/top-10-open-source-graph-databases
20. Knowledge Graph - Graph Database & Analytics - Neo4j, accessed on August 4, 2025, https://neo4j.com/use-cases/knowledge-graph/
21. Knowledge Graph Extraction and Challenges - Graph Database & Analytics - Neo4j, accessed on August 4, 2025, https://neo4j.com/blog/developer/knowledge-graph-extraction-challenges/
22. How to Improve Multi-Hop Reasoning With Knowledge Graphs and LLMs - Neo4j, accessed on August 4, 2025, https://neo4j.com/blog/genai/knowledge-graph-llm-multi-hop-reasoning/
23. The 7 Best Vector Databases in 2025 | DataCamp, accessed on August 4, 2025, https://www.datacamp.com/blog/the-top-5-vector-databases
24. Top 5 Vector Databases in 2025 - CloudRaft, accessed on August 4, 2025, https://www.cloudraft.io/blog/top-5-vector-databases
25. Best Vector Database for RAG : r/vectordatabase - Reddit, accessed on August 4, 2025, https://www.reddit.com/r/vectordatabase/comments/1hzovpy/best_vector_database_for_rag/
26. Understanding Semantic and Hybrid Search with Python and pgvector | by Hasan Sajedi, accessed on August 4, 2025, https://medium.com/@hasansajedi/understanding-semantic-and-hybrid-search-with-python-and-pgvector-0967e83803e6
27. Corrective and Self-Reflective RAG with LangGraph | by Cole McIntosh - Medium, accessed on August 4, 2025, https://medium.com/@colemcintosh6/corrective-and-self-reflective-rag-with-langgraph-364b7452fc3e
28. Self-Reflective RAG with LangGraph - LangChain Blog, accessed on August 4, 2025, https://blog.langchain.com/agentic-rag-with-langgraph/

29. What is RLHF? - Reinforcement Learning from Human Feedback Explained - AWS, accessed on August 4, 2025, https://aws.amazon.com/what-is/reinforcement-learning-from-human-feedback/
30. Reinforcement Learning from Human Feedback (RLHF): A Practical Guide with PyTorch Examples | by Sam Ozturk, accessed on August 4, 2025, https://themeansquare.medium.com/reinforcement-learning-from-human-feedback-rlhf-a-practical-guide-with-pytorch-examples-139cee11fc76
31. RLHF 101: A Technical Tutorial on Reinforcement Learning from Human Feedback, accessed on August 4, 2025, https://blog.ml.cmu.edu/2025/06/01/rlhf-101-a-technical-tutorial-on-reinforcement-learning-from-human-feedback/
32. Build a Knowledge Graph in NLP - GeeksforGeeks, accessed on August 4, 2025, https://www.geeksforgeeks.org/nlp/build-a-knowledge-graph-in-nlp/
33. How Knowledge Graphs Transform Machine Learning in 2025 - TiDB, accessed on August 4, 2025, https://www.pingcap.com/article/machine-learning-knowledge-graphs-2025/
34. Building Self-Evolving Knowledge Graphs Using Agentic Systems | by Modern Data 101, accessed on August 4, 2025, https://medium.com/@community_md101/building-self-evolving-knowledge-graphs-using-agentic-systems-48183533592c
35. LLM Knowledge Graph Builder: From Zero to GraphRAG in Five Minutes - Neo4j, accessed on August 4, 2025, https://neo4j.com/blog/developer/graphrag-llm-knowledge-graph-builder/
36. Re-weighting Tokens: A Simple and Effective Active Learning Strategy for Named Entity Recognition - ACL Anthology, accessed on August 4, 2025, https://aclanthology.org/2023.findings-emnlp.847/
37. Utilizing active learning strategies in machine-assisted annotation for clinical named entity recognition - Oxford Academic, accessed on August 4, 2025, https://academic.oup.com/jamia/article/31/11/2632/7724491?rss=1
38. An active learning-enabled annotation system for clinical named entity recognition - PMC, accessed on August 4, 2025, https://pmc.ncbi.nlm.nih.gov/articles/PMC5506567/
39. 9 Top Open-Source LLMs for 2024 and Their Uses | DataCamp, accessed on August 4, 2025, https://www.datacamp.com/blog/top-open-source-llms
40. 10 Best Open-Source LLMs for Scalable and Ethical AI Development - Kanerika, accessed on August 4, 2025, https://kanerika.com/blogs/open-source-llms-models/
41. Top 10 open source LLMs for 2025 - NetApp Instaclustr, accessed on August 4, 2025, https://www.instaclustr.com/education/open-source-ai/top-10-open-source-llms-for-2025/
42. Alternatives, Inspiration and Comparisons - FastAPI, accessed on August 4, 2025, https://fastapi.tiangolo.com/alternatives/
43. FastAPI vs. Flask: Python web frameworks comparison and tutorial - Contentful, accessed on August 4, 2025, https://www.contentful.com/blog/fastapi-vs-flask/

44. How does FastAPI compare to other Python Web frameworks? - FatCat Coders, accessed on August 4, 2025, https://fatcatremote.com/it-glossary/python/fastapi-vs-python-web-frameworks
45. Comparison of FastAPI with Django and Flask - GeeksforGeeks, accessed on August 4, 2025, https://www.geeksforgeeks.org/python/comparison-of-fastapi-with-django-and-flask/
46. Which Is the Best Python Web Framework: Django, Flask, or FastAPI? | The PyCharm Blog, accessed on August 4, 2025, https://blog.jetbrains.com/pycharm/2025/02/django-flask-fastapi/
47. Top 9 Vector Databases as of July 2025 - Shakudo, accessed on August 4, 2025, https://www.shakudo.io/blog/top-9-vector-databases
48. Best 17 Vector Databases for 2025 [Top Picks] - lakeFS, accessed on August 4, 2025, https://lakefs.io/blog/12-vector-databases-2023/
49. The Top Graph Database Companies to Watch in 2025 - Syntaxia, accessed on August 4, 2025, https://www.syntaxia.com/post/the-top-graph-database-companies-to-watch-in-2025
50. spaCy 101: Everything you need to know, accessed on August 4, 2025, https://spacy.io/usage/spacy-101
51. How RLHF, RAG and Instruction Fine-Tuning Shape the Future | GigaSpaces AI, accessed on August 4, 2025, https://www.gigaspaces.com/blog/rlhf-rag-and-instruction-fine-tuning
52. Optimizing RAG with Reward Modeling and RLHF - arXiv, accessed on August 4, 2025, https://arxiv.org/html/2501.13264v1
53. Feedback Loops With Language Models Drive In-Context Reward Hacking - arXiv, accessed on August 4, 2025, https://arxiv.org/html/2402.06627v3