Proposal: Integrating a Neo4j Knowledge Graph with the Golf Rules RAG System

Introduction and Background

The current Golf Rules Q&A system leverages Retrieval-Augmented Generation (RAG) to answer questions. It uses an LLM with a vector-based retriever over two data sources: (1) the official Rules of Golf (embedded from a PDF rulebook) and (2) a collection of thousands of email Q&As (embedded question-answer pairs). This approach brings relevant text into the LLM's context so it can generate answers grounded in the rulebook. While this system works reasonably well, it faces notable challenges:

- Retrieval Accuracy: Pure vector similarity search sometimes returns incomplete or tangential sections of the rulebook. Ambiguous queries or subtle phrasing can lead to missing the most applicable rule or failing to retrieve a necessary related rule. Vector embeddings alone struggle with ambiguous context and complex relationships – for example, a question about relief might retrieve one rule while overlooking an exception noted in a different rule.
- Rule Interdependencies: The Rules of Golf are highly interrelated – many rules reference others (exceptions, penalties, definitions). The current RAG system has no explicit representation of these connections. It treats the rulebook as isolated text chunks, which makes it hard to capture prerequisites or exceptions unless the text explicitly appears in the same chunk. This can lead to answers that miss important qualifiers or related rules.
- Need for Structured Reasoning: Some questions require applying logical conditions or multi-step reasoning (e.g., "If X, then Y, except in case Z"). An LLM on raw text may struggle to apply such logic consistently, especially if relevant facts are scattered across documents. There is no structured knowledge to guide the reasoning process or ensure all relevant rules are considered.

To address these issues, we propose augmenting the existing RAG architecture with a Neo4j-based knowledge graph of the golf rules. This approach will supplement the current vector retriever with a structured graph of rule relationships – not replacing the RAG system, but enhancing it. By introducing a knowledge graph layer, we aim to preserve the strengths of the current pipeline (flexible semantic search over unstructured text) while adding a layer of explicit rule connections for improved accuracy and reasoning.

Why a Knowledge Graph? Knowledge graphs (KGs) excel at representing complex relationships in data as nodes and edges. In domains like legal or rules-based systems, a KG can model the web of references (e.g. which rules are related or conditional on others) in a way that a vector database cannot . In fact, vector search is known to struggle with ambiguous queries and lacks explicit logical structure, whereas a knowledge graph encodes relationships that help connect concepts and apply logic . By combining the two, we get the best of both worlds: the semantic matching capability of embeddings and the explicit relational knowledge of a graph . This hybrid approach has been recognized as a powerful enhancement to RAG systems, yielding more depth, context, and explainability in retrieved information . In short, the knowledge graph will guide and enrich retrieval without replacing the core RAG mechanism that is already in place.

Importantly, this integration is incremental and low-risk – it will reuse the existing rulebook and Q&A data. The knowledge graph will act as an additional component that interfaces with the current pipeline. Stakeholders can expect improved accuracy (through better context retrieval and completeness) and more transparent reasoning, all while leveraging the investment in the current RAG setup.

Rationale for Knowledge Graph Augmentation (Not Replacement)

Introducing a knowledge graph is motivated by the limitations observed in the current RAG approach and the potential gains from structured knowledge. The key rationale points include:

- Structured Relationships for Interconnected Rules: A knowledge graph will explicitly capture how rules relate (e.g. which rule is an exception to another, which scenario a rule applies to). This structured network addresses the issue of interdependencies by making them first-class citizens in the data model. When the LLM needs to answer a question, it won't be limited to whatever a single retrieved passage contains; the graph can surface connected rules that should also be considered. This directly tackles the problem of missing context in answers. As noted in recent industry analysis, KGs help connect concepts, resolve ambiguity, and apply logic in ways that pure vector search cannot .
- Complementary Strengths – Not Throwing Away RAG: The knowledge graph is meant to supplement the existing retrieval, not to supplant it. We recognize that semantic embeddings are excellent for parsing natural language queries and pulling in roughly relevant passages, especially from the unstructured email Q&A data. However, the KG can provide an additional filter or guide to ensure the retrieved context is precise and complete. By combining both, the system can retrieve knowledge more effectively and ground responses in facts while also being able to provide logical connections and explanations . In practice, this means the LLM's input context can include not just a snippet from Rule 5.3, but also a linked Rule 3.1 that contains an exception, if relevant – yielding a more comprehensive answer.
- Improved Reasoning & Accuracy: The structured nature of a graph allows some reasoning that the LLM might otherwise miss. For example, if a question involves a scenario that triggers multiple rules (like a ball in a penalty area with a lost ball scenario), the graph can help identify all applicable rules and outcomes systematically. This reduces the chance of the LLM hallucinating connections or overlooking a rule. Knowledge Graphs bring explicit, logic-based reasoning into the pipeline, which can help the AI apply rules more like a human expert would . Early evidence in similar domains (like legal QA) shows that integrating KGs with RAG can minimize errors and hallucinations by ensuring all relevant knowledge is retrieved and by providing an explainable chain of reasoning .
- Real-World Validation: The idea of using a knowledge graph for golf rules Q&A is not just theoretical. In fact, a recent product Play Today – Golf AI built a Golf

Rules Knowledge Graph connecting all rules, definitions, clarifications, and procedures, and runs an AI assistant on top of it . This validates that representing the Rules of Golf in a graph structure is feasible and beneficial. Our approach will similarly leverage a graph model for the rulebook, but integrate it with our existing system for a hybrid solution. By augmenting rather than replacing, we ensure continuity with our current RAG approach while injecting the proven advantages of a knowledge graph.

In summary, the knowledge graph will enhance retrieval accuracy, handle rule interdependencies explicitly, and enable structured reasoning, all without discarding the current investment in RAG. Instead of a monolithic change, we add a complementary component – a strategy aligned with best practices for advanced RAG systems (sometimes dubbed "Graph-RAG") where knowledge graphs and embeddings work in tandem .

Hybrid System Architecture Overview

In the proposed solution, the overall architecture becomes a hybrid RAG + KG system. The high-level flow is as follows:

1. User Query Ingestion: A user asks a question in natural language (e.g., "My ball ended up in a water hazard and I can't play it – what are my options?"). The query is received by the system as usual.
2. Dual Retrieval Mechanisms: The query will be processed by two parallel retrieval pathways:
   - (A) Unstructured Vector Retrieval: Using the existing vector database of the rulebook and Q&A, we perform embedding-based similarity search (and/or keyword search) to fetch the top relevant text snippets. This is the same process currently in place – e.g., retrieving a paragraph from the rulebook about penalty relief, or a similar Q&A email if one exists.
   - (B) Knowledge Graph Retrieval: In parallel, we leverage the Neo4j knowledge graph to retrieve a set of structured relevant nodes and relationships. There are a couple of ways to do this (which we will evaluate):
   - If the query explicitly mentions a rule number or a known concept, we directly find that in the graph. For example, a query referencing "Rule 17" or "water hazard" can be mapped to the node representing the Penalty Area rule or the Penalty Area concept.
   - If the query is phrased in general terms, we can use a lightweight NLP step (possibly using the LLM itself or a simple keyword mapping) to identify key entities in the question. For example, detect that "water hazard" corresponds to the concept now termed "Penalty Area". We can then query Neo4j (using a Cypher query or full-text index on node names) to find matching nodes. This is akin to an entity extraction followed by graph lookup, an approach demonstrated in similar LLM+KG pipelines .
   - Alternatively, we might use the initial vector retrieval result as a clue: if the top vector hit is a certain rule, we find that rule's node in the graph. This can serve as a starting point for traversal.
3. Graph Traversal for Context Expansion: Once we have one or more entry nodes in the graph that are relevant to the query, we perform a traversal to gather a

subgraph of connected knowledge. The idea is to collect all closely related nodes that might be needed to fully answer the question. For example, if the entry point is the node for “Rule 17: Penalty Areas,” the graph traversal would fetch: any exception rules linked to Rule 17, any prerequisite conditions (like a node for “ball in penalty area” scenario), and any consequences/actions (like a node representing the “one-stroke penalty drop” action). This traversal would typically be bounded (e.g., one hop outwards) to avoid an explosion of data, focusing on immediate connections. The result is a neighborhood subgraph of the query context. In practice, this could be implemented by a Cypher query that finds all nodes directly connected to the starting node(s) . If multiple entry nodes exist (say the question involves two concepts), we can collect neighbors for each. The Neo4j database will be running locally (or on a server accessible to the application) to allow fast graph queries with low latency.

4. Context Synthesis: The outputs of the two retrieval pathways are then combined into a single context for the LLM. The unstructured retrieval provides verbatim text segments (e.g. exact rule paragraphs or Q&A answers), while the graph retrieval provides a curated set of related rules and facts. We will convert the graph-based results into a textual form suitable for the LLM input – likely by retrieving the actual rule text for each rule node and a short description for scenario/action nodes. This combined context ensures the LLM sees both the primary relevant rule text and any supplemental information (related rules, definitions, etc.). For example, the context might include: the full text of Rule 17, the text of the exception clause from Rule 1.1 if Rule 17 has an exception there, and a note like “Penalty Area – one-stroke penalty relief applies” from an action node. The mechanism to combine can be as simple as concatenating the texts or a more structured prompt sectioning (e.g., “Relevant Rules:” and “Related Provisions:”). The approach described mirrors the hybrid retriever concept where structured and unstructured data are merged for final answer generation .

5. LLM Answer Generation: The LLM (e.g., GPT-4 or another chosen model) takes the compiled context and the user’s question, and produces an answer. Because the context now is richer and more complete, the LLM can generate an answer that is more accurate and comprehensive. It can explicitly reference the correct rule numbers and account for any caveats or exceptions (since those are present in the prompt). The generation step itself remains the same as current RAG (prompting the model with context + question), so no change in the LLM is required – the improvement comes from better context.

6. Output Delivery: The answer is returned to the user, ideally with citations or references to the rule numbers that were used (as is current practice). The knowledge graph can also help produce these references; since we know exactly which rule nodes were included, we can ensure the answer references them. The end-user experience remains a natural language answer with supporting rule references, now with higher confidence that all relevant rules were considered.

System Components: Technically, the system will consist of:

- The existing vector database (for example, a Pinecone/FAISS/Chroma index or even Neo4j’s vector index if we migrate it) containing embeddings of rulebook sections and Q&A pairs.

- A Neo4j graph database running locally (for development we can use Neo4j Desktop or a Docker container; for production, Neo4j Aura or a managed instance is an option, but local is fine initially). This DB holds the knowledge graph with all rules and their relationships. We will define a clear data schema (detailed in the next section) and load the rulebook data into Neo4j accordingly.
- The application logic (Python or Node, etc.) that orchestrates retrieval: it will connect to Neo4j (via the Bolt or HTTP driver) to issue queries for the graph part, and to the vector index for the embedding search part. This logic will implement the hybrid retrieval routine described.
- The LLM (likely accessed via an API if using OpenAI, or locally if using an open-source model) that generates answers from the provided context.
- Optionally, a small cache or index could be used for common queries or to avoid repeated graph queries for the same or similar questions, but this is an optimization detail.

A diagram of this hybrid flow would show the user query feeding into two retrieval components (vector search and graph search), then merging results to the LLM. In essence, the architecture extends the classic RAG workflow by inserting a graph traversal step alongside document retrieval, before the answer generation. This hybrid approach (vector + graph) has been illustrated in literature and by Neo4j's own examples , confirming that it is a practical design. The key integration effort will be ensuring that the graph query results can be seamlessly transformed into additional context for the LLM.

It's worth noting that Neo4j itself now supports vector indexing and full-text search on nodes, meaning in principle we could unify all retrieval in Neo4j . For example, we might store each rule's text as a node property and have Neo4j handle both semantic and graph queries. However, to minimize disruption, we can initially keep the vector search on the existing system and use Neo4j purely for relationships. We can decide later if consolidating into one system is beneficial (it might simplify architecture, but it's optional).

Why Local Neo4j? Running Neo4j locally (or on a controlled server) ensures low-latency access and data privacy (important since the rulebook/Q&A may be proprietary). The integration is straightforward: the application will use Neo4j's driver to run Cypher queries. For the prototype phase, we can set this up quickly on a developer machine. In production, a containerized Neo4j instance could be deployed alongside the app or as a managed service.

In summary, the hybrid architecture will enrich the RAG pipeline with a graph-based retrieval step, yielding a more intelligent selection of context for the LLM. The next sections detail how we will construct the knowledge graph and use it within this architecture.

Knowledge Graph Schema Design for the Rulebook

Structuring the entire rulebook into a graph requires defining what the nodes and edges represent. The goal is to model rules, scenarios, and outcomes in a way that reflects the rulebook's logic. Below is the proposed schema:

Node Types:

- Rule: Each Rule (and possibly each sub-rule or clause) in the Rules of Golf becomes a node. For example, "Rule 17 – Penalty Areas" would be a node, and if needed, sub-sections like "Rule 17.1 – Options for Ball in Penalty Area" could be separate nodes linked under the main rule. Each Rule node will have properties such as the rule number, title, and possibly the full text or a summary of the rule. The full text is already in our documents; we may store the text in the node for convenience or store an identifier that links to the text in the vector store.
- Scenario/Condition: These nodes represent specific conditions or scenarios addressed by the rules. A scenario is essentially a context in which a rule applies. Examples: "Ball in Penalty Area", "Ball Out of Bounds", "Dangerous Animal Condition", "Ball Moved by Outside Influence", "Stroke Play" vs "Match Play" (formats could be considered conditions that affect some rules). Many of these scenarios are defined terms in the rulebook's Definitions section – we can include key definitions as Scenario nodes as well. Each scenario node would have a name/description property.
- Action/Outcome: These nodes represent penalties, relief options, or other outcomes that result from rules. Examples: "One-Stroke Penalty", "Stroke-and-Distance Relief (Replay Shot)", "Free Relief Drop", "No Penalty". Essentially, if a rule prescribes an action or consequence, that action is a node. We might also include "Allowable Action" nodes like "You may move a loose impediment" or "Player must replay the shot".

(Optionally, we could have a node type for Definitions or Clarifications to capture the official definitions and interpretations. However, many of those would surface as Scenario nodes or be linked as attributes. For the core schema, we focus on rules, scenarios, actions as given in the prompt, which align well with what we need.)

Edge Types:

- Exception: An edge that denotes an exception relationship between rules. If Rule A has an exception detailed in Rule B (meaning Rule B is applied in a case where Rule A would normally apply, overriding it), we create an "EXCEPTION" edge from Rule A to Rule B (or vice versa, depending on readability – typically "Rule A has exception Rule B"). For example, "Rule 16.1 (Abnormal Course Conditions relief) has an exception in Rule 16.2 (Dangerous Animal Condition)". In the graph we might link Rule 16.1 –[EXCEPTION]→ Rule 16.2 to indicate that although Rule 16.1 provides free relief generally, Rule 16.2 is a special case that always allows relief even in situations that would otherwise not qualify. This way, a traversal starting at 16.1 would find 16.2 as an exception to consider.
- Prerequisite / AppliesIf: This edge type connects a rule to a scenario or condition that must be true for the rule to apply. It can also connect a specific clause to another rule that needs to be consulted. For example, "Rule 6.3 (Ball Used in Play) applies only in Stroke Play format" – we could represent that as Rule 6.3 –

[APPLIES_IN]→ Scenario:Stroke Play. Or a rule that says "If X, you must do Y" might have an edge linking Scenario X to Rule Y. Prerequisite edges help capture conditions like "if a ball is lost (Scenario) then Rule 18.2 applies" or "Rule 18.2 requires condition 'ball lost or out of bounds'". We might label these edges CONDITION or TRIGGER relationships. The exact naming can be refined, but the idea is to connect rules to the scenarios that trigger them.

- Consequence / Implies: This edge type connects a rule to an outcome (action or penalty) that results from that rule. For instance, Rule 18.2 (Lost Ball) → Action: Stroke-and-Distance Penalty (meaning if a ball is lost, the consequence is stroke-and-distance). Another: Rule 11.1 (Ball in Motion Hits Person) might connect to Action: No Penalty (exception) as a consequence. Label names could be IMPLIES, PENALTY, or RESULTS_IN. These edges allow us to quickly find "what happens under this rule."
- Reference / Related: A generic relationship for cross-references that are not explicitly exceptions or prerequisites. The Rules often say "See Rule X.Y" for related topics. We can capture those as RELATED_TO edges between rule nodes. This ensures that if a rule explicitly mentions another, our graph knows about it. This could help when a question touches a broad area – the graph can pull in related rules.
- Defines / DefinedIn: If we include definition nodes, we'd have edges linking a term to the rule that defines it or uses it. For example, Scenario:Loose Impediment –[DEFINED_IN]→ Rule D (Definitions) or to a specific rule that elaborates it. This might be optional detail; many definitions might just become scenario nodes and be linked to rules via the CONDITION edges.

Each edge will have a type and potentially a short description or properties if needed (though likely just the type is enough since the meaning is clear by context). The direction of edges will be chosen for convenience of query; often one might not even need directed traversal if queries are written to ignore direction (or we can add reciprocal edges if needed for ease of use).

Graph Schema Example: To illustrate, consider the case of Dangerous Animal Condition mentioned earlier:

- Node: Rule 16.2 – Dangerous Animal Condition (this rule states that you can take relief from dangerous animals anywhere on the course).
- Node: Rule 16.1 – Abnormal Course Conditions (general relief rule, which has an exception for dangerous animals).
- Node: Scenario: Dangerous Animal (a condition where an animal interferes with play).
- Node: Action: Free Relief (the outcome – you get to drop without penalty).
- We would have edges: Rule 16.1 –[EXCEPTION]→ Rule 16.2 (Rule 16.2 is an exception to 16.1's normal restrictions). Also Rule 16.2 –[APPLIES_TO]→ Scenario:Dangerous Animal (the rule is triggered by that scenario). And Rule 16.2 –[RESULTS_IN]→ Action:Free Relief (the rule's effect). Additionally, Rule 16.1 –[RESULTS_IN]→ Action:Free Relief as well (because general relief is also free drop, unless overridden by other conditions). Now, if a question is asked "A dangerous snake

is next to my ball, do I get relief even though I couldn't play the shot anyway?", our graph would let us find Rule 16.2 from the "snake/animal" scenario, see it's an exception to 16.1's normal "no relief if no shot" limitation, and gather that free relief is allowed. The LLM, given both Rule 16.1 and 16.2 text, will then answer correctly that the player is allowed relief despite the normally unplayable situation – referencing Rule 16.2 explicitly. Without the graph, a vector search might fetch one rule or the other but not both, leading to a partial or incorrect answer.

Another example: Lost Ball and Out of Bounds (Rule 18.2):

- Node: Rule 18.2 – Ball Lost or Out of Bounds.
- Node: Scenario: Ball Out of Bounds; Node: Scenario: Ball Lost.
- Node: Action: Stroke-and-Distance (the penalty and procedure).
- Edges: Rule 18.2 –[TRIGGERED_BY]→ Scenario: Ball Lost, and … → Scenario: Ball Out of Bounds. Also Rule 18.2 –[RESULTS_IN]→ Action: Stroke-and-Distance.
- Additionally, Rule 18.3 – Provisional Ball might be RELATED to Rule 18.2, etc.

Now a question "I can't find my ball, what do I do?" would via graph know to bring Rule 18.2 (lost ball) and possibly Rule 18.3 (provisional ball, if relevant) into context, ensuring the answer explains stroke-and-distance penalty.

Entire Rulebook Coverage: We will create a Rule node for every rule (the Rules of Golf are numbered 1 through 24, each with sub-clauses). We will also create nodes for each Definition in the rulebook's Definitions section as Scenario nodes (because those are essentially conditions/entities used by the rules). Furthermore, common penalty or relief actions (which are spread in the rules) will be standardized as Action nodes so they can be linked from multiple rules. Edges will be created wherever a rule:

- references another rule (we'll decide if it's exception, prerequisite, or related based on context keywords like "except", "see", "must first" etc.),
- has a penalty or relief (implies action),
- depends on a game format (some rules differ in match play vs stroke play, so we might link those scenario nodes as conditions).

The resulting graph will be a rich network of the entire rulebook. As an external point of reference, a similar knowledge graph built by Play Today encompassed rules, definitions, clarifications, etc., illustrating that such comprehensive modeling is practical . Our schema aligns with that idea but is tailored to the client's needs focusing on rules, scenarios, and actions which cover the essential relationships (implications, prerequisites, exceptions). The design avoids overly complex ontology formality – it's a pragmatic schema capturing what we need for Q&A support (this keeps the project scope manageable while still benefiting from graph structure).

We will document the schema in detail (node and edge types and their meanings) so that it's clear to all stakeholders and for future maintainers. Next, we discuss how we will populate this graph from the existing rule texts.

Building the Knowledge Graph: Extraction and Ingestion

Constructing the knowledge graph from the unstructured rulebook is a critical step. We plan to use a combination of LLM-based parsing and symbolic rule mining (pattern-based extraction) to do this efficiently:

1. LLM-Assisted Rule Parsing: Given the size and complexity of the rulebook, manually encoding it into the graph would be time-consuming. Instead, we will leverage Large Language Models to help interpret the rules and suggest graph entries. Recent experiments have shown that LLMs can identify entities and relationships from text and output structured representations suitable for graph databases . In fact, an "LLM Graph Transformer" module has been added to frameworks like LangChain to automate knowledge graph construction from documents . We can utilize such tools or design custom prompts to extract structured data from each rule. For example, we can prompt the LLM with: "Read the following rule text and identify: (a) any references to other rules (and whether they are exceptions or related), (b) any conditions or scenarios mentioned, (c) any penalties or actions specified." The LLM can output a JSON or triple list that we then convert to Cypher commands to create nodes and edges. Using GPT-4 (with function calling perhaps) could yield quite accurate extractions given its understanding of the language. Of course, we will verify and correct as needed, but this could bootstrap the graph significantly.

Specifically, we could proceed rule by rule: for each rule's text, the LLM would list something like:

- Rule Node: {number, title}.
- References: e.g. "Exception: Rule X.Y", "See also Rule Z".
- Scenarios: e.g. "when ball is in [condition]", "in match play," etc. These could map to existing definition nodes or new scenario nodes.
- Actions/Penalties: e.g. "penalty of one stroke", "must replay the shot", etc.

This information can then be translated into our schema: create nodes if not exist (for rules, scenarios, actions) and create edges of the appropriate type between them. We will take advantage of the LLM's "profound understanding of language and context" to automate significant parts of knowledge graph creation , making the process much faster than purely manual entry.

2. Symbolic and Pattern-based Extraction: In parallel, we can use simpler automated methods for certain obvious relationships: the rulebook often uses standard phrasing. For example:

- The word "Exception" in a rule usually introduces an exception clause referencing another rule or condition. We can parse text after "Exception:" keywords.
- References like "(see Rule 14.3)" can be regex-found to link rules in a RELATED_TO edge.
- Penalty statements like "Penalty for breach: General Penalty" are standardized at the end of rules – we know "General Penalty" means a certain action (two-stroke or loss of hole). We can map that to an Action node.

- Definitions are clearly marked in a Definitions section; we can scrape those definitions to create Scenario nodes upfront, and then link rules to those scenarios by finding where those defined terms appear in rules. For instance, if the term “Loose Impediment” is defined, and Rule 15 deals with it, connect accordingly.

By writing some parsing scripts (using Python with regex or text analysis), we can capture a baseline of graph relationships. This is more error-prone for complex sentences, but it will catch straightforward cases and give the LLM less to figure out.

3. Combined Approach with Verification: We anticipate using the LLM to handle the nuanced language (identifying what type of relationship is implied) and using pattern matching for straightforward cross-references and glossary terms. The output of these processes will be reviewed by our team (including, if possible, a subject matter expert in golf rules to ensure no misinterpretation). The knowledge graph’s accuracy is paramount since it will guide the LLM’s answers. We will likely iterate: build an initial graph automatically, then manually inspect and correct obvious mistakes or omissions. Fortunately, the rulebook is finite (~24 rules plus definitions); even if the LLM gets 80% right, we can manually fix the rest within a reasonable timeframe.

We might start small (a subset of rules) for testing, as described in the prototype phase, and later scale to the full rulebook once our extraction method is refined.

For implementation, we will leverage existing tools where possible. For example, LangChain’s LLMGraphTransformer could be used directly to ingest documents and produce a Neo4j graph . If it works well out-of-the-box, it could save time. Alternatively, we craft custom scripts. We will also use Neo4j’s bulk import or transactional batch inserts to efficiently create the graph once the relationships are identified – this avoids doing it node-by-node slowly.

Example of LLM Extraction: As an illustration, consider Rule 15.2: Movable Obstructions (e.g., removing a rake). The LLM might extract:

- Rule node: “15.2”.
- Mentions “see Rule 15.1 for Loose Impediments” → we create RELATED edge to Rule 15.1.
- Scenario: “movable obstruction” (which is a defined term) → create Scenario node if not exists, link Rule 15.2 APPLIES_TO that scenario.
- Action: no penalty for moving it (so maybe an Action node “No Penalty” or just note that consequence is allowed action).
- If there was an exception like “Exception: Ball in motion deliberately deflected – see Rule 11.2” → then an exception edge linking to Rule 11.2.

We would verify that these captures align with the rule’s intent.

After building the graph, we will have a Neo4j database containing all these nodes and edges. We can visually inspect portions of it in Neo4j Browser to ensure it looks sensible (e.g., pick a rule and see that it connects to the things we expect). This also sets the stage for retrieval usage, where we’ll query this graph.

In summary, constructing the knowledge graph is a feasible task by leveraging automation. Large Language Models can significantly speed up extracting the rule structure , and straightforward references can be caught with simple scripts. The end result will be a robust graph of the rules ready to be integrated with the RAG system.

Graph-Enhanced Retrieval for Improved Context Selection

With the knowledge graph in place, the retrieval process for answering questions will be augmented to use it. Here we detail how graph traversal and subgraph retrieval improve the LLM's context selection and ultimately the answer quality:

1. Identifying Relevant Nodes from a Query:
When a user question comes in, the system needs to determine where in the graph to start. This could happen in a few ways (some of which we will experiment with during development):

- Keyword/Entity Mapping: We maintain a mapping of important keywords to graph nodes. For example, if a question mentions "bunker" we know that relates to "Bunker" (which is covered under "Abnormal Course Conditions" or specifically in definitions). If it says "stroke play" or "match play", those map to scenario nodes for game format. Using either simple keyword matching or a small NLP model, we extract these terms from the query and find corresponding nodes. Many golf rule terms are unique (e.g., "unplayable ball", "out of bounds", "penalty area"), so this is reliable. We can also query Neo4j's full-text index on node names/descriptions to find matching nodes with a query string . Neo4j supports such queries, even with fuzzy matching for misspellings if needed . This means if a user doesn't use the exact terminology, we can still often map it (e.g., user says "water hazard" which the 2019 Rules now call "penalty area" – the full-text search can still find the Penalty Area node).
- Rule Number Detection: If the user question explicitly references a rule number ("According to Rule 12.2, … what about …?"), we will detect that via regex and directly pick the corresponding Rule node. Even if a question doesn't cite a rule, sometimes users include terms like "Rule 12" in their wording, which we shouldn't miss.
- Vector retrieval seeding: Alternatively, run the standard vector retrieval first to see what rule text it brings up, then find the graph node for that rule. For instance, if vector search top result is a chunk from Rule 12.2, we use that as a hint that Rule 12.2 is likely relevant and go to that node in the graph. This approach uses the strength of embeddings to handle tricky phrasing and then uses the graph to expand context around that initial hit.

We may use a combination of these methods for robustness. The outcome of this step is a set of one or more starting nodes in the knowledge graph that are relevant to the query.

2. Retrieving a Subgraph (Graph Traversal):

Once we have the starting point(s), we query Neo4j for the neighborhood around those nodes. Typically, we will fetch nodes that are directly connected (one hop out) via certain edge types, and possibly two hops for specific relationships if needed. The traversal can be constrained by edge type if desired. For example, if we start with a Rule node, we likely want to retrieve:

- Any Rule nodes connected via EXCEPTION or RELATED_TO (one hop). This would bring in exception rules and any directly cross-referenced rules.
- Any Scenario nodes linked (prerequisites/conditions), and any Action nodes linked (consequences).
- Possibly any rules linked to those Scenario nodes (meaning other rules that involve the same scenario) if that seems relevant. However, to avoid going too far afield, we might stick to one hop. (If multiple rules share a scenario, it might not be needed unless the question is broader; this is a tunable aspect.)

The Cypher query might look conceptually like: "MATCH (start)<-[:RELATED_TO|EXCEPTION|PREREQ|…]-(neighbor) WHERE start is {Our Starting Node} RETURN neighbor and relationships". In practice we can use a small Cypher snippet for each starting node to get its immediate neighbors. This neighborhood subgraph represents all information closely tied to the starting concept.

For example, if the query is about "dropping a ball" in a particular situation:

- Starting node: maybe Rule 14.3 – Dropping Ball in Relief Area (just as an example).
- Graph traversal finds that Rule 14.3 is related to Rule 14.4 – When Ball Dropped in Wrong Way (say via RELATED), or to Action: Redrop Required, etc., and any scenarios like "relief area".
- So it would return Rule 14.4, the action node about redropping, etc. These would be added to context so the LLM knows about wrong drops as well, covering a likely follow-up scenario in the question.

If the query had multiple distinct aspects (e.g., "In match play, if my opponent moves my ball, what is the penalty?"), we might have two starting nodes: one for "match play" (Scenario) and one for "ball moved by opponent" (which might map to Rule 9.5, say). The graph could then gather context from the match play node (which might link to some rule differences) and from Rule 9.5's neighbors (like penalties or exceptions if any).

3. Merging Graph Context with Document Context:
The data retrieved from the graph is structural – essentially pointers to relevant rules and concepts. We then need the content of those to feed to the LLM. For each relevant Rule node identified through the graph, we will fetch the actual text of that rule (from our data store or we could have stored it in the graph as a property). For scenario or action nodes, which might not have a full text, we will prepare a short description. For instance, the "Stroke-and-Distance" action node could carry a note like "Stroke-and-Distance: player takes a one-stroke penalty and replays from previous spot." We can store that description as a property when building the graph so it's readily available.

Definitions or scenarios might have their definition text stored as well. These descriptions ensure that even if a node is not a rule with formal text, the LLM still gets an explanation of it.

We will compile a context that includes:

- The most relevant rule text (likely from vector retrieval).
- Additional rule texts brought by the graph traversal.
- Any necessary explanatory notes for scenarios/actions (we might prepend something like "[Definition] A 'Loose Impediment' is a natural loose object (stones, leaves, etc.) that can be removed." if that node was fetched).

This compilation will be carefully done to avoid overloading the LLM context window with redundant info. If the vector retriever and graph retriever end up bringing the same rule, we'll de-duplicate. Our system can prioritize or rank context if needed – e.g., ensure the rule most directly asked about is first. We might also chunk longer rule texts if they are too long, but since we already have an embedding chunking, likely each piece is manageable (and in a real query, only certain parts of a long rule might be needed).

The combined context is then passed to the LLM. Essentially, the graph ensures that no critical piece of related rule is missing in that context. This addresses the earlier issues: an LLM given all relevant pieces (primary rule + exceptions + consequences) is far more likely to answer correctly and completely.

4. Benefits During Answer Generation:
When the LLM generates the answer, the presence of the structured context means:

- The answer can explicitly reference multiple rules accurately (since the text is provided). For example, "Under Rule 16.1 you generally get relief, except Rule 16.2 allows it even in a dangerous situation." The LLM can only do that if it has both rules in context, which our method ensures when needed.
- The reasoning can follow the graph's logic. If the question implicitly requires a two-step reasoning (e.g., identify the ball is lost, then apply the lost ball rule's penalty), the context will already lay that path out (scenario node "ball lost" connected to Rule 18.2 which has the penalty). The LLM doesn't have to make a leap; it sees that connection explicitly.
- Ambiguity in the query can be resolved by graph context. For instance, if a question is phrased vaguely, the vector search might retrieve one interpretation. The knowledge graph might surface another related rule. The LLM can then disambiguate by possibly asking (if interactive) or by covering both in the answer. At least it prevents the answer from focusing on the wrong aspect without acknowledging the other. (Though our system isn't dialog, in testing if we see ambiguous queries, we can ensure the graph fetches both possibilities.)

5. Example Walk-Through:
Consider a concrete example to show the improvement:

- User question: “My ball was accidentally moved by an outside influence. What do I do and is there a penalty?”
- Traditional RAG behavior: It might retrieve a chunk from Rule 9.6 (Ball Lifted or Moved by Outside Influence) which says the ball must be replaced and there’s no penalty. The LLM would answer from that. That’s fine, but what if the outside influence was during search (Rule 7) or what if it was deliberately moved (Rule 9.6 has exceptions)? The plain RAG might not bring those nuances.
- Hybrid approach: We identify “moved by outside influence” as relating to Rule 9.6. Graph retrieval gets Rule 9.6 node. The graph shows an Action: “Replace ball” and that there’s an exception edge to perhaps Rule 9.5 (if the outside influence was a player’s caddie in some cases) – this is just hypothetical. It also pulls the Scenario: “Outside Influence” (which might link to a Definition node clarifying what counts as outside influence). Now the LLM sees Rule 9.6 text (no penalty, replace), and possibly a note “Outside Influence: e.g., an animal or spectator”. The LLM can confidently answer: “Under Rule 9.6, if an outside influence (like a spectator or animal) moves your ball, you must replace the ball to its original spot. There is no penalty to you. (If the ball wasn’t found you would drop a replacement as near as possible.)” – citing Rule 9.6. This answer covers the rule and penalty clearly. If there was a caveat (like if the outside agency was deliberately influencing play, which might invoke another rule), our graph would have provided that too. Essentially, we reduce the risk of missing edge cases.

6. Implementation of the Graph Query:
We will implement the graph traversal likely in the application layer by calling Neo4j with Cypher. For example, a function get_graph_context(query_entities) could:
- For each identified relevant entity or rule: run a Cypher MATCH that finds its neighbors (we can parameterize the node id or name). We might collect neighbors of type Rule, Scenario, Action separately or just together.
- Return a set of node identifiers (or the node data including any text) to include in context.

We might also assign a priority or filter – e.g., if a rule has 10 related nodes, maybe not all are relevant. But given the rules, typically each rule references only a handful of others. We can include them all if in doubt, as long as the context size permits (most rules are short, and an LLM like GPT-4 can handle quite a bit of text). If context size becomes an issue, we can prioritize by relationship type (exceptions and prerequisites are more crucial than loosely related references, for instance).

Neo4j query performance for these localized graph searches will be very fast (millisecond-level) since we are dealing with a modest-sized graph (a few hundred nodes at most for all rules, definitions, actions, and a few thousand relationships perhaps). This won’t be a bottleneck. The LLM call remains the dominant time cost, so we expect negligible impact on overall latency – maybe adding 50-100ms for graph lookup, which is trivial compared to a ~2 second LLM generation.

In code, after retrieval, we simply append the texts and proceed to the prompt assembly. If using a framework like LangChain, this hybrid retriever can be encapsulated; in fact, similar patterns have been published (combining vector and graph retrievers) , which we can draw from.

7. Continuous Learning from Queries:
Though not explicitly asked, it's worth noting: as we get user queries, we might see patterns where certain graph connections are missing or need tuning. Because the knowledge graph is easily inspectable, we can update it (e.g., add a missing edge) and immediately the system will use that next time. This is a nice advantage over purely embedding-based knowledge, which is harder to tweak for specific logical gaps.

In summary, graph traversal allows the system to retrieve a coherent subgraph of knowledge relevant to the question, and using that as additional context ensures the LLM has all the pieces to generate a correct and thorough answer. This method leverages the explicit structure we've encoded: rather than hoping the vector similarity will surface every needed part, we systematically pull them via relationships. The result will be answers that reflect the full breadth of the Rules of Golf for a given situation, enhancing user trust and the system's reliability.

Implementation Plan and Phased Rollout

Implementing this hybrid RAG+KG solution will be tackled in phases to manage risk and demonstrate value quickly. Below we outline the phases, including a rapid prototype and subsequent full-scale development, along with timeline estimates and deliverables for each.

Phase 1: Feasibility Prototype (Approx. 3 Days)

Objective: Validate the concept on a small scale – show that a knowledge graph can be built and used to improve retrieval for the golf rules Q&A.

Scope: In this phase, we will focus on a limited subset of the rulebook – enough to cover a few interesting scenarios. For example, we might take Rule 16 (Relief rules) and Rule 18 (Lost ball/OOB) as they involve exceptions and penalties, which are ideal to test the graph approach. Alternatively, we might choose a handful of disparate rules to see broad coverage. The prototype will not cover all rules, just enough to test end-to-end.

Tasks:
- Set up Neo4j: Install and run a local Neo4j instance. Load a small dataset of nodes and relationships for the chosen subset of rules. This can be done manually or via a quick script. For the prototype, manual entry or very targeted extraction is fine since it's limited data. For instance, create nodes for Rule 16.1, 16.2, 18.2, and their key related nodes (like scenarios "Abnormal Condition", "Dangerous Animal", action "One-Stroke Penalty"). This could be on the order of 10–20 nodes and 15–30 edges – manageable in a short time.

- Integrate Neo4j access: Write a small module in the application to query Neo4j. Test a simple Cypher query from the code to ensure connectivity (e.g., retrieve a known node by name).
- Implement basic graph retrieval logic: For the prototype, we might hardcode the logic for a couple of example queries. For instance, if testing a "dangerous animal relief" question, the code can directly query the neighbors of "Rule 16.2" to simulate the dynamic process. Alternatively, implement a simplified entity extraction (maybe just regex or a lookup table for known terms to node IDs) to convert a user query to a graph query. The emphasis is on demonstrating that we can get the right related info.
- Run a few example Q&A through hybrid retrieval: Use 2-3 sample questions that we know are challenging for the current system. For each, do: vector retrieval as usual, graph retrieval (with our small graph), combine, and feed to the LLM (likely GPT-4 or GPT-3.5 for testing). Examine the answers. We will compare answers with and without the graph component to see the difference. The expectation is that with the graph, the answer includes the relevant exception or related rule that was missing before.
- Evaluate feasibility: During this, we will note how easy/difficult it was to map a question to graph nodes, how much improvement we saw in answers, and any integration hurdles (e.g., combining context effectively, any unexpected noise from the graph). Also, measure roughly the latency overhead (though trivial in a small test).
- Deliverable: A short demo (could be a notebook or script output) showing a question asked, and the answer generated with the augmented context, ideally highlighting that now it includes multiple rules or correct reasoning where previously it might not have. We will also have an initial graph schema confirmation – seeing it work in practice with a subset will validate the schema design or indicate adjustments.

By the end of Phase 1 (within ~3 days), we aim to have convincing evidence for stakeholders that "Yes, the knowledge graph integration can work and adds value." This de-risks the project by tackling unknowns early (like the mechanics of calling Neo4j and merging data). It's essentially a proof-of-concept that we can then scale up.

Phase 2: Full Knowledge Graph Build-Out (Estimated ~2-3 weeks)

Objective: Construct the knowledge graph for the entire Rules of Golf and integrate it fully into the RAG system, ready for internal testing.

Tasks:

- Automate Graph Construction: Using the techniques discussed (LLM parsing, pattern extraction), build the complete graph. We will likely proceed section by section through the rulebook. This involves writing the extraction scripts and running them, then pushing results to Neo4j. We'll keep this process iterative: for example, process all Rules in Chapter 1-10, review, then 11-20, etc. Any anomalies the LLM outputs (like a mistaken relationship) we correct either in code or manually adjust in the graph. Since consistency is important, we may also do spot checks with an expert or at least cross-reference the official interpretations to ensure we didn't mis-link anything critical.

- Ingest Q&A Linkages (optional): The email Q&A data is unstructured, but if time permits, we could link some of those into the graph. For example, if a particular Q&A corresponds to Rule 5.3, we could have an edge from Rule 5.3 node to a "Q&A" node or at least tag it. This is optional and can be added later; the primary focus is the rulebook itself. The Q&As will still be handled by vector search primarily. However, even without explicit graph linking, the benefit might be mutual: the graph can improve Q&A retrieval by guiding to the right rule, and the Q&As can enhance answers but referencing the rule text we ensure via graph.
- Refine Retrieval Implementation: Generalize the prototype retrieval code to handle any user query. This means implementing the entity extraction systematically (we can use a lightweight approach like a dictionary of key terms to nodes, or even an LLM prompt "Which rule or key terms does this question relate to?"). We should also handle multiple possible relevant nodes. We'll implement a function that given a question returns a set of node IDs (or types) to retrieve. We will also implement the merging logic thoroughly, ensuring no prompt formatting issues. Possibly, we'll decide on a consistent prompt style (e.g., always start the context with a list of relevant rules and their text). This is also where we add any needed prompt engineering to ensure the LLM uses the provided context well (for instance, instruct it to ground answers in the rules given).
- Testing and Iteration: We will test the full system on a wide range of questions – including those from the email dataset (since we have a gold standard of expected answers there) and novel questions. This will help us identify if the knowledge graph is missing some connections. For example, if the answer comes out slightly wrong or incomplete, we check if perhaps a needed rule was not retrieved – if so, is it because the graph lacked an edge or the query mapping missed a node? We then fix accordingly. We also ensure that if a question is straightforward (only one rule needed), the system doesn't over-complicate – it should still work, perhaps the graph just finds nothing extra and we proceed normally, which is fine.
- Performance Considerations: During this phase, we will monitor how much the graph retrieval adds to response time and memory. As noted, it should be minor, but if any query is slow (maybe a complex Cypher on a large graph), we can add indexes or revise the query. We will also ensure the combined context stays within model limits (it should; if not, we might limit to top N neighbors). This phase might involve some optimization such as caching the embedding of rule texts in Neo4j (Neo4j allows vector search, but if we already have a vector store it might not be needed to duplicate – however, we could consider using Neo4j's hybrid index to simplify infrastructure). These decisions will be made based on what's most efficient for our setup.
- Deliverable: By the end of Phase 2, we expect to have an integrated hybrid QA system covering the entire rulebook. We'll prepare a demonstration for stakeholders showing multiple example queries, comparing the answers before and after KG integration. For instance, a question that previously omitted an exception now correctly includes it, citing both rules. We will also deliver documentation of the knowledge graph (schema, any scripts used to create it) and the updated system

architecture. This phase should result in a working solution that could be tested by domain experts or even a small group of end users (maybe some golf experts) to gather feedback.

Timeline-wise, building the full graph and integrating could be done in a couple of weeks, but we'll adjust as needed. The key is that after a few iterations of testing, we should reach a point where the system answers most questions at a higher accuracy level than before. (We might quantify this with the Q&A set: e.g., originally X% correct, now improved by some margin – if accessible, though quality is often evident qualitatively given this domain).

Phase 3: Preparation for Public Deployment and Maintenance

Objective: Hardening the system for production use, ensuring maintainability and versioning for the long term.

Once the solution proves itself internally, we'll focus on scaling and maintaining it for a public (or wider) deployment. This involves:

- Optimization and Infrastructure: Ensure that the Neo4j instance is production-ready. For public deployment, we may use a managed Neo4j Aura or host it on a robust server/cluster for reliability. We'll implement monitoring on the Neo4j queries (to catch any slow queries or to watch for unusual patterns). We'll also make sure the LLM inference is scalable (if using an API, handle rate limits; if using a local model, ensure the machine can handle expected load). This phase might include load testing with concurrent requests to see how the system performs under traffic, and adding caching layers if needed (e.g., caching the results of certain graph queries that are very common, or even caching full answers to frequently asked questions – although with thousands of possible questions, the graph approach might be sufficient).
- Versioning of Rules: A key advantage of structuring knowledge is easier updates. The Rules of Golf are updated periodically (e.g., significant changes in 2019, minor updates in 2023, etc.). To handle future versions, we design a strategy in the graph: one approach is to add a property "edition/year" to each rule node and scenario. For example, all current nodes get a tag "2019-2023" rules. If a new 2025 rulebook comes, we can update or add nodes with tag "2025". If rules are mostly unchanged, we update properties; if some are new or renumbered, we add new nodes. This way, the graph can either represent multiple versions side by side or be switched out by version. For initial deployment, we likely stick to the current rule set, but we note this plan. We can also maintain the old version if needed for historical queries, but likely users will care about the latest rules. In any case, we will document how to update the graph when rules change: e.g., "if a Rule is modified, update its node's text and adjust relationships if needed; if a rule is added or removed, add or deactivate the node and re-link edges accordingly; update any definitions changes, etc." Since our graph is all in Neo4j, updating it is straightforward with Cypher queries or by re-running an LLM parse on the updated text and diffing the output. We'll likely script the update process for efficiency.

- Maintainability and Knowledge Updates: Beyond rule changes, the system might need to incorporate new knowledge like additional clarifications or ongoing Q&As. We plan to maintain a procedure for adding new Q&A pairs to the vector index (which likely already exists). With the graph, if a particular Q&A reveals a new needed link (e.g., a rare scenario connecting rules we hadn't linked), we can add that. We might also consider allowing domain experts to suggest graph edits via a simple interface or even via entering data (though not necessarily in initial deployment). Because the graph is a transparent representation, even new team members or rule officials could inspect it to verify correctness. We will provide clear documentation for the graph data model and a cheat-sheet for how to modify it.
- Testing and Validation for Public Launch: We'll do another round of rigorous testing with a larger set of example queries (possibly using the thousands of emails as test cases – run them through and see if the answers align with the known answers). Any deviations are analyzed: if the system is wrong, is it a graph issue or LLM issue? If graph, fix data; if LLM, perhaps adjust the prompt or ensure the context contains what's needed. We want to approach the 95%+ accuracy that top human rules officials have, as was mentioned by Play Today's metrics (they achieved ~88% and aiming for 95%+) . While that is an ambitious benchmark, our system's strength will come from solid data coverage (graph + docs) and using a state-of-the-art LLM.
- User Experience and Explanation: With public deployment, consider how to present the answers. Perhaps we want to show the user not just the answer but the source rules used (for transparency). The knowledge graph can aid in that by essentially providing a mini "explanation graph" for each answer. We can output something like: "Answer based on Rule 16.1 and Rule 16.2." This builds trust. In the background, the graph is ensuring those are the correct rules. We may not need to build a UI for the graph for users, but internally it helps ensure we can explain why a certain answer was given (which is good for debugging or user follow-up). If future requirements include an explainable AI aspect, the graph is a great asset – it provides a natural way to trace which facts led to the conclusion.
- Future Extensions: Although not immediate, this hybrid approach sets us up to incorporate even more advanced features down the line, such as logical inference or simulation. For example, one could imagine adding a layer where an LLM or a rule engine uses the graph to do step-by-step reasoning (like a chain-of-thought explicitly following graph edges). Our current plan doesn't require that complexity (the LLM handles it implicitly), but having the graph structured means we could experiment with such features (perhaps for very complex queries or to double-check the LLM's answer by traversing the graph as validation). We mention this just to highlight that investing in the knowledge graph opens doors for future innovation beyond what a pure RAG approach could do.
- Timeline & Resources: Phase 3 might overlap with Phase 2 towards the end, but essentially once the system works, we allocate time (maybe 1-2 weeks) for these deployment preparations and checks. If a public release is intended soon after Phase 2, we make sure to schedule a buffer for any fine-tuning discovered in wider testing.
- Deliverable: A production-ready hybrid system, documented and maintainable. We will provide a final report or presentation to stakeholders summarizing

the improvements achieved, examples of success cases, and instructions for maintenance. Also, a plan for monitoring post-deployment (how to catch any errors, how to update content) will be delivered.

Value Proposition and Conclusion

In conclusion, this proposal outlines a clear path to integrate a Neo4j knowledge graph with the existing RAG architecture to create a more powerful Golf Rules question-answering system. The value of this hybrid approach can be summarized as follows:

- Higher Accuracy and Completeness: By marrying unstructured text retrieval with structured knowledge, the system will more reliably fetch all relevant information for a query. This directly translates to answers that are more likely to be correct and complete. Ambiguous or complex queries will be handled with the aid of the graph's guidance, reducing the chance of erroneous answers. In a domain where precision is critical (rules advice must be correct), this is a significant win. Studies and industry use cases confirm that such KG-enhanced RAG systems improve factual accuracy and reasoning reliability .
- Explicit Handling of Rule Interdependencies: The knowledge graph encodes the rich web of rule interdependencies. The system will no longer treat the rulebook as disconnected paragraphs, but as a connected knowledge base. This means a question that involves multiple rules (which is common in golf rulings) will automatically lead to retrieving those multiple rules. The answer, therefore, will reflect the appropriate interplay (e.g., citing an exception rule alongside the general rule). This was a pain point in the current approach that the knowledge graph squarely addresses.
- Retention of Existing Workflow: We emphasize that the current RAG pipeline (embeddings, vector search, LLM prompting) remains intact. We are not discarding or replacing the investment made in that system. Instead, we layer on the knowledge graph to augment it. This ensures we avoid regressions – all queries that worked well before will still work (the graph will simply add more context where needed, or do nothing if not needed). It's an additive improvement, not a risky replacement. The LLM still ultimately answers using provided text, so we remain grounded in the rulebook's authoritative wording.
- Feasibility and Low Risk: The phased approach ensures feasibility. A prototype in a few days will prove out the concept on a small scale, giving stakeholders confidence and allowing early input. The full implementation leverages automation (LLMs and scripts) to build the graph without requiring an army of people manually encoding rules. Neo4j is a mature technology that can be run easily and integrated with Python/JavaScript, so technical risk is low. Many team members can likely ramp up on Cypher queries quickly since it's fairly intuitive (similar to SQL but for graphs). Additionally, by aligning with known patterns (like LangChain's GraphRetriever concept), we are using well-trodden paths, not inventing from scratch.
- Maintainability and Future-Proofing: The introduction of a structured knowledge layer makes the system easier to update and expand. New rules or modifications can be done by updating the graph without retraining models or re-embedding everything (though re-embedding a few new texts is trivial anyway). The knowledge graph also makes the system more interpretable – one can inspect the

relationships and understand the domain better, which is useful for onboarding new engineers or rule experts into the project. From a deployment perspective, having explicit rules encoded can help ensure consistency (we could even validate that no contradictory rules are being applied, etc., if needed). For future public usage, this structured approach means the system can more easily meet high accuracy standards and even provide explanations of its answers by referencing the graph it used.

- Competitive and Innovative Edge: As seen with the Play Today Golf AI example, combining AI with a knowledge graph of rules is at the cutting edge of golf tech. By implementing this, the client's solution not only solves the current retrieval issues but also positions itself as an innovative product. It leverages the latest AI techniques (LLMs) in a responsible way by grounding them in a curated knowledge graph. This could be highlighted as a differentiator if the solution is offered to end users or clients (e.g., "our AI not only reads the rules, but truly understands their connections, just like a human expert").

The case for this hybrid approach is strong: it is based on evidence from other domains (like legal AI) where combining KGs with RAG improved performance , and it directly addresses the pain points observed in our current system. Technically, it is achievable with the tools and expertise at hand, and the phased plan ensures a clear, testable path to success.

We recommend proceeding with Phase 1 immediately, given the short turnaround for a prototype, to illustrate these benefits in action. Assuming the prototype confirms our expectations, we can then commit to the full build-out and integration. The end result will be a robust Golf Rules Q&A assistant that provides precise, context-rich answers, meeting the high standards required in interpreting the rules of golf.

By supplementing the existing RAG system with a Neo4j knowledge graph, we are effectively empowering our AI with a form of "understanding" of the rulebook's structure – leading to a smarter and more reliable assistant for all golf rulings inquiries.