Advance Natural Language Processing

MSDS

Submitted by

Shahroz Naveed	25F-8003
Muhammad Umar	25F-8007

Task 01: This project focuses on building a fully functional Retrieval Augmented Generation (RAG) system using LangChain and the free Gemini API. Students will design a dataset pipeline, embed documents, build retrievers, and create an interactive QA/chat interface. The project also requires deployment, evaluation, and documentation through a GitHub repository and a Medium-style article.

Evaluation Report:

Reasoning: The previous error was due to an invalid f-string format specifier when conditionally displaying the average similarity or 'N/A'. I will fix this by separating the conditional logic for the string 'N/A' from the float formatting for the average similarity, ensuring the format specifier is only applied to a float.

Generate Detailed Evaluation Report

Subtask:

Compile a comprehensive evaluation report in markdown format, including overall summary, specific performance metrics, detailed example query results, discussion of strengths and limitations, and recommendations for future improvements. The report will reflect a deep analysis of the evaluation results and be generated as a single markdown string.

Reasoning: The report_content string containing the markdown evaluation report has been generated. The next logical step is to print this content to the console as requested in the instructions**.**

Final Task

Subtask:

Confirm the successful generation and display of the detailed evaluation report, reflecting a thorough analysis of the RAG system's performance.

Summary:

Data Analysis Key Findings

Evaluation Scope: The Medical RAG System was evaluated on 30 medical queries covering 6 medical specialties (Allergy, Cardiology, Neurology, Gastroenterology, Respiratory, Endocrinology).
Source Retrieval: The system found relevant sources for 20 out of 30 queries.
Average Sources: On average, 1.30 sources were retrieved per query.
Context Utilization: The system successfully utilized retrieved context for 15 out of 30 queries, meaning it did not return a "cannot find" response.
Answer Length: The average answer length generated by the system was 266 characters.
System Strengths: The RAG system demonstrated medical accuracy, transparent source citation (with specialty and similarity scores), effective context utilization, and a safe approach by explicitly stating when information was unavailable.
System Limitations: Identified limitations include dependency on the scope and recency of the clinical transcription dataset, potential lack of broad medical knowledge compared to general-purpose AIs, and reliance on the quality of the embedding model.
Example Query Reruns: While example queries were selected and "rerun" for the report, the underlying RAG system failed to initialize properly due to missing FAISS index and metadata files. Consequently, the detailed example results in the report showed "Dummy answer for..." and "N/A" for source details, indicating a functional issue with the live RAG system component during report generation for these specific examples, even though the overall metrics were derived from the evaluation_df.
Insights or Next Steps
Address RAG System Initialization: The most critical next step is to ensure the Medical RAG System components (FAISS index, metadata, embedding model) are correctly loaded and accessible. This will allow the detailed example query results in the report to reflect actual system performance rather than dummy outputs.
Implement Recommended Improvements: Focus on incorporating recommendations such as data augmentation with more diverse and current

medical literature, exploring advanced retrieval algorithms, and developing confidence scoring mechanisms to enhance the system's robustness and utility.

TASK 02

Task

Generate a comprehensive evaluation report for the Policy Compliance Checker RAG System (Task 02). This report should:

1. Summarize the system's key architectural components, including compliance rules, text processing, vector store, custom LangChain tools, and the multi-step agent workflow.
2. Evaluate the functionality of each component based on the execution results, noting successes like chunking, embedding, retrieval, and agent logic, as well as limitations like Gemini API rate limiting.
3. Highlight successful implementations such as custom LangChain tools and intelligent rule extraction within the agent.
4. Detail the primary limitation (Gemini API rate limiting) and suggest solutions or future enhancements.
5. Synthesize all information into a structured report presenting the project's status, achievements, and areas for improvement.

Summarize System Components

Subtask:

Outline the key architectural components of the Policy Compliance Checker RAG System, including compliance rules, text processing, vector store, custom LangChain tools, and the multi-step agent workflow.

Summarize System Components

Architectural Components of the Policy Compliance Checker RAG System:

1. Compliance Rules:

Role: These are predefined sets of legal requirements against which contracts are evaluated. They act as the core criteria for compliance checking.

Details: Based on categories derived from the CUAD dataset, these rules define specific aspects of a contract that need to be present, adhere to certain conditions, or be absent. Each rule has a name, description, compliance requirement, severity, and suggested remediation.

2. Text Processing:

Role: This component is responsible for preparing raw contract text for efficient retrieval and analysis.

Details: It involves two main steps:

Chunking Contracts: Large contract documents are broken down into smaller, manageable text chunks (e.g., using `RecursiveCharacterTextSplitter`) with specified overlap to maintain context.

Extracting Legal Sections: Identifiable legal sections (like 'Governing Law', 'Termination', 'Confidentiality') are specifically extracted, which often provides more granular and contextually rich units for analysis.

3. Vector Store:

Role: Stores the numerical representations (embeddings) of the processed text chunks and legal sections, enabling rapid semantic search.

Details: It utilizes `SentenceTransformer` models (e.g., "all-MiniLM-L6-v2") to convert text into high-dimensional vectors. `FAISS` (Facebook AI Similarity Search) is then used to create an efficient index from these embeddings, allowing for fast retrieval of semantically similar text segments based on a given query.

4. Custom LangChain Tools:

Role: These are specialized functions wrapped as LangChain tools, allowing a conversational agent to interact with the core functionalities of the system.

Details: The system includes:

`CheckSingleRuleComplianceTool`: Evaluates a contract against a single specified compliance rule.

`CheckMultipleRulesComplianceTool`: Checks compliance against a list of rules and provides a summary.

`RetrieveContractSectionsTool`: Queries the vector store to fetch relevant contract sections based on semantic similarity.

`CompareContractComplianceTool`: Compares the compliance of two different contracts for a specific rule.

5. Multi-Step Agent Workflow:

Role: Orchestrates the interaction between the user, the compliance tools, and the underlying RAG components to provide intelligent, multi-step responses.

Details: The agent processes user queries, dynamically analyzes the query type (e.g., compliance check, retrieval, comparison, comprehensive analysis), and then selects and executes the appropriate `Custom LangChain Tools`. It integrates retrieval-augmented generation (RAG) by first fetching relevant information from the vector store or using compliance checks, and then generating a coherent and informative response based on the findings, often involving multi-step reasoning.

Evaluate Individual Component Functionality

Subtask:

Assess the functionality of each component based on the execution results. This includes the successful creation of chunks and embeddings, the working retrieval mechanism, and the agent's logic for query processing. Also, note the current status of the Gemini API integration (e.g., rate limiting).

Subtask: Assess the functionality of each component

Based on the execution results, here's an evaluation of each component's functionality:

1. Creating Text Chunks for RAG System (Cell `heM5SfKIRraE`):
   Status: Successful.
   Details: The `ContractTextProcessor` successfully created 46 chunks from 1 contract. The chunk length statistics are: Min 117 chars, Max 783 chars, Avg 596 chars. It also extracted 6 legal sections. This indicates the text splitting and section extraction mechanisms are working as intended.

2. Creating Embeddings and Vector Store (Cell `Os0pP8JDRrN0`):
   Status: Successful.
   Details: The `ComplianceVectorStore` successfully loaded the `all-MiniLM-L6-v2` embedding model and created 52 embeddings (combined chunks and legal sections) with a dimension of 384. The FAISS index was built and tested for retrieval. The retrieval tests for 'governing law jurisdiction' and 'termination clause' returned relevant-looking snippets with similarity scores, demonstrating the vector store and retrieval mechanism are functioning correctly.

3. Configuring Gemini API and Creating Compliance Tools (Cell `fNRC1debTJnt`):
   Status: Failed due to API error.
   Details: The Gemini API configuration failed with a `429 POST` error, indicating rate limiting or quota issues (`You exceeded your current quota`). Consequently, the `compliance_checker` object, while initialized, cannot perform actual LLM calls and will likely return errors or fall back to mock responses. This is a critical issue impacting all LLM-dependent components.

4. Creating LangChain Tools for Compliance Checking (Cells `FEvT2_lUTJkW` and `_CvanSIzTJij`):

Status: Partially functional, but LLM-dependent tools are failing.

Details:

The initial test in `FEvT2_lUTJkW` for `compliance_tool.run()` failed with a `KeyError: 'risk_level'`. This error likely occurred because the `_parse_compliance_response` method in `ComplianceCheckerTools` (Cell `fNRC1debTJnt`) received an unexpected or incomplete response from the Gemini API due to the rate limit error, leading to a missing 'risk_level' key.

In `_CvanSIzTJij`, the refactored `CheckSingleRuleComplianceTool`, `CheckMultipleRulesComplianceTool`, and `CompareContractComplianceTool` all produced `Status: ERROR` and `Score: 0/10` in their test runs. This directly confirms the impact of the Gemini API issues on any tool relying on the `compliance_checker`'s LLM calls.

5. `retrieve_contract_sections` tool (Cell `_CvanSIzTJij`):

Status: Successful.

Details: The `retrieve_contract_sections` tool, as demonstrated by the test query "What is the governing law of the contract?", successfully returned relevant sections from the `CUAD_v1_README.txt` with similarity scores. This tool relies on the previously created vector store and embeddings, not the Gemini LLM, confirming its independent functionality.

6. Creating Agent Workflow for Multi-Step Compliance Questioning (Cell `py4RaWPcTJgZ`):

Status: Agent logic is sound, but execution for LLM-dependent tasks fails.

Details:

The agent correctly identifies query types, e.g., "Check if the contract has proper governing law provisions" as `compliance_check` and "Find termination clauses in the contracts" as `retrieval`.

For the `compliance_check` query, the agent's response shows `Status: ERROR` and `Score: 0/10` for the 'Governing Law Clause', consistent with the underlying Gemini API failure.

For the `retrieval` query, the agent successfully executes the `retrieve_contract_sections` tool and returns relevant text, indicating that the agent's logic for using non-LLM tools works.

7. `generate_compliance_comparison` and `PolicyComplianceRAGSystem` (Cells `qsIYjbMxRdsU` and `L0Lb3edIRdj1`):

Status: Functionality is severely impacted by the Gemini API issues.

Details:

Both the `generate_compliance_comparison` function and the `PolicyComplianceRAGSystem`'s `analyze_contract` and `generate_compliance_report` methods show `Status: ERROR` and `Score: 0/10` across all rules when attempting to analyze `CUAD_v1_README.txt`. This is a direct consequence of the `compliance_checker.check_rule_compliance` calls failing due to the Gemini API rate limit.

The system correctly structures the output (tables, summary, report format), but the analytical content is compromised by the upstream LLM failures.

Conclusion: The core components for text processing (chunking, section extraction), embedding generation, and vector-based retrieval are functioning correctly. However, all components that rely on the Gemini LLM for compliance analysis (i.e., `ComplianceCheckerTools`, and subsequently the LangChain tools, the agent for compliance checks, and the overall RAG system's analytical functions) are currently failing due to Gemini API rate limiting/quota issues.

Subtask: Assess the functionality of each component

Based on the execution results, here's an evaluation of each component's functionality:

1. Creating Text Chunks for RAG System (Cell `heM5SfKIRraE`):
   Status: Successful.
   Details: The `ContractTextProcessor` successfully created 46 chunks from 1 contract. The chunk length statistics are: Min 117 chars, Max 783 chars, Avg 596 chars. It also extracted 6 legal sections. This indicates the text splitting and section extraction mechanisms are working as intended.

2. Creating Embeddings and Vector Store (Cell `Os0pP8JDRrN0`):
   Status: Successful.
   Details: The `ComplianceVectorStore` successfully loaded the `all-MiniLM-L6-v2` embedding model and created 52 embeddings (combined chunks and legal sections) with a dimension of 384. The FAISS index was built and tested for retrieval. The retrieval tests for 'governing law jurisdiction' and 'termination clause' returned relevant-looking snippets with similarity scores, demonstrating the vector store and retrieval mechanism are functioning correctly.

3. Configuring Gemini API and Creating Compliance Tools (Cell `fNRC1debTJnt`):
   Status: Failed due to API error.
   Details: The Gemini API configuration failed with a `429 POST` error, indicating rate limiting or quota issues (`You exceeded your current quota`). Consequently, the `compliance_checker` object, while initialized, cannot perform actual LLM calls and will likely return errors or fall back to mock responses. This is a critical issue impacting all LLM-dependent components.

4. Creating LangChain Tools for Compliance Checking (Cells `FEvT2_lUTJkW` and `_CvanSIzTJij`):
   Status: Partially functional, but LLM-dependent tools are failing.

Details:

The initial test in `FEvT2_lUTJkW` for `compliance_tool.run()` failed with a `KeyError: 'risk_level'`. This error likely occurred because the `_parse_compliance_response` method in `ComplianceCheckerTools` (Cell `fNRC1debTJnt`) received an unexpected or incomplete response from the Gemini API due to the rate limit error, leading to a missing 'risk_level' key.

In `_CvanSIzTJij`, the refactored `CheckSingleRuleComplianceTool`, `CheckMultipleRulesComplianceTool`, and `CompareContractComplianceTool` all produced `Status: ERROR` and `Score: 0/10` in their test runs. This directly confirms the impact of the Gemini API issues on any tool relying on the `compliance_checker`'s LLM calls.

5. `retrieve_contract_sections` tool (Cell `_CvanSIzTJij`):

Status: Successful.

Details: The `retrieve_contract_sections` tool, as demonstrated by the test query "What is the governing law of the contract?", successfully returned relevant sections from the `CUAD_v1_README.txt` with similarity scores. This tool relies on the previously created vector store and embeddings, not the Gemini LLM, confirming its independent functionality.

6. Creating Agent Workflow for Multi-Step Compliance Questioning (Cell `py4RaWPcTJgZ`):

Status: Agent logic is sound, but execution for LLM-dependent tasks fails.

Details:

The agent correctly identifies query types, e.g., "Check if the contract has proper governing law provisions" as `compliance_check` and "Find termination clauses in the contracts" as `retrieval`.

For the `compliance_check` query, the agent's response shows `Status: ERROR` and `Score: 0/10` for the 'Governing Law Clause', consistent with the underlying Gemini API failure.

For the `retrieval` query, the agent successfully executes the `retrieve_contract_sections` tool and returns relevant text, indicating that the agent's logic for using non-LLM tools works.

7. `generate_compliance_comparison` and `PolicyComplianceRAGSystem` (Cells `qsIYjbMxRdsU` and `L0Lb3edIRdj1`):

Status: Functionality is severely impacted by the Gemini API issues.

Details:

Both the `generate_compliance_comparison` function and the `PolicyComplianceRAGSystem`'s `analyze_contract` and `generate_compliance_report` methods show `Status: ERROR` and `Score: 0/10` across all rules when attempting to analyze `CUAD_v1_README.txt`. This is a direct consequence of the `compliance_checker.check_rule_compliance` calls failing due to the Gemini API rate limit.

The system correctly structures the output (tables, summary, report format), but the analytical content is compromised by the upstream LLM failures.

Conclusion: The core components for text processing (chunking, section extraction), embedding generation, and vector-based retrieval are functioning correctly. However, all components that rely on the Gemini LLM for compliance analysis (i.e., `ComplianceCheckerTools`, and subsequently the LangChain tools, the agent for compliance checks, and the overall RAG system's analytical functions) are currently failing due to Gemini API rate limiting/quota issues.

Highlight Achievements

Subtask:

List the successful implementations, such as the custom LangChain tools, the intelligent rule extraction in the agent, and the overall integration into a complete RAG system capable of generating reports and comparisons.

Highlight Achievements

Subtask:
List the successful implementations, such as the custom LangChain tools, the intelligent rule extraction in the agent, and the overall integration into a complete RAG system capable of generating reports and comparisons.

Achievements and Successful Implementations:

Despite facing API quota issues with the Gemini model, the core architecture and many functional components of the Policy Compliance Checker RAG System were successfully implemented and demonstrated. Here are the key achievements:

1. Effective Text Processing and Chunking:
   The `ContractTextProcessor` successfully created manageable text chunks from the raw contract data, including both standard chunks and specifically extracted legal sections. This demonstrated robust text preparation capabilities essential for RAG systems. (Ref: Cell `heM5SfKIRraE` output: "Created 46 chunks from 1 contracts", "Extracted 6 legal sections").

2. Robust Vector Store and Embeddings:
   The `ComplianceVectorStore` was successfully initialized, creating high-dimensional embeddings for all processed text units using `SentenceTransformer`. A FAISS index was efficiently built and populated, ready for semantic search. (Ref: Cell `Os0pP8JDRrN0` output: "Embeddings

shape: (52, 384)", "FAISS index created with 52 vectors", "Vector store created and tested successfully!").

3. Functional Retrieval Mechanism:

The `retrieve_contract_sections` tool, powered by the FAISS vector store, proved capable of retrieving relevant contract sections based on semantic queries. This validates the core RAG component's ability to fetch contextually similar information. (Ref: Cell `py4RaWPcTJgZ` output: "Retrieved 3 relevant sections for: 'Find termination clauses in the contracts'").

4. Well-Structured Custom LangChain Tools:

Custom LangChain `BaseTool` implementations (`CheckSingleRuleComplianceTool`, `CheckMultipleRulesComplianceTool`, `RetrieveContractSectionsTool`, `CompareContractComplianceTool`) were successfully defined and integrated. The input schemas (`BaseModel`) were correctly configured, enabling clear communication within the agent framework. Although the `compliance_checker` encountered API errors during execution, the design and instantiation of these tools were complete and correctly structured. (Ref: Cell `FEvT2_lUTJkW`, `_CvanSIzTJij` which shows the tool instantiation and test calls).

5. Intelligent Agent Logic for Query Routing:

The `ComplianceAgent` demonstrated intelligent routing capabilities by analyzing user queries and accurately classifying them (e.g., "compliance_check", "retrieval"). This allowed the agent to select and invoke the appropriate custom tool, forming a multi-step reasoning workflow. (Ref: Cell `py4RaWPcTJgZ` output: "Query type: compliance_check", "Query type: retrieval").

6. Comprehensive System Integration and Report Generation:

The `PolicyComplianceRAGSystem` successfully integrated all components – vector store, compliance checker, and agent – into a cohesive

system. It showcased the ability to orchestrate complex tasks, such as generating compliance reports and performing multi-rule analyses, producing structured output and visualizations, even with placeholder data due to LLM limitations. (Ref: Cell `L0Lb3edIRdj1` output: "Policy Compliance Checker RAG System initialized", "Complete RAG system initialized successfully!"). The system successfully generated both a detailed analysis and a comprehensive report structure, including overall status, scores, and recommendations. (Ref: Cell `NPF1YS3CTJbl` output: "POLICY COMPLIANCE CHECKER - ANALYSIS REPORT").

Highlight Achievements

Subtask:

List the successful implementations, such as the custom LangChain tools, the intelligent rule extraction in the agent, and the overall integration into a complete RAG system capable of generating reports and comparisons.

Achievements and Successful Implementations:

Despite facing API quota issues with the Gemini model, the core architecture and many functional components of the Policy Compliance Checker RAG System were successfully implemented and demonstrated. Here are the key achievements:

1. Effective Text Processing and Chunking:

   The `ContractTextProcessor` successfully created manageable text chunks from the raw contract data, including both standard chunks and specifically extracted legal sections. This demonstrated robust text preparation capabilities essential for RAG systems. (Ref: Cell `heM5SfKIRraE` output: "Created 46 chunks from 1 contracts", "Extracted 6 legal sections").

2. Robust Vector Store and Embeddings:

The `ComplianceVectorStore` was successfully initialized, creating high-dimensional embeddings for all processed text units using `SentenceTransformer`. A FAISS index was efficiently built and populated, ready for semantic search. (Ref: Cell `Os0pP8JDRrN0` output: "Embeddings shape: (52, 384)", "FAISS index created with 52 vectors", "Vector store created and tested successfully!").

3. Functional Retrieval Mechanism:

The `retrieve_contract_sections` tool, powered by the FAISS vector store, proved capable of retrieving relevant contract sections based on semantic queries. This validates the core RAG component's ability to fetch contextually similar information. (Ref: Cell `py4RaWPcTJgZ` output: "Retrieved 3 relevant sections for: 'Find termination clauses in the contracts'").

4. Well-Structured Custom LangChain Tools:

Custom LangChain `BaseTool` implementations (`CheckSingleRuleComplianceTool`, `CheckMultipleRulesComplianceTool`, `RetrieveContractSectionsTool`, `CompareContractComplianceTool`) were successfully defined and integrated. The input schemas (`BaseModel`) were correctly configured, enabling clear communication within the agent framework. Although the `compliance_checker` encountered API errors during execution, the design and instantiation of these tools were complete and correctly structured. (Ref: Cell `FEvT2_lUTJkW`, `_CvanSIzTJij` which shows the tool instantiation and test calls).

5. Intelligent Agent Logic for Query Routing:

The `ComplianceAgent` demonstrated intelligent routing capabilities by analyzing user queries and accurately classifying them (e.g., "compliance_check", "retrieval"). This allowed the agent to select and invoke the appropriate custom tool, forming a multi-step reasoning workflow. (Ref:

Cell `py4RaWPcTJgZ` output: "Query type: compliance_check", "Query type: retrieval").

6. Comprehensive System Integration and Report Generation:

The `PolicyComplianceRAGSystem` successfully integrated all components – vector store, compliance checker, and agent – into a cohesive system. It showcased the ability to orchestrate complex tasks, such as generating compliance reports and performing multi-rule analyses, producing structured output and visualizations, even with placeholder data due to LLM limitations. (Ref: Cell `L0Lb3edIRdj1` output: "Policy Compliance Checker RAG System initialized", "Complete RAG system initialized successfully!"). The system successfully generated both a detailed analysis and a comprehensive report structure, including overall status, scores, and recommendations. (Ref: Cell `NPF1YS3CTJbl` output: "POLICY COMPLIANCE CHECKER - ANALYSIS REPORT").

Highlight Achievements

Subtask:
List the successful implementations, such as the custom LangChain tools, the intelligent rule extraction in the agent, and the overall integration into a complete RAG system capable of generating reports and comparisons.

Achievements and Successful Implementations:

Despite facing API quota issues with the Gemini model, the core architecture and many functional components of the Policy Compliance Checker RAG System were successfully implemented and demonstrated. Here are the key achievements:

1. Effective Text Processing and Chunking:

The `ContractTextProcessor` successfully created manageable text chunks from the raw contract data, including both standard chunks and specifically extracted legal sections. This demonstrated robust text preparation capabilities essential for RAG systems. (Ref: Cell `heM5SfKIRraE` output: "Created 46 chunks from 1 contracts", "Extracted 6 legal sections").

2. Robust Vector Store and Embeddings:

The `ComplianceVectorStore` was successfully initialized, creating high-dimensional embeddings for all processed text units using `SentenceTransformer`. A FAISS index was efficiently built and populated, ready for semantic search. (Ref: Cell `Os0pP8JDRrN0` output: "Embeddings shape: (52, 384)", "FAISS index created with 52 vectors", "Vector store created and tested successfully!").

3. Functional Retrieval Mechanism:

The `retrieve_contract_sections` tool, powered by the FAISS vector store, proved capable of retrieving relevant contract sections based on semantic queries. This validates the core RAG component's ability to fetch contextually similar information. (Ref: Cell `py4RaWPcTJgZ` output: "Retrieved 3 relevant sections for: 'Find termination clauses in the contracts'").

4. Well-Structured Custom LangChain Tools:

Custom LangChain `BaseTool` implementations (`CheckSingleRuleComplianceTool`, `CheckMultipleRulesComplianceTool`, `RetrieveContractSectionsTool`, `CompareContractComplianceTool`) were successfully defined and integrated. The input schemas (`BaseModel`) were correctly configured, enabling clear communication within the agent framework. Although the `compliance_checker` encountered API errors during execution, the design and instantiation of these tools were complete and correctly structured. (Ref: Cell `FEvT2_lUTJkW`, `_CvanSIzTJij` which shows the tool instantiation and test calls).

5. Intelligent Agent Logic for Query Routing:

The `ComplianceAgent` demonstrated intelligent routing capabilities by analyzing user queries and accurately classifying them (e.g., "compliance_check", "retrieval"). This allowed the agent to select and invoke the appropriate custom tool, forming a multi-step reasoning workflow. (Ref: Cell `py4RaWPcTJgZ` output: "Query type: compliance_check", "Query type: retrieval").

6. Comprehensive System Integration and Report Generation:

The `PolicyComplianceRAGSystem` successfully integrated all components – vector store, compliance checker, and agent – into a cohesive system. It showcased the ability to orchestrate complex tasks, such as generating compliance reports and performing multi-rule analyses, producing structured output and visualizations, even with placeholder data due to LLM limitations. (Ref: Cell `L0Lb3edIRdj1` output: "Policy Compliance Checker RAG System initialized", "Complete RAG system initialized successfully!"). The system successfully generated both a detailed analysis and a comprehensive report structure, including overall status, scores, and recommendations. (Ref: Cell `NPF1YS3CTJbl` output: "POLICY COMPLIANCE CHECKER - ANALYSIS REPORT").

Highlight Achievements

Subtask:

List the successful implementations, such as the custom LangChain tools, the intelligent rule extraction in the agent, and the overall integration into a complete RAG system capable of generating reports and comparisons.

Achievements and Successful Implementations:

Despite facing API quota issues with the Gemini model, the core architecture and many functional components of the Policy Compliance Checker RAG System were successfully implemented and demonstrated. Here are the key achievements:

1. Effective Text Processing and Chunking:

The `ContractTextProcessor` successfully created manageable text chunks from the raw contract data, including both standard chunks and specifically extracted legal sections. This demonstrated robust text preparation capabilities essential for RAG systems. (Ref: Cell `heM5SfKIRraE` output: "Created 46 chunks from 1 contracts", "Extracted 6 legal sections").

2. Robust Vector Store and Embeddings:

The `ComplianceVectorStore` was successfully initialized, creating high-dimensional embeddings for all processed text units using `SentenceTransformer`. A FAISS index was efficiently built and populated, ready for semantic search. (Ref: Cell `Os0pP8JDRrN0` output: "Embeddings shape: (52, 384)", "FAISS index created with 52 vectors", "Vector store created and tested successfully!").

3. Functional Retrieval Mechanism:

The `retrieve_contract_sections` tool, powered by the FAISS vector store, proved capable of retrieving relevant contract sections based on semantic queries. This validates the core RAG component's ability to fetch contextually similar information. (Ref: Cell `py4RaWPcTJgZ` output: "Retrieved 3 relevant sections for: 'Find termination clauses in the contracts'").

4. Well-Structured Custom LangChain Tools:

Custom LangChain `BaseTool` implementations (`CheckSingleRuleComplianceTool`, `CheckMultipleRulesComplianceTool`, `RetrieveContractSectionsTool`, `CompareContractComplianceTool`) were successfully defined and integrated. The input schemas (`BaseModel`) were

correctly configured, enabling clear communication within the agent framework. Although the `compliance_checker` encountered API errors during execution, the design and instantiation of these tools were complete and correctly structured. (Ref: Cell `FEvT2_lUTJkW`, `_CvanSIzTJij` which shows the tool instantiation and test calls).

5. Intelligent Agent Logic for Query Routing:

The `ComplianceAgent` demonstrated intelligent routing capabilities by analyzing user queries and accurately classifying them (e.g., "compliance_check", "retrieval"). This allowed the agent to select and invoke the appropriate custom tool, forming a multi-step reasoning workflow. (Ref: Cell `py4RaWPcTJgZ` output: "Query type: compliance_check", "Query type: retrieval").

6. Comprehensive System Integration and Report Generation:

The `PolicyComplianceRAGSystem` successfully integrated all components – vector store, compliance checker, and agent – into a cohesive system. It showcased the ability to orchestrate complex tasks, such as generating compliance reports and performing multi-rule analyses, producing structured output and visualizations, even with placeholder data due to LLM limitations. (Ref: Cell `L0Lb3edIRdj1` output: "Policy Compliance Checker RAG System initialized", "Complete RAG system initialized successfully!"). The system successfully generated both a detailed analysis and a comprehensive report structure, including overall status, scores, and recommendations. (Ref: Cell `NPF1YS3CTJbl` output: "POLICY COMPLIANCE CHECKER - ANALYSIS REPORT").

Highlight Achievements

Subtask:

List the successful implementations, such as the custom LangChain tools, the intelligent rule extraction in the agent, and the overall integration into a complete RAG system capable of generating reports and comparisons.

Achievements and Successful Implementations:

Despite facing API quota issues with the Gemini model, the core architecture and many functional components of the Policy Compliance Checker RAG System were successfully implemented and demonstrated. Here are the key achievements:

1. Effective Text Processing and Chunking:

The `ContractTextProcessor` successfully created manageable text chunks from the raw contract data, including both standard chunks and specifically extracted legal sections. This demonstrated robust text preparation capabilities essential for RAG systems. (Ref: Cell `heM5SfKIRraE` output: "Created 46 chunks from 1 contracts", "Extracted 6 legal sections").

2. Robust Vector Store and Embeddings:

The `ComplianceVectorStore` was successfully initialized, creating high-dimensional embeddings for all processed text units using `SentenceTransformer`. A FAISS index was efficiently built and populated, ready for semantic search. (Ref: Cell `Os0pP8JDRrN0` output: "Embeddings shape: (52, 384)", "FAISS index created with 52 vectors", "Vector store created and tested successfully!").

3. Functional Retrieval Mechanism:

The `retrieve_contract_sections` tool, powered by the FAISS vector store, proved capable of retrieving relevant contract sections based on semantic queries. This validates the core RAG component's ability to fetch contextually similar information. (Ref: Cell `py4RaWPcTJgZ` output: "Retrieved 3 relevant sections for: 'Find termination clauses in the contracts'").

4. Well-Structured Custom LangChain Tools:

Custom LangChain `BaseTool` implementations (`CheckSingleRuleComplianceTool`, `CheckMultipleRulesComplianceTool`, `RetrieveContractSectionsTool`, `CompareContractComplianceTool`) were successfully defined and integrated. The input schemas (`BaseModel`) were correctly configured, enabling clear communication within the agent framework. Although the `compliance_checker` encountered API errors during execution, the design and instantiation of these tools were complete and correctly structured. (Ref: Cell `FEvT2_lUTJkW`, `_CvanSIzTJij` which shows the tool instantiation and test calls).

5. Intelligent Agent Logic for Query Routing:

The `ComplianceAgent` demonstrated intelligent routing capabilities by analyzing user queries and accurately classifying them (e.g., "compliance_check", "retrieval"). This allowed the agent to select and invoke the appropriate custom tool, forming a multi-step reasoning workflow. (Ref: Cell `py4RaWPcTJgZ` output: "Query type: compliance_check", "Query type: retrieval").

6. Comprehensive System Integration and Report Generation:

The `PolicyComplianceRAGSystem` successfully integrated all components – vector store, compliance checker, and agent – into a cohesive system. It showcased the ability to orchestrate complex tasks, such as generating compliance reports and performing multi-rule analyses, producing structured output and visualizations, even with placeholder data due to LLM limitations. (Ref: Cell `L0Lb3edIRdj1` output: "Policy Compliance Checker RAG System initialized", "Complete RAG system initialized successfully!"). The system successfully generated both a detailed analysis and a comprehensive report structure, including overall status, scores, and recommendations. (Ref: Cell `NPF1YS3CTJbl` output: "POLICY COMPLIANCE CHECKER - ANALYSIS REPORT").

Highlight Achievements

Subtask:
List the successful implementations, such as the custom LangChain tools, the intelligent rule extraction in the agent, and the overall integration into a complete RAG system capable of generating reports and comparisons.

Achievements and Successful Implementations:

Despite facing API quota issues with the Gemini model, the core architecture and many functional components of the Policy Compliance Checker RAG System were successfully implemented and demonstrated. Here are the key achievements:

1. Effective Text Processing and Chunking:
   The `ContractTextProcessor` successfully created manageable text chunks from the raw contract data, including both standard chunks and specifically extracted legal sections. This demonstrated robust text preparation capabilities essential for RAG systems. (Ref: Cell `heM5SfKIRraE` output: "Created 46 chunks from 1 contracts", "Extracted 6 legal sections").

2. Robust Vector Store and Embeddings:
   The `ComplianceVectorStore` was successfully initialized, creating high-dimensional embeddings for all processed text units using `SentenceTransformer`. A FAISS index was efficiently built and populated, ready for semantic search. (Ref: Cell `Os0pP8JDRrN0` output: "Embeddings shape: (52, 384)", "FAISS index created with 52 vectors", "Vector store created and tested successfully!").

3. Functional Retrieval Mechanism:

The `retrieve_contract_sections` tool, powered by the FAISS vector store, proved capable of retrieving relevant contract sections based on semantic queries. This validates the core RAG component's ability to fetch contextually similar information. (Ref: Cell `py4RaWPcTJgZ` output: "Retrieved 3 relevant sections for: 'Find termination clauses in the contracts'").

4. Well-Structured Custom LangChain Tools:

Custom LangChain `BaseTool` implementations (`CheckSingleRuleComplianceTool`, `CheckMultipleRulesComplianceTool`, `RetrieveContractSectionsTool`, `CompareContractComplianceTool`) were successfully defined and integrated. The input schemas (`BaseModel`) were correctly configured, enabling clear communication within the agent framework. Although the `compliance_checker` encountered API errors during execution, the design and instantiation of these tools were complete and correctly structured. (Ref: Cell `FEvT2_lUTJkW`, `_CvanSIzTJij` which shows the tool instantiation and test calls).

5. Intelligent Agent Logic for Query Routing:

The `ComplianceAgent` demonstrated intelligent routing capabilities by analyzing user queries and accurately classifying them (e.g., "compliance_check", "retrieval"). This allowed the agent to select and invoke the appropriate custom tool, forming a multi-step reasoning workflow. (Ref: Cell `py4RaWPcTJgZ` output: "Query type: compliance_check", "Query type: retrieval").

6. Comprehensive System Integration and Report Generation:

The `PolicyComplianceRAGSystem` successfully integrated all components – vector store, compliance checker, and agent – into a cohesive system. It showcased the ability to orchestrate complex tasks, such as generating compliance reports and performing multi-rule analyses, producing structured output and visualizations, even with placeholder data due to LLM limitations. (Ref: Cell `L0Lb3edIRdj1` output: "Policy Compliance Checker

RAG System initialized", "Complete RAG system initialized successfully!"). The system successfully generated both a detailed analysis and a comprehensive report structure, including overall status, scores, and recommendations. (Ref: Cell `NPF1YS3CTJbl` output: "POLICY COMPLIANCE CHECKER - ANALYSIS REPORT").

Highlight Achievements

Subtask:
List the successful implementations, such as the custom LangChain tools, the intelligent rule extraction in the agent, and the overall integration into a complete RAG system capable of generating reports and comparisons.

Achievements and Successful Implementations:

Despite facing API quota issues with the Gemini model, the core architecture and many functional components of the Policy Compliance Checker RAG System were successfully implemented and demonstrated. Here are the key achievements:

1. Effective Text Processing and Chunking:
   The `ContractTextProcessor` successfully created manageable text chunks from the raw contract data, including both standard chunks and specifically extracted legal sections. This demonstrated robust text preparation capabilities essential for RAG systems. (Ref: Cell `heM5SfKIRraE` output: "Created 46 chunks from 1 contracts", "Extracted 6 legal sections").

2. Robust Vector Store and Embeddings:
   The `ComplianceVectorStore` was successfully initialized, creating high-dimensional embeddings for all processed text units using `SentenceTransformer`. A FAISS index was efficiently built and populated,

ready for semantic search. (Ref: Cell `Os0pP8JDRrN0` output: "Embeddings shape: (52, 384)", "FAISS index created with 52 vectors", "Vector store created and tested successfully!").

3. Functional Retrieval Mechanism:

The `retrieve_contract_sections` tool, powered by the FAISS vector store, proved capable of retrieving relevant contract sections based on semantic queries. This validates the core RAG component's ability to fetch contextually similar information. (Ref: Cell `py4RaWPcTJgZ` output: "Retrieved 3 relevant sections for: 'Find termination clauses in the contracts'").

4. Well-Structured Custom LangChain Tools:

Custom LangChain `BaseTool` implementations (`CheckSingleRuleComplianceTool`, `CheckMultipleRulesComplianceTool`, `RetrieveContractSectionsTool`, `CompareContractComplianceTool`) were successfully defined and integrated. The input schemas (`BaseModel`) were correctly configured, enabling clear communication within the agent framework. Although the `compliance_checker` encountered API errors during execution, the design and instantiation of these tools were complete and correctly structured. (Ref: Cell `FEvT2_lUTJkW`, `_CvanSIzTJij` which shows the tool instantiation and test calls).

5. Intelligent Agent Logic for Query Routing:

The `ComplianceAgent` demonstrated intelligent routing capabilities by analyzing user queries and accurately classifying them (e.g., "compliance_check", "retrieval"). This allowed the agent to select and invoke the appropriate custom tool, forming a multi-step reasoning workflow. (Ref: Cell `py4RaWPcTJgZ` output: "Query type: compliance_check", "Query type: retrieval").

6. Comprehensive System Integration and Report Generation:

The `PolicyComplianceRAGSystem` successfully integrated all components – vector store, compliance checker, and agent – into a cohesive system. It showcased the ability to orchestrate complex tasks, such as generating compliance reports and performing multi-rule analyses, producing structured output and visualizations, even with placeholder data due to LLM limitations. (Ref: Cell `L0Lb3edIRdj1` output: "Policy Compliance Checker RAG System initialized", "Complete RAG system initialized successfully!"). The system successfully generated both a detailed analysis and a comprehensive report structure, including overall status, scores, and recommendations. (Ref: Cell `NPF1YS3CTJbl` output: "POLICY COMPLIANCE CHECKER - ANALYSIS REPORT").

Highlight Achievements

Subtask:

List the successful implementations, such as the custom LangChain tools, the intelligent rule extraction in the agent, and the overall integration into a complete RAG system capable of generating reports and comparisons.

Achievements and Successful Implementations:

Despite facing API quota issues with the Gemini model, the core architecture and many functional components of the Policy Compliance Checker RAG System were successfully implemented and demonstrated. Here are the key achievements:

1. Effective Text Processing and Chunking:

The `ContractTextProcessor` successfully created manageable text chunks from the raw contract data, including both standard chunks and specifically extracted legal sections. This demonstrated robust text preparation capabilities essential for RAG systems. (Ref: Cell `heM5SfKIRraE` output: "Created 46 chunks from 1 contracts", "Extracted 6 legal sections").

2. Robust Vector Store and Embeddings:

The `ComplianceVectorStore` was successfully initialized, creating high-dimensional embeddings for all processed text units using `SentenceTransformer`. A FAISS index was efficiently built and populated, ready for semantic search. (Ref: Cell `Os0pP8JDRrN0` output: "Embeddings shape: (52, 384)", "FAISS index created with 52 vectors", "Vector store created and tested successfully!").

3. Functional Retrieval Mechanism:

The `retrieve_contract_sections` tool, powered by the FAISS vector store, proved capable of retrieving relevant contract sections based on semantic queries. This validates the core RAG component's ability to fetch contextually similar information. (Ref: Cell `py4RaWPcTJgZ` output: "Retrieved 3 relevant sections for: 'Find termination clauses in the contracts'").

4. Well-Structured Custom LangChain Tools:

Custom LangChain `BaseTool` implementations (`CheckSingleRuleComplianceTool`, `CheckMultipleRulesComplianceTool`, `RetrieveContractSectionsTool`, `CompareContractComplianceTool`) were successfully defined and integrated. The input schemas (`BaseModel`) were correctly configured, enabling clear communication within the agent framework. Although the `compliance_checker` encountered API errors during execution, the design and instantiation of these tools were complete and correctly structured. (Ref: Cell `FEvT2_lUTJkW`, `_CvanSIzTJij` which shows the tool instantiation and test calls).

5. Intelligent Agent Logic for Query Routing:

The `ComplianceAgent` demonstrated intelligent routing capabilities by analyzing user queries and accurately classifying them (e.g., "compliance_check", "retrieval"). This allowed the agent to select and invoke the appropriate custom tool, forming a multi-step reasoning workflow. (Ref:

Cell `py4RaWPcTJgZ` output: "Query type: compliance_check", "Query type: retrieval").

6. Comprehensive System Integration and Report Generation:

The `PolicyComplianceRAGSystem` successfully integrated all components – vector store, compliance checker, and agent – into a cohesive system. It showcased the ability to orchestrate complex tasks, such as generating compliance reports and performing multi-rule analyses, producing structured output and visualizations, even with placeholder data due to LLM limitations. (Ref: Cell `L0Lb3edIRdj1` output: "Policy Compliance Checker RAG System initialized", "Complete RAG system initialized successfully!"). The system successfully generated both a detailed analysis and a comprehensive report structure, including overall status, scores, and recommendations. (Ref: Cell `NPF1YS3CTJbl` output: "POLICY COMPLIANCE CHECKER - ANALYSIS REPORT").

Highlight Achievements

Subtask:
List the successful implementations, such as the custom LangChain tools, the intelligent rule extraction in the agent, and the overall integration into a complete RAG system capable of generating reports and comparisons.

Achievements and Successful Implementations:

Despite facing API quota issues with the Gemini model, the core architecture and many functional components of the Policy Compliance Checker RAG System were successfully implemented and demonstrated. Here are the key achievements:

1. Effective Text Processing and Chunking:

The `ContractTextProcessor` successfully created manageable text chunks from the raw contract data, including both standard chunks and specifically extracted legal sections. This demonstrated robust text preparation capabilities essential for RAG systems. (Ref: Cell `heM5SfKIRraE` output: "Created 46 chunks from 1 contracts", "Extracted 6 legal sections").

2. Robust Vector Store and Embeddings:

The `ComplianceVectorStore` was successfully initialized, creating high-dimensional embeddings for all processed text units using `SentenceTransformer`. A FAISS index was efficiently built and populated, ready for semantic search. (Ref: Cell `Os0pP8JDRrN0` output: "Embeddings shape: (52, 384)", "FAISS index created with 52 vectors", "Vector store created and tested successfully!").

3. Functional Retrieval Mechanism:

The `retrieve_contract_sections` tool, powered by the FAISS vector store, proved capable of retrieving relevant contract sections based on semantic queries. This validates the core RAG component's ability to fetch contextually similar information. (Ref: Cell `py4RaWPcTJgZ` output: "Retrieved 3 relevant sections for: 'Find termination clauses in the contracts'").

4. Well-Structured Custom LangChain Tools:

Custom LangChain `BaseTool` implementations (`CheckSingleRuleComplianceTool`, `CheckMultipleRulesComplianceTool`, `RetrieveContractSectionsTool`, `CompareContractComplianceTool`) were successfully defined and integrated. The input schemas (`BaseModel`) were correctly configured, enabling clear communication within the agent framework. Although the `compliance_checker` encountered API errors during execution, the design and instantiation of these tools were complete and correctly structured. (Ref: Cell `FEvT2_lUTJkW`, `_CvanSIzTJij` which shows the tool instantiation and test calls).

5. Intelligent Agent Logic for Query Routing:

The `ComplianceAgent` demonstrated intelligent routing capabilities by analyzing user queries and accurately classifying them (e.g., "compliance_check", "retrieval"). This allowed the agent to select and invoke the appropriate custom tool, forming a multi-step reasoning workflow. (Ref: Cell `py4RaWPcTJgZ` output: "Query type: compliance_check", "Query type: retrieval").

6. Comprehensive System Integration and Report Generation:

The `PolicyComplianceRAGSystem` successfully integrated all components – vector store, compliance checker, and agent – into a cohesive system. It showcased the ability to orchestrate complex tasks, such as generating compliance reports and performing multi-rule analyses, producing structured output and visualizations, even with placeholder data due to LLM limitations. (Ref: Cell `L0Lb3edIRdj1` output: "Policy Compliance Checker RAG System initialized", "Complete RAG system initialized successfully!"). The system successfully generated both a detailed analysis and a comprehensive report structure, including overall status, scores, and recommendations. (Ref: Cell `NPF1YS3CTJbl` output: "POLICY COMPLIANCE CHECKER - ANALYSIS REPORT").

Highlight Achievements

Subtask:

List the successful implementations, such as the custom LangChain tools, the intelligent rule extraction in the agent, and the overall integration into a complete RAG system capable of generating reports and comparisons.

Achievements and Successful Implementations:

Despite facing API quota issues with the Gemini model, the core architecture and many functional components of the Policy Compliance Checker RAG System were successfully implemented and demonstrated. Here are the key achievements:

1. Effective Text Processing and Chunking:

The `ContractTextProcessor` successfully created manageable text chunks from the raw contract data, including both standard chunks and specifically extracted legal sections. This demonstrated robust text preparation capabilities essential for RAG systems. (Ref: Cell `heM5SfKIRraE` output: "Created 46 chunks from 1 contracts", "Extracted 6 legal sections").

2. Robust Vector Store and Embeddings:

The `ComplianceVectorStore` was successfully initialized, creating high-dimensional embeddings for all processed text units using `SentenceTransformer`. A FAISS index was efficiently built and populated, ready for semantic search. (Ref: Cell `Os0pP8JDRrN0` output: "Embeddings shape: (52, 384)", "FAISS index created with 52 vectors", "Vector store created and tested successfully!").

3. Functional Retrieval Mechanism:

The `retrieve_contract_sections` tool, powered by the FAISS vector store, proved capable of retrieving relevant contract sections based on semantic queries. This validates the core RAG component's ability to fetch contextually similar information. (Ref: Cell `py4RaWPcTJgZ` output: "Retrieved 3 relevant sections for: 'Find termination clauses in the contracts'").

4. Well-Structured Custom LangChain Tools:

Custom LangChain `BaseTool` implementations (`CheckSingleRuleComplianceTool`, `CheckMultipleRulesComplianceTool`, `RetrieveContractSectionsTool`, `CompareContractComplianceTool`) were successfully defined and integrated. The input schemas (`BaseModel`) were

correctly configured, enabling clear communication within the agent framework. Although the `compliance_checker` encountered API errors during execution, the design and instantiation of these tools were complete and correctly structured. (Ref: Cell `FEvT2_lUTJkW`, `_CvanSIzTJij` which shows the tool instantiation and test calls).

5. Intelligent Agent Logic for Query Routing:

The `ComplianceAgent` demonstrated intelligent routing capabilities by analyzing user queries and accurately classifying them (e.g., "compliance_check", "retrieval"). This allowed the agent to select and invoke the appropriate custom tool, forming a multi-step reasoning workflow. (Ref: Cell `py4RaWPcTJgZ` output: "Query type: compliance_check", "Query type: retrieval").

6. Comprehensive System Integration and Report Generation:

The `PolicyComplianceRAGSystem` successfully integrated all components – vector store, compliance checker, and agent – into a cohesive system. It showcased the ability to orchestrate complex tasks, such as generating compliance reports and performing multi-rule analyses, producing structured output and visualizations, even with placeholder data due to LLM limitations. (Ref: Cell `L0Lb3edIRdj1` output: "Policy Compliance Checker RAG System initialized", "Complete RAG system initialized successfully!"). The system successfully generated both a detailed analysis and a comprehensive report structure, including overall status, scores, and recommendations. (Ref: Cell `NPF1YS3CTJbl` output: "POLICY COMPLIANCE CHECKER - ANALYSIS REPORT").

Identify Limitations and Next Steps

Subtask:

Detail the primary limitation (Gemini API rate limiting) and suggest potential solutions or workarounds. Also, propose future enhancements for the system.

Identify Limitations and Next Steps

Subtask:

Detail the primary limitation (Gemini API rate limiting) and suggest potential solutions or workarounds. Also, propose future enhancements for the system.

Limitations and Next Steps

Primary Limitation: Gemini API Rate Limiting

During the development and testing of the Policy Compliance Checker RAG System, a significant limitation encountered was related to the Gemini API's rate limits and quota restrictions. Specifically, the model frequently returned `429 POST` errors, indicating that the request rate or quota was exceeded. This issue has several impacts:

- Reduced Reliability: The compliance checking tools, which rely on the Gemini model for analysis, become unreliable, often failing to process requests, especially when performing multiple checks (e.g., `check_multiple_rules_compliance`, `generate_compliance_comparison`).
- Slowed Development/Testing: Iterative testing and debugging are severely hampered by the need to wait for quota resets or reduce the scope of tests.
- Scalability Concerns: For real-world applications involving many contracts or frequent compliance checks, the current rate limits make the system unscalable without significant adjustments.
- Incomplete Outputs: In some cases, the model's responses during rate limiting can be incomplete or truncated, leading to `KeyError` exceptions

when parsing the output, as seen in the execution logs (e.g., `KeyError: 'risk_level'` due to incomplete Gemini responses).

Potential Solutions and Workarounds for Gemini API Limitation

To address the Gemini API rate limiting, the following strategies can be considered:

1. Optimize API Calls: Review the prompts and logic to ensure minimal API calls are made. Combine multiple small queries into larger, more comprehensive ones where possible.
2. Implement Retry Mechanism with Exponential Backoff: Build a robust retry logic that automatically re-sends failed requests after increasing delays. This helps manage temporary rate limit spikes.
3. Explore Alternative Models: Investigate and integrate other large language models (LLMs) that might offer more flexible rate limits, higher quotas, or a more cost-effective pricing structure for the required scale. Options could include:
    Open-source LLMs: Deploying models like Llama 3, Mistral, or Gemma locally or on cloud-based GPU instances, offering full control over rate limits.
    Other Commercial APIs: Evaluate alternatives like OpenAI's GPT models or Anthropic's Claude, which might have different rate limit policies.
4. Local Model Deployment: For privacy-sensitive or high-throughput scenarios, consider deploying a smaller, fine-tuned LLM directly on dedicated hardware or cloud VMs. This eliminates reliance on external API quotas.
5. Batch Processing: For compliance checks that don't require immediate responses, implement batch processing to space out API calls over a longer period, adhering to rate limits.
6. Increase Quota: If feasible and within budget, apply for a higher API quota from Google Cloud for the Gemini API.

Future Enhancements for the RAG System

Beyond addressing the immediate API limitations, several enhancements can significantly improve the Policy Compliance Checker RAG System's capabilities and user experience:

1. Improved Rule Extraction and Management: Develop more sophisticated methods for automatically extracting rules from regulatory documents or legal texts, reducing manual rule creation. Implement a system for versioning and managing rules.
2. Expanded Rule Base: Continuously grow the library of compliance rules to cover a broader range of legal domains and contract types.
3. Advanced RAG Retrieval Strategies: Enhance the retrieval component to include:
    Re-ranking: After initial retrieval, use a re-ranking model to score retrieved chunks for relevance more accurately.
    Hybrid Search: Combine keyword search (e.g., BM25) with vector search for better recall and precision.
    Graph-based Retrieval: Represent contracts as knowledge graphs to enable more nuanced and relational querying.
4. Conversational Memory for the Agent: Implement more advanced conversational memory beyond simple buffer memory, allowing the agent to maintain context over longer interactions and multi-turn conversations. This could involve summarization, entity extraction, and structured memory.
5. Dynamic Context Window Management: Automatically adjust the amount of contract text passed to the LLM based on query complexity and available token limits, optimizing performance and cost.
6. Enhanced Output Formatting and Visualization: Improve the readability and interpretability of compliance reports with more structured JSON/XML outputs and advanced data visualizations (e.g., interactive dashboards, timeline views of contract changes).

7. User Interface (UI): Develop a simple web-based UI for easier interaction with the system, allowing users to upload contracts, select rules, and view reports intuitively.
8. Automated Remediation Suggestions: Refine the LLM's ability to not just identify non-compliance but also provide more actionable, context-aware suggestions for remediation or clause drafting.
9. Integration with Legal Databases: Connect the system with external legal databases or regulatory feeds to automatically update rules and retrieve relevant legal precedents.

By addressing these limitations and implementing these enhancements, the Policy Compliance Checker RAG System can evolve into a more robust, scalable, and intelligent legal AI assistant.

Identify Limitations and Next Steps

Subtask:
Detail the primary limitation (Gemini API rate limiting) and suggest potential solutions or workarounds. Also, propose future enhancements for the system.

Limitations and Next Steps

Primary Limitation: Gemini API Rate Limiting

During the development and testing of the Policy Compliance Checker RAG System, a significant limitation encountered was related to the Gemini API's rate limits and quota restrictions. Specifically, the model frequently returned `429 POST` errors, indicating that the request rate or quota was exceeded. This issue has several impacts:

Reduced Reliability: The compliance checking tools, which rely on the Gemini model for analysis, become unreliable, often failing to process

requests, especially when performing multiple checks (e.g., `check_multiple_rules_compliance`, `generate_compliance_comparison`).

Slowed Development/Testing: Iterative testing and debugging are severely hampered by the need to wait for quota resets or reduce the scope of tests.

Scalability Concerns: For real-world applications involving many contracts or frequent compliance checks, the current rate limits make the system unscalable without significant adjustments.

Incomplete Outputs: In some cases, the model's responses during rate limiting can be incomplete or truncated, leading to `KeyError` exceptions when parsing the output, as seen in the execution logs (e.g., `KeyError: 'risk_level'` due to incomplete Gemini responses).

Potential Solutions and Workarounds for Gemini API Limitation

To address the Gemini API rate limiting, the following strategies can be considered:

1. Optimize API Calls: Review the prompts and logic to ensure minimal API calls are made. Combine multiple small queries into larger, more comprehensive ones where possible.
2. Implement Retry Mechanism with Exponential Backoff: Build a robust retry logic that automatically re- sends failed requests after increasing delays. This helps manage temporary rate limit spikes.
3. Explore Alternative Models: Investigate and integrate other large language models (LLMs) that might offer more flexible rate limits, higher quotas, or a more cost-effective pricing structure for the required scale. Options could include:

    Open-source LLMs: Deploying models like Llama 3, Mistral, or Gemma locally or on cloud-based GPU instances, offering full control over rate limits.

    Other Commercial APIs: Evaluate alternatives like OpenAI's GPT models or Anthropic's Claude, which might have different rate limit policies.

4. Local Model Deployment: For privacy-sensitive or high-throughput scenarios, consider deploying a smaller, fine-tuned LLM directly on dedicated hardware or cloud VMs. This eliminates reliance on external API quotas.
5. Batch Processing: For compliance checks that don't require immediate responses, implement batch processing to space out API calls over a longer period, adhering to rate limits.
6. Increase Quota: If feasible and within budget, apply for a higher API quota from Google Cloud for the Gemini API.

Future Enhancements for the RAG System

Beyond addressing the immediate API limitations, several enhancements can significantly improve the Policy Compliance Checker RAG System's capabilities and user experience:

1. Improved Rule Extraction and Management: Develop more sophisticated methods for automatically extracting rules from regulatory documents or legal texts, reducing manual rule creation. Implement a system for versioning and managing rules.
2. Expanded Rule Base: Continuously grow the library of compliance rules to cover a broader range of legal domains and contract types.
3. Advanced RAG Retrieval Strategies: Enhance the retrieval component to include:
    Re-ranking: After initial retrieval, use a re-ranking model to score retrieved chunks for relevance more accurately.
    Hybrid Search: Combine keyword search (e.g., BM25) with vector search for better recall and precision.
    Graph-based Retrieval: Represent contracts as knowledge graphs to enable more nuanced and relational querying.
4. Conversational Memory for the Agent: Implement more advanced conversational memory beyond simple buffer memory, allowing the agent to

maintain context over longer interactions and multi-turn conversations. This could involve summarization, entity extraction, and structured memory.
5. Dynamic Context Window Management: Automatically adjust the amount of contract text passed to the LLM based on query complexity and available token limits, optimizing performance and cost.
6. Enhanced Output Formatting and Visualization: Improve the readability and interpretability of compliance reports with more structured JSON/XML outputs and advanced data visualizations (e.g., interactive dashboards, timeline views of contract changes).
7. User Interface (UI): Develop a simple web-based UI for easier interaction with the system, allowing users to upload contracts, select rules, and view reports intuitively.
8. Automated Remediation Suggestions: Refine the LLM's ability to not just identify non-compliance but also provide more actionable, context-aware suggestions for remediation or clause drafting.
9. Integration with Legal Databases: Connect the system with external legal databases or regulatory feeds to automatically update rules and retrieve relevant legal precedents.

By addressing these limitations and implementing these enhancements, the Policy Compliance Checker RAG System can evolve into a more robust, scalable, and intelligent legal AI assistant.

Compile Final Evaluation Report

Subtask:

Synthesize all the gathered information into a structured evaluation report for Task 02, presenting a clear picture of the project's status, successes, and areas for improvement.

Policy Compliance Checker RAG System - Final Evaluation Report

1. Introduction

This report provides a comprehensive evaluation of the Policy Compliance Checker RAG (Retrieval-Augmented Generation) System developed for Task 02. The system aims to automate contract compliance analysis by leveraging advanced NLP techniques, including embeddings, vector stores, and a generative AI model (Gemini) integrated with LangChain tools. The evaluation covers the system's architecture, component functionality, achievements, identified limitations, and recommendations for future development.

2. System Components

The Policy Compliance Checker RAG System is comprised of several key components:

CUAD Dataset Integration: The system is designed to work with the CUAD (Contract Understanding Atticus Dataset), loading contracts for analysis.

Text Processing (Chunking): Contracts are processed into smaller, manageable `chunks` and `legal sections` using `RecursiveCharacterTextSplitter` to facilitate efficient retrieval. The system generated 46 standard chunks and 6 legal sections from the sample data.

Vector Store: A `FAISS` vector store is utilized to store `embeddings` of the contract chunks. The `all-MiniLM-L6-v2` SentenceTransformer model is used for generating 384-dimensional embeddings. The vector store allows for rapid retrieval of relevant contract sections based on a given query.

Compliance Rules: A predefined set of 18 compliance rules, inspired by CUAD categories, are used to evaluate contract clauses. Each rule includes a description, compliance requirement, severity, and suggested remediation.

Gemini API Integration: The system integrates with the `Google Gemini API` for generative AI capabilities, used by the `ComplianceCheckerTools` to analyze contract text against specific rules.

LangChain Tools: Custom `LangChain tools` (`CheckSingleRuleComplianceTool`, `CheckMultipleRulesComplianceTool`, `RetrieveContractSectionsTool`, `CompareContractComplianceTool`) encapsulate specific functionalities, making them accessible to an AI agent.

Compliance Agent: An `Agent Workflow` is implemented to process multi-step compliance questions by intelligently using the available `LangChain tools` and maintaining conversational memory.

Compliance Comparison and Reporting: The system generates compliance comparison tables, summary statistics, and visualizations (e.g., heatmaps, bar charts) to present evaluation results, highlighting compliant and non-compliant areas.

3. Individual Component Functionality Evaluation

Data Loading and Preprocessing: Successfully loaded sample contract data and created chunks. The text processor effectively segmented contracts into meaningful units, extracting both general chunks and specific legal sections.

Embeddings and Vector Store: The `SentenceTransformer` model successfully created embeddings, and the `FAISS` index was built and tested for retrieval. Retrieval of similar chunks for queries like 'governing law jurisdiction' and 'termination clause' demonstrated its functionality, albeit with `CUAD_v1_README.txt` content as the primary source, which is not a typical contract.

Gemini API and Compliance Checker: The Gemini API was configured, and `ComplianceCheckerTools` were initialized. However, during testing, recurrent `429 POST` errors (rate limits) and `KeyError: 'risk_level'` were encountered, preventing full functional testing of the Gemini-powered compliance checks. This led to 'ERROR' statuses and 0 scores in most compliance results.

LangChain Tools: The custom LangChain tools were successfully defined and integrated. Initial testing, despite the underlying Gemini API issues, confirmed their structure and ability to call the `compliance_checker` methods.

Agent Workflow: The `ComplianceAgent` was initialized and demonstrated its ability to interpret queries (e.g., 'compliance_check', 'retrieval') and attempt to use the appropriate tools. The agent's `_extract_rule_name` method showed a robust approach to identify rules from natural language queries.

Compliance Comparison and Reporting: The system successfully generated comparison dataframes and visualizations (heatmaps, bar charts) based on the (limited) results from the compliance checks. This demonstrated the reporting capabilities, even with erroneous compliance scores.

4. Achievements

End-to-End RAG System Architecture: Successfully designed and implemented an integrated RAG system from data ingestion and chunking to vector storage, LLM integration, LangChain tools, and an intelligent agent.

Custom Compliance Rule Definition: Developed a structured approach to define compliance rules based on CUAD categories, enabling flexible policy enforcement.

Modular Design with LangChain: Leveraged LangChain's tool abstraction to create a modular and extensible system, allowing the agent to dynamically interact with different compliance functionalities.

Data Visualization for Insights: Implemented effective data visualization techniques (heatmaps, bar charts) to present complex compliance results in an easily digestible format, aiding in quick identification of high-risk areas.

Interactive Agent for Multi-Step Queries: The agent demonstrated potential for handling multi-step user queries, a critical feature for a comprehensive compliance analysis tool.

5. Limitations and Next Steps

API Quota Limitations: The most significant limitation encountered during this task was the frequent `429 (Too Many Requests)` error from the Gemini API. This severely hampered the real-time compliance checking functionality

and skewed test results to 'ERROR' or 0 scores. Next Step: Implement robust API error handling with retry mechanisms and consider using a paid tier or alternative local LLM for higher throughput, or reducing the number of LLM calls in the evaluation phase.

Dataset Scope: The current implementation primarily used the `CUAD_v1_README.txt` file as the 'contract' due to issues in extracting other contract types (PDFs, DOCXs) from the provided zip. This meant actual contract compliance couldn't be fully tested. Next Step: Ensure proper extraction and loading of diverse contract documents (PDF, DOCX, TXT) from the CUAD dataset to truly test the compliance checking against real-world contracts.

Parsing Robustness: The `_parse_compliance_response` method in `ComplianceCheckerTools` experienced a `KeyError: 'risk_level'`, indicating the Gemini response format was not consistently matching expected patterns. Next Step: Enhance the parsing logic to be more resilient to variations in LLM output, potentially using `JsonOutputParser` or more flexible regex, and ensure all expected fields (`risk_level`, `evidence`, `suggestions`) are always returned or gracefully handled with defaults.

Scalability for Large Contracts: While chunking is implemented, the `check_rule_compliance` method still passes up to 3000 characters of contract text. For very large contracts, this might still hit token limits or reduce efficiency. Next Step: Optimize chunk selection for compliance checks by using retrieval to provide only the most relevant sections for a given rule to the LLM.

Refinement of Rule Extraction: While the `_extract_rule_name` method is a good start, it could be further refined using more advanced NLP techniques (e.g., named entity recognition, semantic similarity) to accurately identify rules from complex queries.

6. Conclusion

The Policy Compliance Checker RAG System successfully establishes a foundational architecture for automated contract compliance analysis. Despite

significant challenges posed by API quota limitations and dataset loading, the system demonstrates strong potential through its modular design, intelligent agent, and comprehensive reporting capabilities. Addressing the identified limitations, particularly the API stability and robust content parsing, will be crucial for moving this project towards a fully functional and reliable solution for legal tech applications.

Final Task

Subtask:

Provide the complete evaluation report, summarizing the entire Task 02 development and its current state.

Summary:

Data Analysis Key Findings

System Architecture Successfully Defined: The core architectural components of the Policy Compliance Checker RAG System, including compliance rules, text processing, vector store, custom LangChain tools, and a multi-step agent workflow, were clearly outlined.

Effective Text Processing and Vectorization: The `ContractTextProcessor` successfully created 46 chunks and extracted 6 legal sections from a contract. The `ComplianceVectorStore` effectively generated 52 embeddings of 384 dimensions using "all-MiniLM-L6-v2" and built a functional FAISS index for semantic search.

Functional Retrieval Mechanism: The `retrieve_contract_sections` tool, relying on the vector store, successfully retrieved relevant sections for queries like "Find termination clauses in the contracts", demonstrating a working core RAG component.

Well-Structured Custom LangChain Tools and Agent Logic: The custom LangChain tools were correctly defined with appropriate input schemas, and the `ComplianceAgent` demonstrated intelligent routing by accurately classifying user queries (e.g., "compliance_check", "retrieval") and attempting to invoke the correct tools.
Comprehensive System Integration and Report Generation (Structural Success): The `PolicyComplianceRAGSystem` successfully integrated all components and generated structured reports, including analysis tables and visualizations, showcasing its capability to orchestrate complex tasks, even with placeholder analytical content.
Primary Limitation: Gemini API Rate Limiting: All components dependent on the Gemini API for compliance analysis (e.g., `ComplianceCheckerTools`, LangChain tools for compliance, agent's compliance checks) consistently failed due to `429 POST` errors, indicating rate limiting or quota issues. This led to `KeyError: 'risk_level'` in parsing responses and `Status: ERROR` with `Score: 0/10` across all compliance-related evaluations.

Insights or Next Steps

Prioritize API Stability: The immediate next step is to address the Gemini API rate limiting issue by implementing robust retry mechanisms with exponential backoff, optimizing API call frequency, and actively exploring alternative LLMs (open-source or other commercial APIs) or requesting an increased quota to ensure the system's core analytical capabilities function reliably.
Enhance Data Handling and Parsing Robustness: Improve the system's ability to handle diverse contract formats by ensuring proper extraction from various document types (PDF, DOCX) and refine the parsing logic for LLM responses to be more resilient to incomplete or varied output structures, potentially using structured output parsers.

End of Assignment