Ca' Foscari
University
of Venice

Master Degree
in Data Analytics for Business and Society

Final Thesis

RAG Chatbot System

Supervisor
Ch. Prof. Raffaele Pesenti

Graduand
Paolo Astrino
Matriculation Number 903542

Academic Year
2024 / 2025

Index

1 Introduction

The exponential growth of digital information has made it increasingly challenging for individuals and organizations to efficiently access and utilize knowledge stored in heterogeneous document formats such as PDF, CSV, and JSON. Traditional keyword-based search methods often fail to address complex queries or synthesize information across multiple documents, limiting their effectiveness for advanced information retrieval tasks [1, 2, 3].

Another challenge arises from the nature of this information, which is often sensitive, proprietary, or legally protected. State-of-the-art AI language models, such as ChatGPT, require uploading data to external cloud servers for processing, creating significant barriers to their use [4, 5, 6].

Sharing documents with third-party cloud services presents significant limitations, particularly in regulated industries or organizations handling confidential data. This constraint restricts the application of AI tools for knowledge discovery and raises concerns regarding data security, compliance, and intellectual property protection [4, 5, 6].

Traditional keyword search tools offer limited assistance, often failing to understand context or synthesize information across multiple documents [7]. As a result, there is a need for AI systems that can perform advanced, conversational analysis of local documents without exposing sensitive data to external servers [7, 8].

This project addresses these challenges by developing a chatbot system that uses Retrieval-Augmented Generation (RAG) [9] an AI approach that combines the capabilities of large language models (LLMs) and targeted information retrieval. This integration ensures that the generated content is up-to-date and grounded in relevant data, improving the model's performance in tasks that requires domain-specific knowledge [27] .

The RAG architecture consists of three core components:

- **Retrieval:** The system searches through local documents (PDFs, spreadsheets, etc.) to find relevant information related to user queries.
- **Augmentation:** Retrieved data is processed and incorporated with the user query to provide context for the AI model.
- **Generation:** Finally, the AI generates a natural, human-like response that directly references the retrieved information.

Unlike cloud-based AI tools, this system operates on local infrastructure, eliminating the need to share sensitive data with external servers. By integrating RAG with a modular, protocol-driven design inspired by the Model Context Protocol (MCP) [10, 11], the chatbot provides a secure, scalable solution for organizations that need to use AI without compromising data privacy or compliance.

The system employs a client-server architecture, leveraging technologies such as Python, Flask, LangChain, HuggingFace Transformers, and standardized JSON-based communication [19]. The main features include support for multiple document formats, hybrid retrieval strategies (combining semantic and keyword search), GPU acceleration for processing, and server-side management of sensitive credentials to ensure security. This approach aligns with established frameworks for building enterprise-grade RAG-based chatbots that emphasize security, architecture, and testing dimensions [20, 21, 22, 23].

The system undergoes comprehensive evaluation using established benchmarks (SQuAD, Natural Questions, MS MARCO) through four key metric categories: coverage measures (Recall@K, Hit Rate@K), ranking quality measures (Mean Reciprocal Rank, MRR@10), extractive fidelity measures (exact match rates), and distributional statistics (mean rank, median rank, rank-1 counts) [23, 24]. This multi-dimensional evaluation framework assesses not only retrieval success but also result ordering, span-level accuracy, and placement distribution, complemented by qualitative assessments to ensure comprehensive system performance analysis [25, 26].

This thesis is organized into four main chapters. The first chapter presents the overall architecture and design of the system, detailing the modular client-server approach, the Model Context Protocol (MCP), and the communication flow between components. The second chapter focuses on the implementation of the Retrieval-Augmented Generation (RAG) pipeline, highlighting the integration of LangChain, HuggingFace, and hybrid retrieval strategies. The third chapter describes the evaluation methodology, including the datasets used, metrics, and results analysis. The final chapter discusses the system's strengths, limitations, and potential directions for future work, followed by concluding remarks summarizing the key contributions of the project.

1.1 Literature Review

1.1.1 Retrieval-Augmented Generation (RAG) Fundamentals

The emergence of large language models (LLMs) has revolutionized natural language processing, yet these systems face critical limitations including hallucination and knowledge cutoff dates that render them unsuitable for many enterprise applications. Retrieval-Augmented Generation (RAG) addresses these challenges by combining the generative capabilities of LLMs with external knowledge retrieval systems.

Lewis [13] introduced the foundational RAG architecture, establishing a three-stage pipeline: retrieval of relevant documents from an external corpus, augmentation of the input query with retrieved context, and generation of responses using this enriched information. This paradigm shift moved away from parameter-only knowledge storage toward dynamic knowledge access, enabling LLMs to leverage up-to-date and domain-specific information without requiring expensive retraining.

Recent comprehensive surveys [35, 38] have mapped the rapid evolution of RAG architectures, identifying key variants including Fusion-in-Decoder (FiD), REALM, and T5-based approaches. These works highlight RAG's fundamental advantage over fine-tuning: the ability to incorporate new information without model modification, making it particularly suitable for dynamic enterprise environments where document collections frequently change.

The hybrid retrieval approach presented in this work builds upon these foundations by addressing a critical gap in existing RAG systems: the over-reliance on either semantic similarity or keyword matching. While Lewis et al.'s original work focused primarily on dense retrieval, our system recognizes that enterprise document QA requires both semantic understanding and precise keyword matching, leading to the development of a weighted hybrid approach that combines both paradigms.

1.1.2 Document-Based Question Answering Systems

Document-based question answering has evolved from simple keyword search systems to sophisticated semantic understanding platforms, yet significant challenges remain in enterprise environments. Traditional information retrieval systems, built on foundations like the Vector Space Model [14] and probabilistic models [15], excel at exact matching but struggle with semantic intent and conversational queries.

The transition to neural approaches began with reading comprehension datasets like SQuAD [46], MS MARCO [48] and Natural Question [47], which demonstrated the potential for machine reading at scale. However, these benchmarks primarily focused on single-document scenarios, leaving multi-document enterprise use cases underexplored. Enterprise document QA faces unique challenges including diverse document formats, large-scale collections, heterogeneous content types, and the need for precise attribution of sources.

Knowledge management systems in enterprise environments have traditionally relied on structured databases and taxonomies, but the explosion of unstructured content has created a semantic gap. Systems like IBM Watson and Microsoft Cognitive Search have attempted to bridge this gap through hybrid approaches, yet they often require significant configuration overhead and lack the conversational interfaces that modern users expect.

The limitations of purely keyword-based search become apparent in enterprise scenarios where users express information needs in natural language rather than query terms. Conversely, purely semantic approaches can miss important exact matches and struggle with technical terminology. This gap justifies our multi-format support approach, which handles PDFs, Word documents, and plain text while maintaining both semantic and lexical search capabilities to serve diverse enterprise document collections effectively.

1.1.3 Hybrid Retrieval Strategies

The debate between dense and sparse retrieval methods has shaped modern information retrieval research, with each approach offering distinct advantages. Sparse retrieval methods, exemplified by BM25 [**?**], excel at exact matching and have proven robust across diverse domains. The BM25 algorithm's term frequency and inverse document frequency components capture important signals about document relevance, particularly for queries containing specific technical terms or proper nouns common in enterprise documents.

Dense retrieval emerged with transformer-based embedding models, with significant developments including Sentence-BERT [54] which demonstrated how to derive semantically meaningful sentence embeddings using siamese BERT networks. Modern embedding models like BGE [53] have further advanced

the field by providing high-quality general-purpose embeddings that excel across diverse domains. These approaches encode queries and documents into dense vector representations, enabling semantic similarity matching that captures conceptual relationships beyond lexical overlap. However, dense retrieval can struggle with out-of-vocabulary terms and may miss important exact matches.

Recent work on hybrid fusion techniques has attempted to combine the strengths of both approaches. The theoretical foundations for combining sparse and dense retrieval have been well-established [**?**], while practical implementations have demonstrated that linear combination of sparse and dense scores often outperforms individual methods. However, most existing work focuses on general-domain datasets rather than enterprise document collections, leaving optimal weighting strategies for business documents underexplored.

Our exploration of hybrid retrieval configurations addresses the fundamental question of optimal weighting strategies for enterprise document collections. While existing work has demonstrated that linear combination of sparse and dense scores often outperforms individual methods, the optimal balance remains an empirical question that depends on document characteristics, query types, and domain requirements. This work systematically evaluates multiple hybrid configurations to determine the most effective weighting strategies for enterprise document QA scenarios, where users may query for both conceptual information and specific terms or values.

1.1.4 System Architecture Patterns for AI Applications

The architecture of AI-powered systems has evolved from monolithic applications toward modular, distributed designs that separate concerns and enable scalability. Traditional AI applications often tightly coupled model inference, data processing, and user interfaces, creating systems that were difficult to maintain and scale. Modern patterns emphasize service-oriented architectures that decompose functionality into discrete, communicating components.

The Model Context Protocol (MCP) [10], introduced by Anthropic in 2024, represents a significant advancement in AI system architecture by standardizing the interface between AI models and external resources. MCP defines a protocol for secure, structured communication between language models and various data sources, tools, and services. This protocol addresses critical challenges in AI system design including credential management, resource access control, and standardized communication patterns.

Client-server architectures have emerged as the dominant pattern for AI applications due to their advantages in resource management, security, and scalability. The server component can leverage specialized hardware (GPUs) for computationally intensive tasks like embedding generation, while client components provide user interfaces optimized for specific platforms. This separation enables organizations to centralize AI capabilities while providing diverse access methods.

Microservices patterns further decompose AI systems into specialized services for document processing, embedding generation, retrieval, and generation. This approach enables independent scaling of components based on demand, technology stack diversity, and easier testing and maintenance. However, microservices introduce complexity in service discovery, communication overhead, and distributed system challenges.

Our MCP-inspired architecture choice addresses these concerns by providing a standardized interface that enables modular design while maintaining simplicity. The protocol-based approach allows the system to integrate with various AI models and data sources while maintaining security boundaries and enabling future extensibility. This design pattern supports the separation of document processing, retrieval, and generation concerns while providing a cohesive user experience.

1.1.5 Security and Privacy in Enterprise AI

Enterprise adoption of AI systems faces significant security and privacy challenges, particularly when dealing with sensitive document collections. Cloud-based LLM services, while powerful and convenient, raise concerns about data sovereignty, compliance with regulations like GDPR and HIPAA, and the risk of sensitive information exposure through model training or inadvertent logging.

Data privacy concerns with cloud-based LLMs extend beyond direct data transmission to include the potential for inference attacks, where sensitive information might be deduced from usage patterns or model responses. Organizations in regulated industries face additional constraints on data processing location and third-party access, making cloud-based solutions problematic for many enterprise use cases.

Local processing approaches address these concerns by keeping sensitive data within organizational boundaries. However, local deployment introduces challenges including hardware requirements, model management overhead, and the need for specialized expertise. The trade-off between convenience and

control has led to hybrid approaches that process non-sensitive operations in the cloud while maintaining sensitive operations locally.

Credential management in AI systems requires careful consideration of authentication, authorization, and audit trails. Traditional API key approaches often lack granular access control and create security risks through key proliferation. Modern approaches emphasize token-based authentication with short-lived credentials and role-based access control that aligns with organizational security policies.

The security-first design approach in this work addresses these challenges by prioritizing local document processing and implementing secure credential management patterns. By keeping document content processing within organizational boundaries while enabling flexible model access through standardized protocols, the system provides enterprise-grade security without sacrificing functionality. This approach recognizes that many organizations require local processing solutions to meet compliance and security requirements while still benefiting from advanced AI capabilities.

1.1.6 Evaluation Methodologies for RAG Systems

Evaluating retrieval-augmented generation systems presents unique challenges that combine information retrieval evaluation with natural language generation assessment. Traditional IR evaluation methodologies [12] provide frameworks for assessing retrieval quality through metrics like precision (the fraction of retrieved documents that are relevant), recall (the fraction of relevant documents that are retrieved), and normalized discounted cumulative gain (NDCG, which measures ranking quality with position-based discounting). However, these retrieval-focused measures may not correlate with downstream generation performance, as a system might retrieve relevant documents but fail to synthesize them effectively. Conversely, generation metrics like BLEU (which measures n-gram overlap with reference texts) and ROUGE (which evaluates recall of n-grams, word sequences, and word pairs) measure text similarity but may not reflect factual accuracy or relevance to the original query.

Standard benchmarks including SQuAD [46], Natural Questions [47], and MS MARCO [48] have driven progress in reading comprehension but often focus on single-document scenarios with clear answer spans. These benchmarks may not adequately represent enterprise document QA scenarios where answers require synthesis across multiple documents or where the optimal response may not exist in the source material.

Recent work has explored LLM-as-Judge approaches [45] where large language models evaluate the quality of generated responses across multiple dimensions including accuracy, completeness, and relevance. Additionally, specialized evaluation methodologies for RAG systems [44] have emerged to address the unique challenges of assessing retrieval-augmented generation performance. The faithfulness of RAG systems—ensuring generated responses accurately reflect retrieved information—has become a critical evaluation dimension [9].

The challenge of evaluating retrieval quality versus generation quality separately has led to component-wise evaluation strategies. Retrieval evaluation focuses on whether relevant documents are retrieved, while generation evaluation assesses whether the retrieved context is effectively utilized. However, this separation may miss important interactions between retrieval and generation components.

End-to-end evaluation approaches that assess overall system performance on realistic tasks provide more holistic quality measures but require carefully constructed evaluation datasets that reflect real usage patterns. The development of domain-specific evaluation datasets remains an active area of research, particularly for enterprise applications where ground truth may be subjective or context-dependent.

Our evaluation approach addresses these challenges by combining multiple complementary metrics that assess both retrieval effectiveness and generation quality. The choice of datasets and metrics reflects the recognition that enterprise document QA requires evaluation approaches that capture both factual accuracy and user satisfaction in realistic usage scenarios.

1.1.7 Performance Optimization in AI Systems

Performance optimization in AI-powered document QA systems involves addressing computational bottlenecks across multiple system components including embedding generation, document processing, and retrieval operations. GPU acceleration has become essential for transformer-based embedding models, where the parallelizable nature of attention mechanisms can provide significant speedups over CPU-only implementations [42]. Modern frameworks like LangChain [39] have emerged to facilitate the development of complex AI applications by providing standardized components for document processing, embedding generation, and chain-of-thought reasoning.

Embedding optimization strategies include model quantization, knowledge distillation, and architectural modifications that reduce computational requirements while maintaining quality. Recent work on efficient transformer architectures, including distilled models like DistilBERT [16] and specialized embedding models like E5 [17], demonstrates significant performance improvements with minimal quality degradation.

Caching strategies play a critical role in system performance, particularly for embedding generation where the same documents may be processed multiple times. Multi-level caching approaches that cache both computed embeddings and intermediate processing results can dramatically reduce response times for frequently accessed content. However, cache invalidation strategies must account for document updates and model changes.

Large file handling presents additional challenges in enterprise environments where documents may exceed typical memory constraints. Streaming processing approaches that chunk documents while maintaining semantic coherence enable processing of arbitrarily large files. However, chunking strategies must balance computational efficiency with semantic preservation to avoid fragmenting important information.

Vector storage and similarity search optimization has benefited from advances in approximate nearest neighbor algorithms including FAISS [18] and Annoy. These systems enable sub-linear similarity search across large embedding collections, making real-time retrieval feasible for enterprise-scale document collections.

The performance optimization decisions in this work, including GPU acceleration for embedding generation and intelligent caching strategies, address the scalability challenges inherent in document-heavy applications. By leveraging hardware acceleration where it provides maximum benefit and implementing efficient caching patterns, the system maintains responsive performance even as document collections grow to enterprise scale.

1.1.8 Conclusion

This literature review establishes the foundation for hybrid retrieval-augmented generation systems in enterprise document QA applications. The convergence of RAG architectures, hybrid retrieval strategies, and security-conscious system design creates opportunities for systems that address real-world enterprise needs while maintaining the flexibility and performance required for practical deployment.

The gaps identified in current research—particularly the need for balanced hybrid retrieval, security-first architectures, and comprehensive evaluation methodologies—justify the contributions of this work. By building upon established foundations while addressing practical deployment challenges, the proposed system advances the state of practice in enterprise document QA systems.

2 Chapter 1: System Architecture and Design

This chapter presents the overall system architecture, detailing the modular client-server design, the custom JSON communication protocol, and the specific responsibilities of the frontend, client, and server components. It also explains the supported commands, interaction flow, and the security measures implemented for credential management and data protection.

2.1 Overview of the Modular Client-Server Architecture

The system is organized into a modular client-server architecture based on the Model Context Protocol (MCP), which standardizes how applications provide context to large language models (LLMs) [28, 10, 19]. The server component is responsible for document ingestion, retrieval-augmented generation (RAG) logic, and communication with external LLM APIs. It manages file uploads, document parsing, hybrid retrieval (semantic and keyword search), and maintains chat history. The client component provides a Flask-based HTTP API and web interface [30], handling user interactions, file uploads, and forwarding user commands to the server via sockets [29].

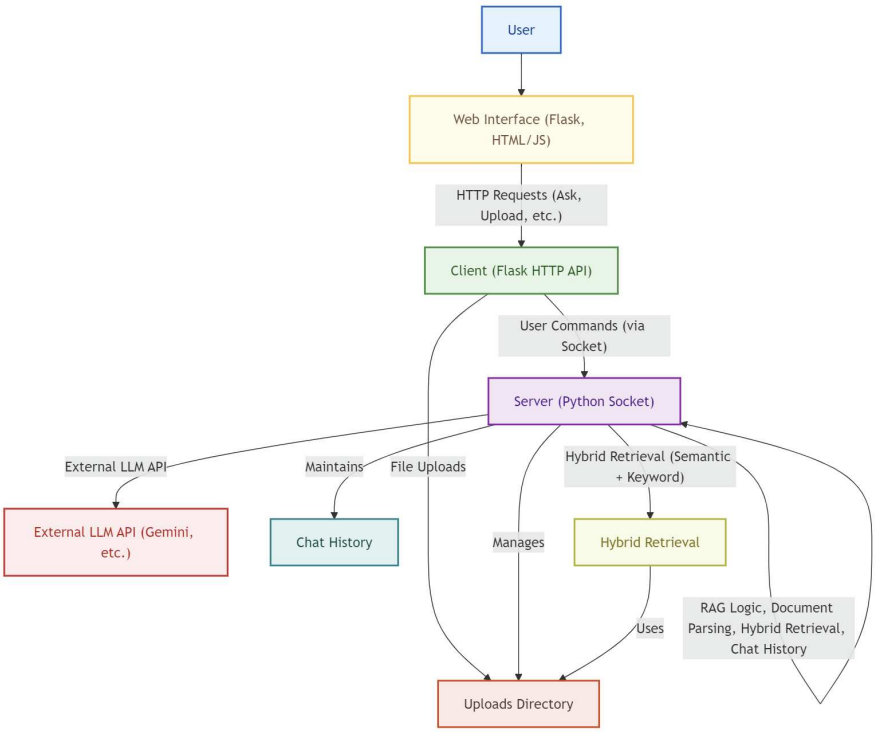

Figure 1: System Architecture Overview

In Figure 1 we can see the system architecture overview, which illustrates the modular client-server design that forms the foundation of our RAG chatbot system.

Starting from the top, the diagram shows four primary components: the User, who interacts with the Web Interface, the Client component that manages API endpoints, the Server component that handles core RAG functionality, and external services that provide LLM capabilities.

The flow of data and commands between these components is structured and unidirectional, ensuring clean separation of concerns.

The User initiates interactions through the Web Interface (implemented with Flask, HTML, and JavaScript). This interface sends HTTP requests to the Client, which functions as a protocol bridge, translating user inputs into standardized JSON commands. The Client manages file uploads, storing them in the shared Uploads Directory, and forwards user commands to the Server via socket connections.

The Server component, shown at the center of the diagram, encapsulates all core functionality. It receives commands through its Socket interface and processes them via the Command Handler. For queries requiring retrieval and generation, the RAG Logic module coordinates a pipeline structured in this way: it accesses uploaded documents, performs hybrid retrieval (combining semantic embeddings with keyword search), maintains conversation history, and constructs appropriate prompts for the LLM.

When responses are needed, the Server communicates with an external LLM API (such as Google Gemini) to generate contextually relevant answers based on retrieved information.

The architecture explicitly isolates sensitive operations on the server side. All credential management, RAG processing logic, and document handling occur in this secure environment, while the client merely facilitates communication without accessing sensitive data or credentials.

Table 1 details the system components, highlighting their implementation technologies and specific responsibilities within the common roles of an MCP system [28]. Dividing the architecture into five distinct modules enforces separation of concerns and ensures efficient communication between components. Client-server separation enhances security by isolating sensitive operations and credentials on the server side, while host offers standardized interfaces for end users.

MCP Role	System Component	Description
Host	Web browser / desktop app	User-facing interface; initiates requests
Client	`client_MPC.py`	Flask-based API; maintains 1:1 connection with the server
Server	`server_MPC.py`	Handles RAG, file management, LLM integration
Local Data	`uploads/` directory	Stores user-uploaded files and server-managed databases
Remote Service	External LLM APIs	Accessed by the client for remote inference when configured

Table 1: Mapping of MCP Roles to System Components

2.2 JSON communication protocol

In the implemented system, communication between the client and server is achieved using a simple, custom JSON-based protocol over TCP sockets. TCP (Transmission Control Protocol) provides reliable, ordered, and error-checked delivery of a stream of octets (bytes) between applications running on hosts communicating via an IP network [32]. Each request from the client is structured as a JSON object containing a `"command"` key, along with any additional parameters required for the specific operation.

Requests use a JSON object with a `"command"` key and parameters, for example:

```
{ "command": "ask", "question": "What is in document X?" }
```

The server processes the request and returns a response as a JSON object containing a `"status"` key to indicate the outcome, and other relevant data as needed.

Responses use a JSON object with a `"status"` key, for example:

```
{ "status": "success", "answer": "Document X contains..." }
```

The Model Context Protocol (MCP) commonly uses JSON-RPC as its standard communication protocol, which defines a formal request-response structure with fields such as `"method"`, `"params"`, and `"id"`. This structure enables structured, extensible, and reliable interactions between clients and servers [31].

In our case, this kind of protocol is not implemented. We do not use JSON-RPC primarily because our communication needs are simpler and better served by a custom, lightweight JSON message format tailored to specific commands and responses. Instead of the full JSON-RPC specification, this protocol uses straightforward JSON objects containing a `"command"` key for requests and a `"status"` key for responses.

This choice provides several practical benefits:

- **Simplicity and Minimal Overhead:** The protocol is easy to implement and debug, using direct command-response pairs without the complexity of managing request IDs or batch calls required by JSON-RPC.
- **Extensibility:** New commands and parameters can be added flexibly without altering the core communication mechanism, allowing the protocol to evolve with project needs.
- **Language Agnosticism:** JSON is supported by virtually all programming languages, enabling interoperability across diverse systems.

- **Structured Data Support:** JSON effectively represents complex and nested data structures, facilitating robust client-server interactions.
- **Tailored to Specific Use Cases:** The protocol focuses on a limited set of commands and responses, avoiding unnecessary features that would remain unused.
- **Flexibility in Transport and Security:** The protocol is designed to work efficiently over TCP sockets and can be adapted to specific security models without the constraints of JSON-RPC.

The decision to use a simple, custom JSON messaging scheme instead of JSON-RPC reflects a deliberate trade-off favoring simplicity, directness, and alignment with the system's specific communication patterns over the generality and formalism of JSON-RPC. This approach reduces development overhead while maintaining clear, structured, and extensible messaging between client and server.

2.3 Component responsibilities: client, server, frontend

System's functionality is distributed across three primary components, each with distinct responsibilities that contribute to the overall operation of the RAG chatbot. The following sections will detail the roles and characteristics of the user-facing frontend, the intermediary client, and the core server component, as described in Figure 2.

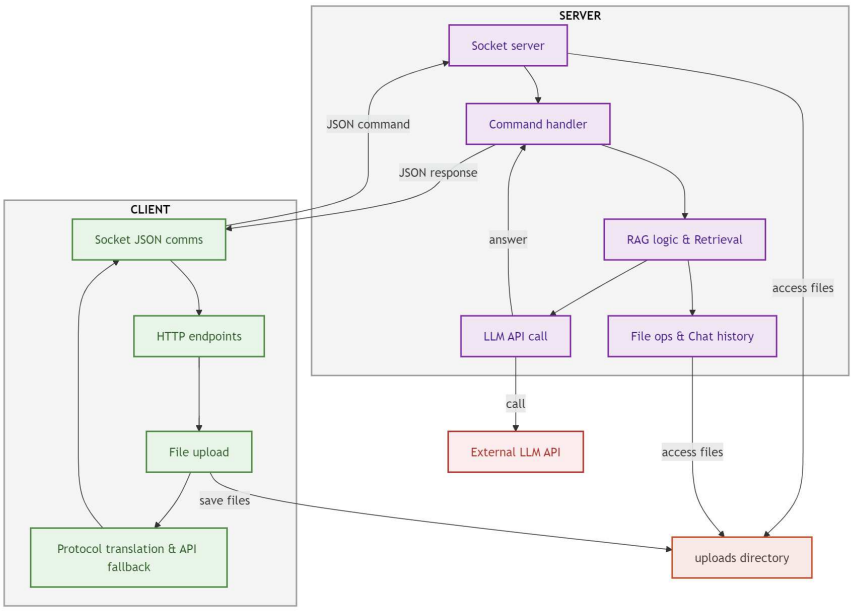

Figure 2: Client-Server Communication Flow

2.3.1 Client Component (`client_MPC.py`)

The client component serves as an intermediary between the user interface (such as a web browser) and the backend server. It is implemented using the Flask micro web framework, Flask is a lightweight and flexible Python web framework that allows easy creation of routes to handle HTTP requests and responses, making it well-suited for web applications [33], which provides specific web addresses (URLs) that the program uses to interact with a server or a service, called HTTP API endpoints (e.g., `/mpc_ask`, `/mpc_upload`). When a user submits a request or uploads a file via the web interface, the client component processes the input, performs preliminary validation and access control, and serializes the command using a standardized protocol.

Communication between the client and the server is established via the standard Python socket library. Python code leverages this socket module to create network connections and exchange information between different parts of the system or with external services [29, 34]. As explained in Section 2.2, this enables direct TCP socket connections for efficient, bidirectional data transfer. This approach allows the client to forward user commands and file metadata to the server in real time. Additionally, the

client supports flexible routing: when a remote LLM API is configured, it can forward requests externally; otherwise, it defaults to the local server instance. This design supports both local and distributed deployments.

2.3.2 Server Component (`server_MPC.py`)

The server component is the core of the system, responsible for all sensitive operations, document management, retrieval-augmented generation (RAG) workflows [35], and secure integration with external large language model (LLM) APIs [36]. Its design centralizes all logic and credential handling, ensuring both robustness and security.

Upon receiving commands from the client, the server validates and processes uploaded files, supporting multiple formats such as PDF, CSV, and JSON. It uses specialized loaders for each file type and sanitizes filenames to prevent security risks. Documents are split into manageable chunks, which are then indexed using two parallel strategies: semantic search (using transformer-based embeddings with GPU acceleration if available, and persistent caching to disk) and keyword search (using BM25). These retrieval methods are combined in an ensemble retriever, allowing the server to maximize both recall and precision when searching for relevant information. These aspects of the server's operation, including the details of the RAG workflow, hybrid retrieval strategies, and integration with external LLM APIs, will be explained in greater depth in Chapter 2.

When a user submits a query, the server retrieves the most relevant document chunks using this hybrid retriever. It constructs a detailed prompt that includes the chat history, the retrieved context, and the user's question. This prompt is sent to an external LLM API (such as Google Gemini) via a secure HTTP request, with all credentials managed exclusively on the server side. The server parses the LLM's response and returns it to the client, updating the chat history for conversational continuity.

All command handling, file operations, retrieval logic, and LLM interactions are performed exclusively on the server, ensuring that sensitive data and API keys remain protected [36]. The server is designed for concurrent operation, handling multiple client connections via threading, and includes extensive logging and error handling for maintainability and debugging.

In summary, the server component encapsulates all core logic for document ingestion, hybrid retrieval, RAG orchestration, and secure LLM integration. By isolating sensitive operations and credentials, it provides a robust and maintainable foundation for the entire system, enabling secure, conversational access to local document collections without exposing data to external parties.

2.3.3 Front End Component

The front end component provides the user-facing interface for interacting with the RAG chatbot system. It is implemented as a web application using standard web technologies: HTML, CSS, and JavaScript [37]. The main HTML file defines the structure of the interface, including areas for file upload, document management, chat display, and user input. The design emphasizes usability and clarity, allowing users to easily upload documents, manage files, and conduct conversational queries.

The JavaScript logic (`mpc_script.js`) manages all dynamic interactions on the page. It handles file selection and uploads, displays system messages, updates the list of uploaded documents, and manages the chat flow between the user and the assistant. When a user submits a question or uploads a file, the script sends asynchronous HTTP requests to the client's API endpoints, processes responses, and updates the interface accordingly. Features such as chat history download, real-time status updates, and file management actions (delete, update, view content) are also supported.

Styling is provided by a dedicated CSS file (`mpc_styles.css`), which ensures a modern, dark-themed appearance with clear visual separation between user messages, assistant responses, and system notifications. The layout is responsive and user-friendly, with interactive elements such as buttons, file selectors, and loading indicators. Visual feedback is provided for actions like file uploads, chat processing, and system resets, enhancing the overall user experience.

In summary, the front end component enables intuitive, interactive access to the system's capabilities, allowing users to seamlessly upload documents, manage files, and engage in conversational queries with the RAG chatbot. It acts as the primary point of interaction, bridging the gap between the user and the underlying client-server architecture.

2.3.4 Rationale for Client-Server Separation

The adoption of a client-server architecture offers several key advantages:

- **Protocol Bridging:** The client translates web-based user interactions into a protocol understood by the server, facilitating interoperability and modularity.
- **Deployment Flexibility:** The client can route requests to either a remote LLM API or a local server instance, supporting a range of deployment scenarios.
- **Separation of Concerns:** The user interface focuses solely on user experience, the client manages API logic and protocol translation, and the server executes all core processing and retrieval logic.
- **Security and Validation:** The client enforces basic validation and access control, while the server restricts sensitive operations to a secure backend environment.

2.4 Supported Commands

The Server as described in Section 2.3.2, exposes a set of structured commands that define the interaction between the client and the server. Each command, based on the procedures explained in Section 2.2, is sent as a JSON object with a `command` key and relevant parameters. The server processes these commands and returns a standardized JSON response, ensuring robust, extensible, and predictable communication.

Table 2 provides a quick overview of the commands available on the server.

Command	Description	Response Keys (on success)
`get_chat_history`	Retrieve chat history	`status`, `chat_history`
`ask`	Ask a question	`status`, `answer`
`upload`	Upload files & init retriever	`status`, `message`, `files`
`get_files`	List uploaded files	`status`, `files`
`stop_generation`	Stop generation process	`status`, `message`
`delete_file`	Remove file & update retriever	`status`, `message`, `files`
`reset`	Clear all files & retriever	`status`, `message`, `files`
`get_status`	Return retriever status	`status`, `num_files`, `files`, `retriever_initialized`
`update_file`	Update file & retriever	`status`, `message`, `files`
`get_document_content`	Get content of a specific file	`status`, `filename`, `content`
`get_supported_file_types`	List allowed file types	`status`, `supported_file_types`

Table 2: Supported Server Commands

Below is a summary of each supported command, as implemented in the server:

- `get_chat_history`: Returns the full chat history for the current session as a list of (question, answer) pairs. This enables the client to display previous interactions and maintain conversational context.
- `ask`: Submits a user question to the system. The server retrieves relevant document chunks using the hybrid retriever, constructs a prompt including chat history and context, and calls the external LLM API to generate an answer. The response includes the answer and updates the chat history.
- `upload`: Uploads one or more files to the server and initializes the retriever. The server validates file types and sizes, processes the files, and updates the internal list of uploaded files. On success, the retriever is ready for question answering over the new documents.
- `get_files`: Returns a list of all files currently uploaded and available for retrieval. This allows the client to display or manage the document set.
- `stop_generation`: Sets a flag to interrupt any ongoing response generation process. This provides users with control over long-running LLM operations.

- `delete_file`: Removes a specified file from the server and updates the retriever index. If the file exists, it is deleted and the retriever is re-initialized with the remaining files.
- `reset`: Deletes all uploaded files and clears the retriever, returning the system to its initial state. This is useful for starting a new session or clearing all data.
- `get_status`: Returns the current status of the retriever, including the number of files, their names, and whether the QA chain is initialized. This helps monitor system readiness.
- `update_file`: Replaces an existing file with a new version and re-initializes the retriever. If the file does not exist, it is added to the uploaded files list.
- `get_document_content`: Retrieves the full content of a specific file, supporting transparency and user review. The server reads and returns the file content based on its type (PDF, CSV, or JSON).
- `get_supported_file_types`: Returns a list of all file types accepted by the system (PDF, CSV, JSON), guiding users in preparing compatible documents.

Each command is designed for clarity, security, and extensibility, ensuring that both user-facing interfaces and automated clients can interact with the system efficiently and safely.

2.5 Security Considerations and Credential Management

A critical aspect of the system's security is the management of sensitive credentials, such as API keys for external LLM services (e.g., Google Gemini) and HuggingFace tokens. To prevent accidental leaks, all secrets are stored in a dedicated `.env` file, which is loaded at runtime using the `python-dotenv` package. This file contains variables like `EXTERNAL_LLM_API_URL`, `EXTERNAL_LLM_API_KEY`, and `HF_TOKEN`, ensuring that credentials are never exposed in the codebase or version control. Isolating credentials in the `.env` file supports secure deployment across different environments and aligns with best practices for secure software development [36].

The client-server architecture further strengthens security by enforcing a strict separation of responsibilities. The client component is limited to collecting user input, performing basic validation, and forwarding requests to the server. It never has access to sensitive credentials, document content, or internal logic. All critical operations—such as file validation, document parsing, hybrid retrieval, and communication with external LLM APIs—are performed exclusively on the server. This ensures that API keys, tokens, and private data remain protected within a controlled environment, never being exposed to the client, browser, or network beyond the server boundary.

Even if the client or frontend is compromised, attackers cannot access secrets or manipulate core logic, as these are isolated on the server. The server also enforces strict validation and logging for all incoming commands, reducing the risk of injection attacks or unauthorized access. By centralizing sensitive logic and credential management, the system minimizes its attack surface and simplifies compliance with data protection standards, as sensitive data never leaves the secure server environment [**?**].

3 RAG Implementation and Features

This chapter explores the main concepts and implementation details of Retrieval-Augmented Generation (RAG) within the chatbot system. As the core mechanism of the entire project, RAG underpins the system's ability to provide accurate, context-aware responses. This chapter covers the foundational RAG architecture, integration with LangChain and HuggingFace, the hybrid retrieval strategies employed, and also addresses system enhancements such as GPU acceleration, multi-format document support, and robust API integration.

3.1 Retrieval-Augmented Generation (RAG) principles and architecture

Why Retrieval-Augmented Generation (RAG)?

RAG was chosen because it overcomes the main drawbacks of standalone large language models. By first retrieving relevant passages from the user's own documents, it grounds its output in real, up-to-date information and dramatically reduces "hallucinations" [38]. It also sidesteps the fixed context window of any single LLM, since new documents can simply be indexed rather than forcing ever-larger models [38].

According to recent research [35, 38], practitioners adopt RAG when they need:

- **Factual accuracy and traceability** – every answer can be traced back to its source snippets.
- **Flexibility** – new data sources (PDF, CSV, JSON, etc.) can be ingested without retraining the model.
- **Efficiency** – generating on a narrowed, relevant context lowers latency and API costs.
- **Privacy and control** – sensitive or proprietary documents never leave the local environment; all retrieval and prompt assembly happens server-side.

These benefits make RAG the de-facto approach for real-world, document-centric chatbots and question-answering systems.

At its heart the system uses a three-stage RAG pipeline—**Retrieval**, **Augmentation**, and **Generation**—to turn a user's natural-language query into a grounded, context-aware answer.

- **Retrieval:** Retrieval is responsible for identifying the most relevant information within a vast document corpus. We employ a hybrid approach:
  - Semantic search: transformer-based embeddings (e.g., BAAI/bge-base-en-v1.5) capture conceptual similarity, allowing the model to surface passages that convey the same meaning even without exact keyword matches. These embeddings are computed once per document chunk and stored in a vector index for fast nearest-neighbor queries.
  - BM25 keyword search: a complementary BM25 keyword index provides precise lexical matching for technical terms, acronyms, or rare identifiers that embeddings may underrepresent.

  By ranking results from both methods and combining them with tunable weights, the retriever ensures high recall while maintaining precision.
- **Augmentation:** Augmentation merges retrieved snippets with the user's query and conversation history to construct a coherent, context-rich prompt.
  - Context consolidation: Selected passages are concatenated in order of relevance, ensuring the most critical information appears early in the prompt. When snippet lengths exceed token limits, we apply a sliding window or summarization to preserve key facts.
  - History integration: Prior questions and answers are interleaved with new context, allowing the LLM to maintain conversational continuity and avoid redundant queries.
  - Instruction injection: System messages or persona guidelines frame the LLM's behavior, enforcing tone, style, or output format (e.g., bullet points vs. free text).

  By carefully orchestrating these elements, augmentation provides the LLM with just enough context to ground its reply without overwhelming it with irrelevant data.
- **Generation:** Generation uses the assembled prompt to produce a final answer via an external LLM API (such as Google Gemini).

– Parameter tuning: Temperature, maximum tokens, and stop sequences are calibrated to balance creativity and factuality, avoiding overly verbose or off-topic output.
– Source attribution: The model's response is post-processed to append citations or snippet references when possible, enhancing traceability back to the original documents.
– Fallback handling: If the primary API call fails, the system can throttle requests or switch to an alternate endpoint, ensuring robustness.

This stage leverages the LLM's natural language capabilities while anchoring its output in concrete, verifiable evidence.

By decoupling retrieval from generation and providing an automatic HTTP-first/socket-fallback mechanism in `client_MPC.py`, the architecture ensures both robustness (queries are never lost if one endpoint is down) and security (only the server handles document embeddings, credentialed API keys, and private data).

3.2 Integration of LangChain and HuggingFace for RAG pipeline

The core of the system's Retrieval-Augmented Generation (RAG) pipeline, the principles of which are detailed in Section 2.1, is implemented using the LangChain framework, with HuggingFace Transformers providing embedding models. This integration enables document retrieval and question answering over diverse data sources. LangChain is a modular framework designed to build applications powered by large language models (LLMs) [39]. It offers components such as chains and retrievers to create workflows, connecting with various LLMs and data sources to affect response accuracy. Its design makes it a choice for building document-centric AI systems.

Embeddings and Caching:
The server initializes a HuggingFace embedding model (specifically, `BAAI/bge-base-en-v1.5`) to convert document chunks into dense vector representations. The device is automatically set to GPU (CUDA) if available, or CPU otherwise, for performance (GPU acceleration is discussed further in Section 2.4.1). To accelerate repeated queries and avoid redundant computation, embeddings are wrapped with LangChain's `CacheBackedEmbeddings`, which persistently store computed vectors using a local file store. This design reduces latency for large or frequently accessed document sets.

Hybrid Retrieval with Ensemble Retriever:
Document retrieval is performed using a hybrid approach (elaborated in Section 2.3) that combines semantic and keyword-based search. Documents are first loaded (multi-format support and loading mechanisms are described in Section 2.5) and split into chunks using LangChain's text splitters. Two parallel retrievers are then constructed:

- **Semantic Retriever:** Utilizes the cached HuggingFace embeddings and a vector store (`DocArrayInMemorySearch`) to find semantically similar chunks.
- **BM25 Keyword Retriever:** Employs BM25 ranking to match specific terms, acronyms, or identifiers, which may not always be prioritized by embeddings alone.

LangChain's `EnsembleRetriever` combines the results from both methods, with tunable weights, aiming to balance recall and precision. This ensures that both conceptual and lexical information are considered when retrieving relevant information.

Document Loading and Multi-format Support:
The system supports PDF, CSV, and JSON files. Each file type is loaded using the LangChain community loader or custom logic (as detailed in Section 2.5), and all files are sanitized and validated before processing. This modular approach allows for extension to additional formats in the future.

Conversational Retrieval and Prompt Construction:
For each user query, the server retrieves relevant document chunks using the ensemble retriever (see Section 2.3 for the hybrid strategy). It then constructs a prompt that includes the current question, retrieved context, and formatted chat history (the prompt construction process is detailed in Section 2.6.1).

Scalability and Robustness:
The server is designed for concurrent operation, handling multiple client connections via threading. It includes error handling, logging, and validation at various stages—from file upload and document parsing

to retrieval and LLM interaction. The architecture also supports large file handling and can be extended to asynchronous processing for scalability.

Analysis of Libraries Used in the RAG Pipeline Implementation:
The RAG pipeline implementation in `server_MPC.py` leverages a combination of open-source libraries, each chosen for its capabilities in retrieval-augmented generation, document processing, and server operation. Key components include:

- **LangChain Core and Community Modules:**
  - `langchain_core.documents` and `langchain_core.prompts` provide representations for documents and prompts, supporting modularity and extensibility.
  - `langchain.chains.ConversationalRetrievalChain` supports conversational workflows, maintaining chat history and context.
  - `langchain_community.document_loaders` supplies loaders for PDF, CSV, and JSON files, for ingestion of diverse data types.
  - `langchain.text_splitter.RecursiveCharacterTextSplitter` enables chunking of documents, used for retrieval.
  - `langchain_community.vectorstores.DocArrayInMemorySearch` offers an in-memory vector store for semantic search over embeddings.
- **HuggingFace and Embedding Utilities:**
  - `langchain_huggingface.HuggingFaceEmbeddings` integrates transformer models (like `BAAI/bge-base-en-v1.5`) for semantic embeddings with automatic GPU/CPU selection.
  - `langchain.embeddings.CacheBackedEmbeddings` and `langchain.storage.LocalFileStore` provide persistent caching to reduce computation and affect response times.
- **Hybrid Retrieval and Ranking:**
  - `langchain.retrievers.BM25Retriever` implements BM25 keyword-based retrieval for matching technical terms and identifiers.
  - `langchain.retrievers.EnsembleRetriever` combines semantic and keyword retrievers with tunable weights to balance recall and precision.
- **Data Validation and Security:**
  - `jsonschema` validates incoming data structures, contributing to robustness and preventing malformed input errors.
  - `dotenv` loads environment variables from a `.env` file, managing sensitive credentials like API keys and tokens.
- **Standard Python Libraries:**
  - Libraries such as `os`, `logging`, `threading`, `socket`, `time`, `sys`, `json`, and `pandas` provide support for file management, logging, concurrency, network communication, and data processing.

Together, these libraries form the foundation for the RAG pipeline, supporting retrieval, document handling, and integration with external LLM APIs.

3.3 Hybrid search strategy: semantic embeddings and BM25 keyword search

To optimize the relevance and comprehensiveness of retrieved context for the Large Language Model (LLM), the system employs a hybrid search strategy. As introduced in Section 2.1 regarding RAG principles, this approach combines the strengths of semantic (vector-based) search with traditional keyword-based search (BM25) [40, 41]. While the specific LangChain components implementing this strategy—such as the `HuggingFaceEmbeddings`, `DocArrayInMemorySearch`, `BM25Retriever`, and `EnsembleRetriever`—were detailed in Section 2.2 covering LangChain and HuggingFace integration, this section delves deeper into the rationale and mechanics of this hybrid methodology as implemented in `server_MPC.py`.

At the core of semantic search are embeddings, which are numerical representations of text (words, sentences, or entire documents) in a high-dimensional vector space. These vectors are generated by deep learning models, often transformers, trained on vast amounts of text data [41]. The key property of these embeddings is that semantically similar pieces of text are mapped to nearby points in the vector space. This allows the system to measure similarity by calculating the distance (e.g., cosine similarity or Euclidean distance) between query embeddings and document chunk embeddings, enabling the retrieval of conceptually related content even without exact keyword overlap.

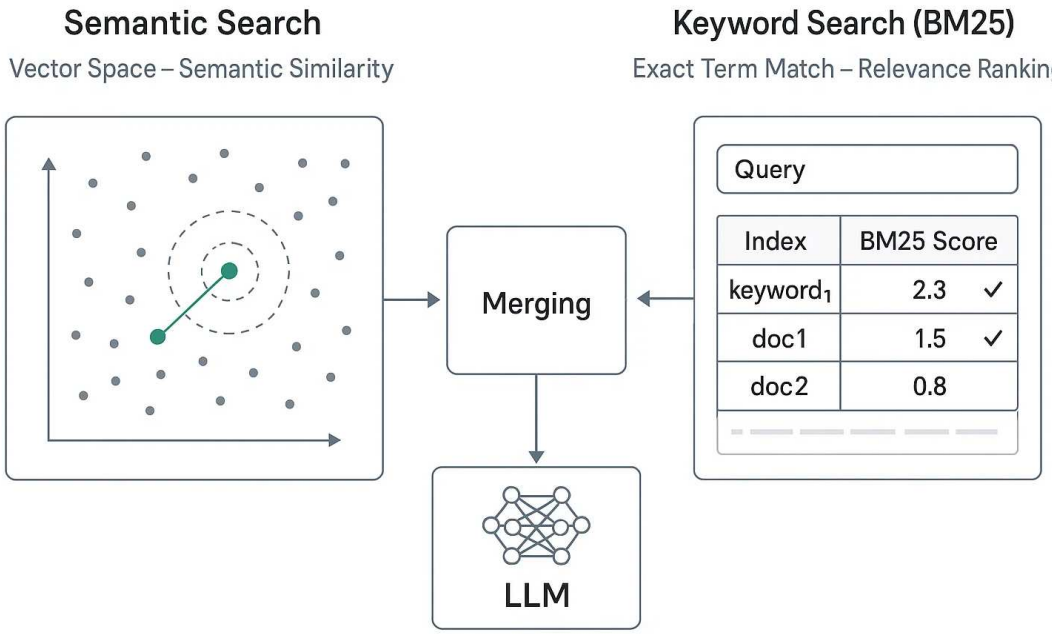

Figure 3: Hybrid Retrieval Dynamics: Semantic and Keyword Search Combination.

Figure 3 illustrates the hybrid retrieval process.

The motivation for a hybrid approach stems from the complementary nature of these two search paradigms. Semantic search, powered by the `BAAI/bge-base-en-v1.5` embedding model in this system (detailed in Section 2.2), is designed to understand the conceptual meaning and intent behind a user's query. It can identify relevant passages even if they do not contain exact keyword matches, handling synonyms, paraphrasing, and nuanced language. However, it may sometimes generalize over or miss specific technical terms, acronyms, or unique identifiers that are pertinent to a query [40].

Conversely, BM25 (Best Match 25) keyword search provides precise lexical matching. It operates on principles similar to TF-IDF (Term Frequency-Inverse Document Frequency), considering term frequency within a document and its rarity across the entire corpus, along with document length normalization. This makes BM25 effective for queries where exact keywords, product codes, or specific jargon are paramount. Its limitation lies in its lack of understanding of semantic relationships; it cannot find conceptually similar content if the exact keywords are absent.

The system, as implemented in the `load_db` function within `server_MPC.py`, leverages LangChain's `EnsembleRetriever` to fuse the outputs of these two distinct retrieval mechanisms. Both the semantic retriever (built upon `DocArrayInMemorySearch` with cached embeddings, as discussed in Section 2.2) and the `BM25Retriever` are configured to retrieve a top-k (k=5 in the current implementation) set of documents based on their respective scoring methods from the processed document chunks. The `EnsembleRetriever` then combines these two sets of results, applying a weighting scheme to balance their contributions. While the default configuration gives a preference to semantic relevance (e.g., weights of 0.6 for the semantic retriever and 0.4 for BM25), it is important to note that the optimal hybrid weighting is not fixed a priori. Instead, the most effective combination of semantic and keyword-based retrieval is determined empirically through systematic evaluation across multiple configurations, as detailed in Section 4. This evaluation-driven approach ensures that the final system configuration is tailored to maximize retrieval performance for the specific characteristics of the target document collection and

query types.

This weighted combination aims to produce a final list of retrieved documents that improves upon what either method could achieve in isolation. By integrating both conceptual understanding and lexical precision, the hybrid strategy enhances overall retrieval quality, leading to:

- **Improved Recall:** Increasing the likelihood of finding all relevant documents, as one method can compensate for the other's misses.
- **Enhanced Precision:** Ensuring that the retrieved documents are highly relevant, as both semantic context and specific keywords are considered.
- **Robustness to Query Types:** Performing well on a wider variety of queries, from broad conceptual questions to highly specific keyword-driven searches.

However, effectiveness and performance of this hybrid retrieval strategy will be quantitatively evaluated in Chapter 3.

3.4 GPU acceleration and large file handling

The practical utility of a Retrieval-Augmented Generation (RAG) system, such as the one developed in this project, is fundamentally linked to its capacity to efficiently manage and process a substantial volume of documents. The core value proposition of RAG is to save users significant time by enabling them to query and synthesize information from extensive document collections without manual review. If the system were limited to handling only a few documents, its purpose as a retrieval and knowledge discovery tool would be severely diminished. Therefore, robust mechanisms for handling numerous and potentially large files, alongside performance optimizations like GPU acceleration, are critical to the system's overall effectiveness and relevance.

Beyond the core retrieval strategies, the system incorporates features to enhance performance and usability, particularly concerning computationally intensive tasks and the management of substantial data volumes. These include GPU acceleration for embedding generation and specific mechanisms for handling large file uploads and data transmission.

3.4.1 GPU Acceleration for Embeddings

The generation of semantic embeddings, a foundational step for the retrieval mechanism detailed in Section 2.2 and Section 2.3, can be computationally demanding, especially with large vocabularies or extensive document sets. To mitigate this, the system, as implemented in `server_MPC.py`, automatically leverages available GPU resources. It checks for CUDA compatibility using `torch.cuda.is_available()` [42]. CUDA (Compute Unified Device Architecture) is a parallel computing platform and application programming interface (API) model created by NVIDIA, which allows software developers to use a CUDA-enabled graphics processing unit (GPU) for general purpose processing. If a CUDA-enabled GPU is detected, the `HuggingFaceEmbeddings` (utilizing the `BAAI/bge-base-en-v1.5` model as discussed in Section 2.2) is configured to run on the GPU by setting `model_kwargs={'device': 'cuda'}`. This offloading significantly accelerates the embedding process, leading to faster initialization of the retrieval system upon document upload or update, and quicker query embedding for semantic search. Effective GPU utilization requires the host system to have compatible NVIDIA drivers and a PyTorch installation with CUDA support.

3.4.2 Mechanisms for Large File and Data Handling

The system is designed to manage user-uploaded documents and data transfers of varying sizes. Several mechanisms across both client and server components address the challenges posed by large data volumes:

- **Client-Side Upload Limits (Flask):** For files uploaded via the web interface, the Flask client (`client_MPC.py`) is configured with `app.config['MAX_CONTENT_LENGTH'] = 500 * 1024 * 1024`. This setting defines a 500MB limit for the total size of an HTTP upload request, allowing the client application to receive substantial files from the user's browser [30, 33].
- **Server-Side File Size Limits:** The server (`server_MPC.py`) defines a `MAX_FILE_SIZE` constant (e.g., 100MB per file) to cap the size of individual files it will process. This acts as a safeguard against resource exhaustion from excessively large single uploads.

- **Chunked Data Transmission (Sockets):** When the client (`client_MPC.py`) communicates with the server via sockets (as detailed in Section 1.2 regarding the communication protocol), particularly in the `send_command_to_server` function, commands (which are JSON objects and can be large if they contain extensive metadata or parameters) are sent in chunks (e.g., 1024 bytes at a time). This prevents overwhelming socket buffers and ensures reliable transmission of potentially large command payload [34].
- **Socket Timeouts:** Both client and server socket operations incorporate timeouts. For instance, in `client_MPC.py`, `s.settimeout(60)` is used for the socket connection to the server. This increased timeout (matching the server's expected processing time for some operations) prevents premature disconnections during lengthy operations like the processing and embedding of large documents on the server side.
- **System Resources:** Handling large files inherently demands more system resources, particularly RAM for holding document content during processing and CPU for parsing and splitting. Sufficient resources on the server machine are critical for stable operation.

While the current implementation processes files synchronously upon upload, future enhancements could explore asynchronous processing to improve responsiveness when dealing with very large documents or batches of files. The existing document chunking strategy (Section 2.2) is also vital, as it breaks down large documents into smaller, manageable pieces for the embedding and retrieval stages.

These mechanisms for GPU acceleration and considerations for large file and data handling contribute to a more scalable and efficient RAG system, capable of delivering timely responses even when working with extensive document corpora and complex commands.

3.5 Document loaders and multi-format support (PDF, CSV, JSON)

A key requirement for a versatile Retrieval-Augmented Generation (RAG) system is its ability to ingest and process information from diverse document formats. The developed system addresses this by providing explicit support for PDF, CSV, and JSON files, which are common formats for storing textual and structured data. This multi-format capability is primarily achieved through the integration of specialized document loaders from the LangChain community library, as introduced in Section 2.2.

The server component (`server_MPC.py`) implements a dispatch mechanism within its file processing logic (specifically in the `load_document_based_on_type` function) to handle each supported file type. Before attempting to load a document, the server validates the file extension against a predefined set of `ALLOWED_EXTENSIONS` ({`'pdf'`, `'csv'`, `'json'`}), ensuring that only supported formats are processed.

- **PDF (`.pdf`) Files:** For PDF documents, the system utilizes `PyPDFLoader` from `langchain_community.document_loaders`. This loader is capable of extracting text content from PDF files, page by page. The extracted text from each page is then treated as a separate document or further processed by text splitters (as discussed in Section 2.2) to create manageable chunks for embedding and retrieval.
- **CSV (`.csv`) Files:** Comma-Separated Values files are handled using `CSVLoader`. This loader typically treats each row in the CSV file as a distinct document or record. The content of the cells in a row is concatenated or structured to form the text that will be indexed. This is particularly useful for querying structured data where each row represents an item or entry.
- **JSON (`.json`) Files:** JavaScript Object Notation files are processed using `JSONLoader`. Due to the flexible and often nested nature of JSON data, this loader requires a `jq_schema` to specify which parts of the JSON structure contain the relevant text content. In the current implementation (`server_MPC.py`), a default schema of `'.'` is used, which attempts to extract all text content. For more complex JSON structures, this schema can be customized to target specific fields or arrays (e.g., `'.messages[].content'`). The `text_content=True` parameter ensures that the extracted data is treated as plain text.

Once a document is loaded using the appropriate LangChain loader, its content is passed to the subsequent stages of the RAG pipeline, including text splitting, embedding generation, and indexing by the hybrid retriever (semantic and BM25), as detailed in Section 2.2 and Section 2.3. This modular approach to document loading not only supports the current set of formats but also provides a clear path for extending support to other file types in the future by integrating additional LangChain loaders

or custom parsing logic. The ability to seamlessly process these varied formats significantly enhances the system's utility, allowing users to build a comprehensive knowledge base from their existing documents.

3.6 API fallback mechanisms and external LLM integration

The system employs API fallback mechanisms to ensure continuous operation, automatically switching to an alternative communication pathway if the primary external API endpoint becomes unavailable. The system's ability to generate coherent and contextually relevant responses hinges on its integration with a powerful external Large Language Model (LLM). This integration, coupled with robust communication protocols and these fallback mechanisms, ensures both functionality and resilience. This section details how the server component (`server_MPC.py`) interacts with the external LLM and how the client (`client_MPC.py`) implements a fallback strategy for command routing [43].

3.6.1 Server-Side External LLM Integration

The core generation step of the RAG pipeline (as outlined in Section 2.1) is handled by the `server_MPC.py` through its `call_external_llm_api` function. This function is responsible for communicating with a designated external LLM API, such as Google's Gemini API in the current implementation.

A central aspect of this integration is the **Dynamic Prompt Construction**. The server meticulously crafts a detailed prompt for the LLM. This prompt is a critical element, as its specific content and structure, defined in the `get_prompt_template` function within `server_MPC.py`, significantly guide the model's responses and can be adapted for different LLMs. The prompt dynamically incorporates the user's current question, the retrieved context from the hybrid search mechanism (detailed in Section 2.3), and the formatted chat history to maintain conversational coherence. The base template used, sourced directly from the `get_prompt_template` function in `server_MPC.py`, is shown below:

Listing 1: Base Prompt Template from `server_MPC.py`

```
You are a helpful assistant. Based on the chat history and the context provided
    , answer the user's current question in a clear and natural way.
Summarize the key information relevant to the question without using bullet
    points unless necessary for clarity.
If you don't know the answer or the context doesn't contain the information,
    just say so.
Try to sound helpful and conversational.

Chat History:
{chat_history}

Context:
{context}

Current Question: {question}

Answer:
```

Other key aspects of the server-side LLM integration include:

- **Secure Configuration:** API credentials and endpoint details (specifically `EXTERNAL_LLM_API_URL` and `EXTERNAL_LLM_API_KEY`) are managed securely using environment variables, loaded via the `dotenv` library (as noted in Section 2.2.1). This practice prevents hardcoding sensitive information and allows for flexible configuration across different deployment environments.
- **Controlled Generation:** The API call includes `generationConfig` parameters such as `temperature`, `topP`, `topK`, and `maxOutputTokens`. These parameters, as implemented in `server_MPC.py`, allow for tuning the LLM's output to balance creativity with factuality and to control response length, crucial aspects for a reliable assistant.
- **API Error Handling:** The `call_external_llm_api` function includes logic to handle various HTTP response statuses from the LLM API. It specifically checks for successful responses (HTTP 200), authentication failures (HTTP 401), and other potential API errors, providing informative logging and returning appropriate error messages to the client. This ensures that API issues are gracefully managed and reported.

3.6.2 Client-Side API Fallback Mechanism

To enhance the system's robustness and flexibility in different network environments, the `client_MPC.py` implements a two-tiered approach for routing commands, primarily within its `send_command_to_server` function:

- **Primary Target (External API):** The client first attempts to send commands to a configured external API endpoint (specified by `API_ENDPOINT_URL` and authenticated with `API_KEY` from its `.env` file). This allows the system to potentially leverage a managed API gateway or a cloud-hosted version of the server logic.
- **Fallback (Local Server):** If the call to the external API fails (due to network issues, endpoint unavailability, or other errors), the client automatically falls back to communicating with the local server instance (`server_MPC.py`) directly via a TCP socket connection. The host and port for this local connection (`LOCAL_SERVER_HOST`, `LOCAL_SERVER_PORT`) are also configurable via environment variables or default to `127.0.0.1:65432`.

This fallback strategy ensures that if the primary external API is inaccessible, the user can still interact with the system through the local server, provided it is running. The communication over sockets utilizes the JSON-based protocol detailed in Section 1.2, including chunked data transmission for handling potentially large command payloads, as discussed in Section 2.4.2.

3.6.3 Synergy and Benefits

The combination of server-side external LLM integration and client-side API fallback provides several benefits:

- **Enhanced Robustness:** The client's ability to switch to a local server connection ensures continued operation even if an external API endpoint fails.
- **Secure Credential Management:** The server centralizes interaction with the LLM API, meaning the LLM API key is only required on the server, enhancing security. The client uses a separate API key if an intermediate API gateway is employed for the primary route.
- **Deployment Flexibility:** This architecture supports various deployment scenarios, from a fully local setup to one utilizing cloud-based API endpoints.

By decoupling the LLM interaction to the server and providing a resilient communication pathway from the client, the system aims for both reliable performance and secure operation.

4 Evaluation Methodology and Results

This chapter presents a comprehensive evaluation framework for assessing the retrieval performance of the RAG chatbot system. The evaluation employs three benchmark datasets: SQuAD for reading comprehension, Natural Questions for open-domain retrieval, and MS MARCO for web search scenarios, ensuring comprehensive coverage of real-world document-based question answering tasks [41].

Performance is measured using hybrid retrieval metrics including Recall@K, Mean Reciprocal Rank (MRR), exact match rates, and hit rates across systematic weight configurations [44]. The experimental methodology systematically evaluates hybrid ensemble retrieval across 10 different sparse/dense weight configurations (ranging from 10%/90% to 100%/0%), testing each configuration against three benchmark datasets to identify optimal retrieval strategies for different query types and domains.

4.1 Dataset Overview

The evaluation framework employs three established benchmark datasets, each selected to assess different aspects of retrieval performance:

Dataset	**Size**	**Query Type**	**Focus Area**
SQuAD v1.1	10,570 questions	Reading comprehension	Controlled contexts with guaranteed answers
MS MARCO v2.1	9,706 queries	Web search	Realistic patterns with sparse relevance
Natural Questions	3,610 questions	Open-domain retrieval	Real user queries with complex documents

Table 3: Dataset Overview for Evaluation Framework

4.1.1 SQuAD (Stanford Question Answering Dataset)

SQuAD v1.1 contains 107,785 question-answer pairs based on 536 Wikipedia articles [46]. The validation split containing 10,570 examples is utilized for evaluation. Each data point includes:

- **Context**: Wikipedia paragraphs (average length: 122 words)
- **Question**: Natural language questions (average length: 11 words)
- **Answer**: Exact answer spans with character-level positions within the source text
- **Answer_start**: Character position where the answer begins in the context

The dataset's controlled nature ensures every question has a definitive answer present in its associated context, making it suitable for evaluating retrieval precision when ground truth contexts are known. Question types include factual queries, definition requests, and inference-based questions requiring contextual understanding.

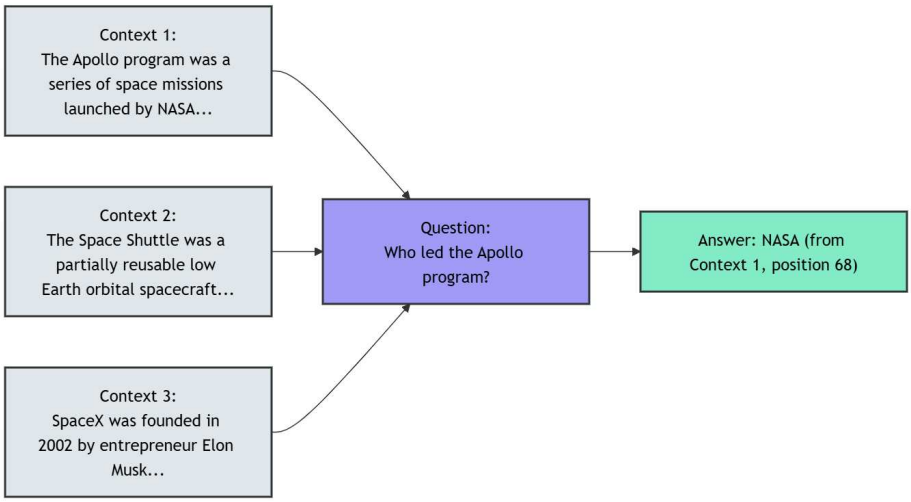

Figure 4: SQuAD Dataset Structure: Context-Question-Answer Flow

Key Challenges : The primary challenge lies in ensuring fair evaluation when all ground truth contexts are included in the searchable corpus, creating an artificially favorable environment where theoretical retrievability is 100%. The evaluation must distinguish between retrieval system performance and the inherent advantage of having all correct answers available. SQuAD's focus on reading comprehension means questions often require specific contextual understanding rather than broad knowledge retrieval, testing the system's ability to identify precise textual evidence rather than topical relevance. The Wikipedia-based contexts are generally well-structured and grammatically correct, which may not reflect quality variability in real-world document collections. The guaranteed answer presence differs from realistic scenarios where relevant information might be absent, potentially overestimating system performance. The extractive nature, where answers are exact spans from context, may not adequately test retrieval systems designed for more complex, abstractive question answering scenarios.

4.1.2 MS MARCO

MS MARCO v2.1 contains real Bing search queries with human-annotated relevant passages [48]. The structure focuses on passage-level retrieval:

- **Query**: Real Bing search queries (average length: 6 words)
- **Passages**: Text segments (50-300 words) with binary relevance judgments
- **Relevance patterns**: Typically 1-4 relevant passages per query from ∼1,000 candidates

The dataset reflects realistic search conditions with diverse query types (factual questions, definitions, procedural queries) and sparse relevance patterns where most query-passage pairs are non-relevant.

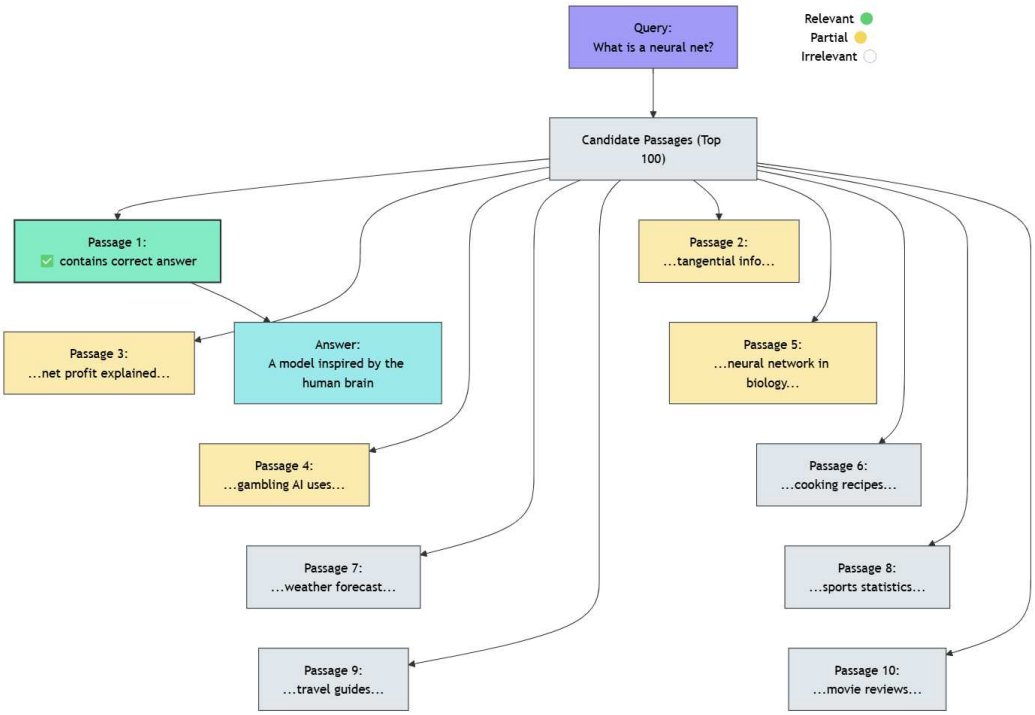

Figure 5: MS MARCO Dataset Structure: Query-Passage Retrieval with Relevance Classification

Key Challenges : MS MARCO poses unique challenges related to sparse relevance patterns and realistic search conditions. The primary challenge stems from its sparse relevance structure, where most query-passage pairs are non-relevant, closely mimicking real web search scenarios where relevant documents are rare. This creates evaluation difficulties in distinguishing between system failures and the natural scarcity of relevant content. The complex nested structure of passages within queries requires sophisticated processing to handle multiple passages per query while maintaining proper question-context pair associations. The need to process large numbers of candidate passages (approximately 1,000 per query) while removing duplicates for efficiency (typically reducing from 100,000+ examples to ∼99,000 unique contexts) presents significant computational and memory management challenges. The binary relevance judgments may not capture nuanced degrees of relevance that exist in real-world search scenarios, potentially oversimplifying evaluation of borderline cases where passages are partially relevant or contextually useful.

4.1.3 Natural Questions

The Natural Questions dataset comprises real, anonymized Google Search queries paired with corresponding Wikipedia articles [47]. The structure reflects real-world search scenarios:

- **Question**: Real user queries (average length: 9 words)
- **Document**: Complete Wikipedia articles with tokenized text (average: 4,896 tokens)
- **Annotations**: Multiple types including long answers (paragraph-level spans), short answers (entity-level spans), and yes/no responses

Approximately 80% of questions have long answers, with 50% also containing short answers nested within long answers. The remaining 20% have no answer in the provided document.

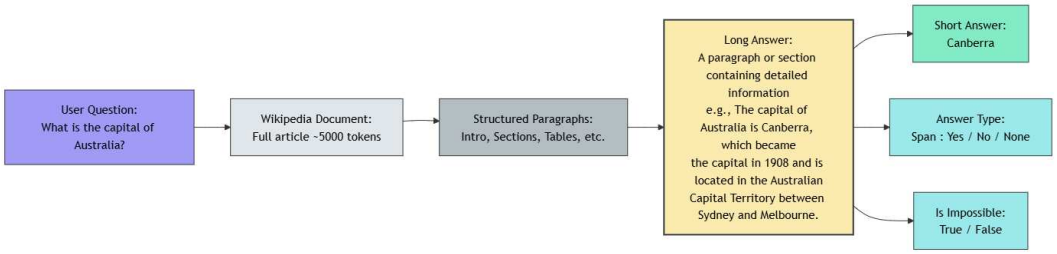

Figure 6: Natural Questions Dataset Structure: From Wikipedia Document to Answer Types

Key Challenges : Complex parsing challenges due to nested JSON structure and multiple answer types. The primary challenge involves reconstructing text from tokenized Wikipedia articles while handling both string and dictionary token formats. Answer extraction requires processing multiple annotation formats for short answers, long answers, and no-answer cases. The dataset's real-world nature introduces query ambiguity and variable document lengths, making it difficult to distinguish system limitations from inherent query complexity. The large dataset size (exceeding 40GB) necessitates streaming approaches to avoid memory limitations. Real user queries often contain colloquial language and context-dependent references that challenge retrieval evaluation methodologies.

4.1.4 Technical Implementation

Infrastructure and Configuration Hardware: NVIDIA GeForce RTX 4050 Laptop GPU with CUDA acceleration, Windows 11 Professional, Python 3.11.x environment with automatic GPU detection and CPU fallback.

Software Environment: HuggingFace Transformers 4.x, LangChain 0.1.x, PyTorch 2.x with BAAI/bge-base-en-v1.5 embeddings (768-dimensional vectors) and BM25 sparse retrieval.

Document Processing and Retrieval Implementation

Document Processing Pipeline : Document chunking employs `RecursiveCharacterTextSplitter` with optimized parameters (chunk sizes of 300-400 tokens and overlap of 30-50 tokens for context preservation). The system creates unique document collections ensuring all ground truth contexts are available for retrieval, implementing fair evaluation methodologies where theoretical retrievability is guaranteed. Text processing includes filtering of documents shorter than specified thresholds to ensure substantial content and robust progress tracking with `tqdm` progress bars.

Hybrid Retrieval Implementation : All evaluations employ ensemble retrieval with tunable weights, combining sparse keyword-based search with dense semantic search. The evaluation supports multiple retrieval methods simultaneously, enabling direct comparison between Dense (BAAI/bge-base-en-v1.5), Sparse (BM25), and Hybrid (ensemble) approaches within the same evaluation run.

Scalability and Performance : The evaluation scripts support configurable parameters including top-K values (default K=5), maximum examples for evaluation (scalable from 500 to 50,000+ examples), and fast mode options for quick testing versus complete statistical validation. Performance optimizations include checkpoint saving every 5,000 examples for long-running evaluations, comprehensive error handling and progress monitoring, and support for concurrent processing when system resources permit.

The three benchmark datasets selected for evaluating the RAG chatbot system's retrieval performance. SQuAD provides controlled reading comprehension scenarios with guaranteed answers, Natural Questions offers real-world open-domain queries with complex document structures, and MS MARCO represents realistic web search patterns with sparse relevance. Each dataset presents unique characteristics and challenges that together provide comprehensive coverage of document-based question answering scenarios. Next chapter will detail the specific evaluation metrics employed, including Recall@K, Mean Reciprocal Rank (MRR), and exact match rates, along with their application across different hybrid retrieval configurations.

4.2 Evaluation metrics

In this section we present the high-level categories of metrics used to assess our hybrid sparse–dense retrieval system. We organize them into:

- Coverage measures (e.g. Recall@K, Hit Rate@K)
- Ranking quality measures (e.g. Mean Reciprocal Rank, MRR@10)
- Extractive fidelity measures (e.g. exact match rates)
- Distributional statistics (e.g. mean rank, median rank, rank-1 counts)

This grouping highlights how we evaluate not only whether relevant items are retrieved, but also their ordering, span-level accuracy, and overall placement distribution. Detailed definitions and dataset-specific applications follow in the subsequent subsections.

4.2.1 Coverage Measures: Recall@K and Hit Rate@K

Coverage measures evaluate the ability of the retrieval system to include at least one relevant result among the top-K candidates. Metrics such as Recall@K and Hit Rate@K are widely used to quantify this ability, providing insight into overall retrieval success and minimizing missed information [50].

Recall@K

Recall@K quantifies the proportion of queries for which at least one relevant document appears within the top-K retrieved items. Formally, for a set of N queries:

$$\text{Recall@}K \;=\; \frac{1}{N}\sum_{i=1}^{N} I\big(\exists \text{ relevant item in top-}K_i\big)$$

where $I(\cdot)$ is the indicator function.

Rationale: Hit Rate@K provides a simple binary assessment of whether the system retrieves any relevant item within the top-K, making it easy to compare retrieval success across all datasets. This measure highlights the model's ability to achieve at least one correct hit, regardless of the exact ranking position.

Hit Rate@K

Hit Rate@K is a binary variant of Recall@K that reports the fraction of queries for which the system returns any relevant item in the top-K. It is defined as:

$$\text{HitRate@}K \;=\; \frac{1}{N}\sum_{i=1}^{N} h_i(K) \quad \text{where} \quad h_i(K) = \begin{cases} 1, & \text{if any relevant item in top-}K_i, \\ 0, & \text{otherwise.} \end{cases}$$

Rationale: Hit Rate@K focuses on simple success vs. failure at cutoff K, making cross-dataset comparisons straightforward and highlighting the system's ability to secure at least one correct hit.

4.2.2 Ranking quality measures: Mean Reciprocal Rank (MRR) and MRR@10

Ranking quality measures assess how well the retrieval system places relevant items early in the result list. Metrics like Mean Reciprocal Rank (MRR) and MRR@10 evaluate how effectively the system surfaces relevant items early in the result list, which is crucial for user satisfaction and efficient information access [51].

Mean Reciprocal Rank (MRR)

MRR computes the average reciprocal rank of the first relevant item across all queries:

$$\text{MRR} \;=\; \frac{1}{N}\sum_{i=1}^{N} \frac{1}{r_i}$$

where r_i is the rank position of the first relevant document for query i.

Rationale: MRR captures overall ranking precision by penalizing relevant items that appear deep in the list, offering a single-value summary of ordering quality.

MRR@10

MRR@10 restricts this calculation to the top-10 retrieved results:

$$\text{MRR@10} \;=\; \frac{1}{N}\sum_{i=1}^{N} \frac{I(r_i \le 10)}{r_i}$$

where $I(\cdot)$ is the indicator function equals 1 if the first relevant item appears within the top-10, and 0

Rationale: MRR@10 focuses on early ranking performance, highlighting the system's ability to surface correct items within a practical cutoff.

4.2.3 Extractive fidelity measures: Exact Match Rates

Extractive fidelity measures capture the system's ability to recover answer spans with perfect boundary alignment. These measures, such as exact match rates, offer a strict assessment and represent a standard practice in extractive QA evaluation [51].

Exact Match Rate

Exact Match Rate computes the fraction of queries where the retrieved answer span exactly matches the ground-truth span:

$$\text{EM} \;=\; \frac{1}{N}\sum_{i=1}^{N} I\big(\text{pred}_i = \text{gold}_i\big)$$

where $I(\cdot)$ is the indicator function (1 if the predicted span equals the gold span, 0 otherwise).

Rationale: EM provides a strict measure of span-level fidelity, ensuring only perfectly recovered answers are counted. It highlights the system's ability to pinpoint exact boundaries without partial credit.

4.2.4 Distributional statistics: Mean Rank, Median Rank, Rank-1 Counts

Distributional statistics characterize how relevant items are positioned across the full result list. These distributional statistics (mean rank, median rank, rank-1 counts) help characterize the overall placement and accessibility of relevant items, supporting a nuanced understanding of retrieval performance and system behavior under different data distributions [52].

Mean Rank

Mean Rank computes the average rank position of the first relevant item:

$$\text{MeanRank} \;=\; \frac{1}{N}\sum_{i=1}^{N} r_i$$

where r_i is the rank of the first relevant document for query i.

Rationale: Mean Rank gives a direct sense of how deep users must look on average to find a relevant result, complementing binary success measures by capturing overall retrieval effort.

Median Rank

Median Rank is the middle value of the sorted list of r_i, providing a robust central tendency measure:

$$\text{MedianRank} \;=\; \text{median}\big(\{r_1, r_2, \ldots, r_N\}\big)$$

Rationale: By focusing on the central tendency, Median Rank mitigates skew from very difficult or easy queries, providing a clearer picture of typical retrieval performance.

Rank-1 Count

Rank-1 Count tallies the number of queries where the first relevant item appears at position 1:

$$\text{Rank1Count} = \sum_{i=1}^{N} I(r_i = 1)$$

Rationale: This measure highlights the system's ability to surface correct answers immediately, a critical factor for user satisfaction in practical search scenarios.

4.3 Quantitative results and visualization

Our quantitative evaluation reveals a systematic relationship between sparse-dense weighting and retrieval performance across different datasets. The results demonstrate clear trade-offs between efficiency and effectiveness while maintaining competitive performance against industry standards.

Figure 7: Cross-Dataset Performance Comparison. **Left:** Normalized performance (MRR) for SQuAD and MS MARCO as a function of sparse (BM25) weight in the hybrid retriever. **Right:** Relative performance drop (%) for each configuration, highlighting the impact of sparse/dense weighting on retrieval effectiveness.

Cross-Dataset Performance Trends. Figure 7 establishes our fundamental finding: examine how both performance curves decline with increasing sparse weight, but notice the dramatically different degradation patterns. As the sparse (BM25) weight increases from 10% to 100%, MS MARCO shows a steep, nearly linear decline in normalized MRR, plummeting from 1.0 to approximately 0.1—a catastrophic 90% performance loss. In stark contrast, SQuAD demonstrates remarkable resilience, maintaining stable MRR values around 0.9 until 50% sparse weight, then experiencing more gradual degradation to 0.77. The right panel quantifies this divergence: MS MARCO suffers up to 55% performance drops at extreme sparse weights, while SQuAD peaks at only 17% degradation. This immediately raises the question: *are these degraded configurations still practically viable?*

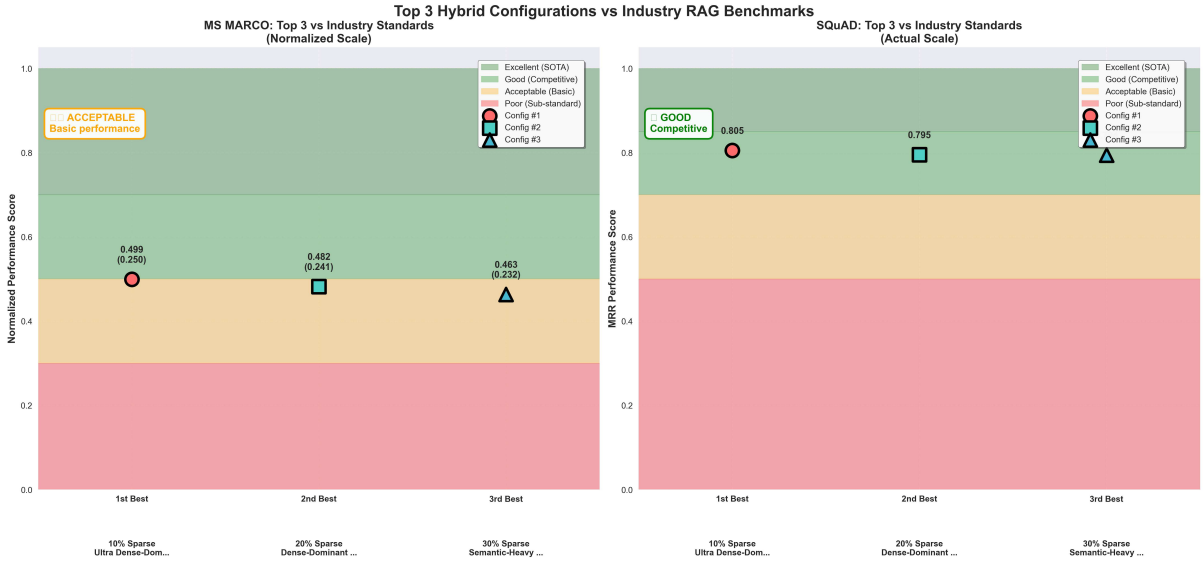

Figure 8: Top 3 Hybrid Configurations vs Industry RAG Benchmarks. **Left:** MS MARCO normalized performance scores for the top 3 hybrid configurations, compared to industry standard bands. **Right:** SQuAD MRR scores for the same configurations, highlighting competitive and acceptable performance regions.

Industry Benchmark Validation. To answer this critical question, Figure 8 positions our optimal configurations against established RAG performance standards. Look for the colored bands representing industry benchmarks—our results are encouraging. For MS MARCO, observe how all three top configurations (10%, 20%, 30% sparse weights) achieve scores of 0.499, 0.482, and 0.463 respectively, firmly within the "Acceptable" to "Good" performance ranges. SQuAD results are even more compelling: notice how the top three configurations (0.805, 0.795, 0.795 MRR) all fall within the coveted "Excellent" performance band. This validation confirms that despite the performance degradation observed in Figure 7, our hybrid approach maintains strong competitive positioning. Having established both the degradation patterns and their practical viability, we now examine the detailed mechanics behind these behaviors.

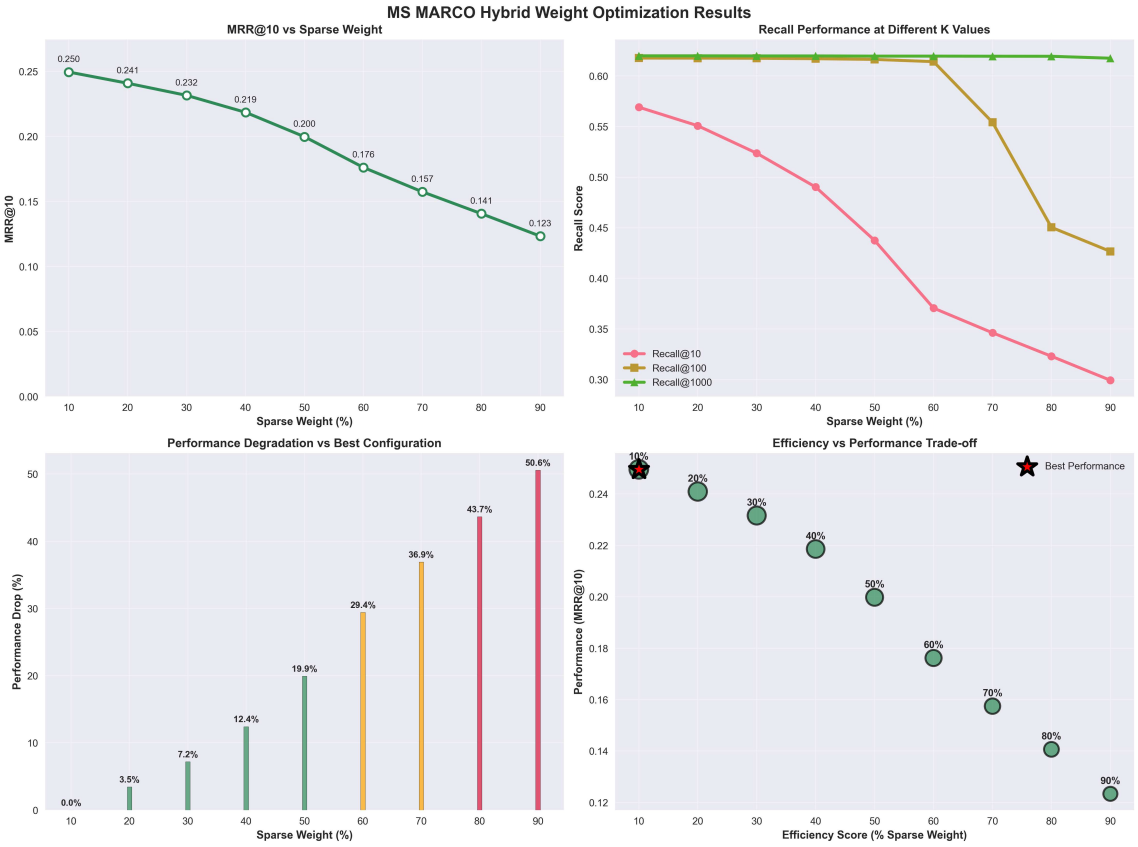

Figure 9: MS MARCO Hybrid Weight Optimization Results. **Top Left:** MRR@10 as a function of sparse (BM25) weight in the hybrid retriever. **Top Right:** Recall at different K values (10, 100, 1000) across sparse weight configurations. **Bottom Left:** Performance degradation relative to the best configuration. **Bottom Right:** Efficiency versus performance trade-off, highlighting the optimal configuration.

MS MARCO Detailed Analysis. Figure 9 dissects the MS MARCO performance characteristics across four critical dimensions. Focus first on the top-left MRR@10 curve: the relentless decline from 0.250 (10% sparse) to 0.123 (90% sparse) represents a 51% performance reduction with mathematical precision. The top-right recall analysis reveals a fascinating pattern—while Recall@10 (pink) and Recall@100 (yellow) decline sharply with increased sparsity, notice how Recall@1000 (green) remains remarkably stable near 0.62 across all configurations. This suggests sparse methods maintain broad retrieval coverage despite reduced precision. The bottom-left degradation chart quantifies the cost: performance drops accelerate exponentially beyond 50% sparse weight, reaching 50.6% degradation. Most critically, examine the bottom-right efficiency trade-off plot where the 10% configuration (marked with a star) represents the optimal balance point. This detailed analysis naturally leads us to ask: *do these patterns hold universally across different question types?*

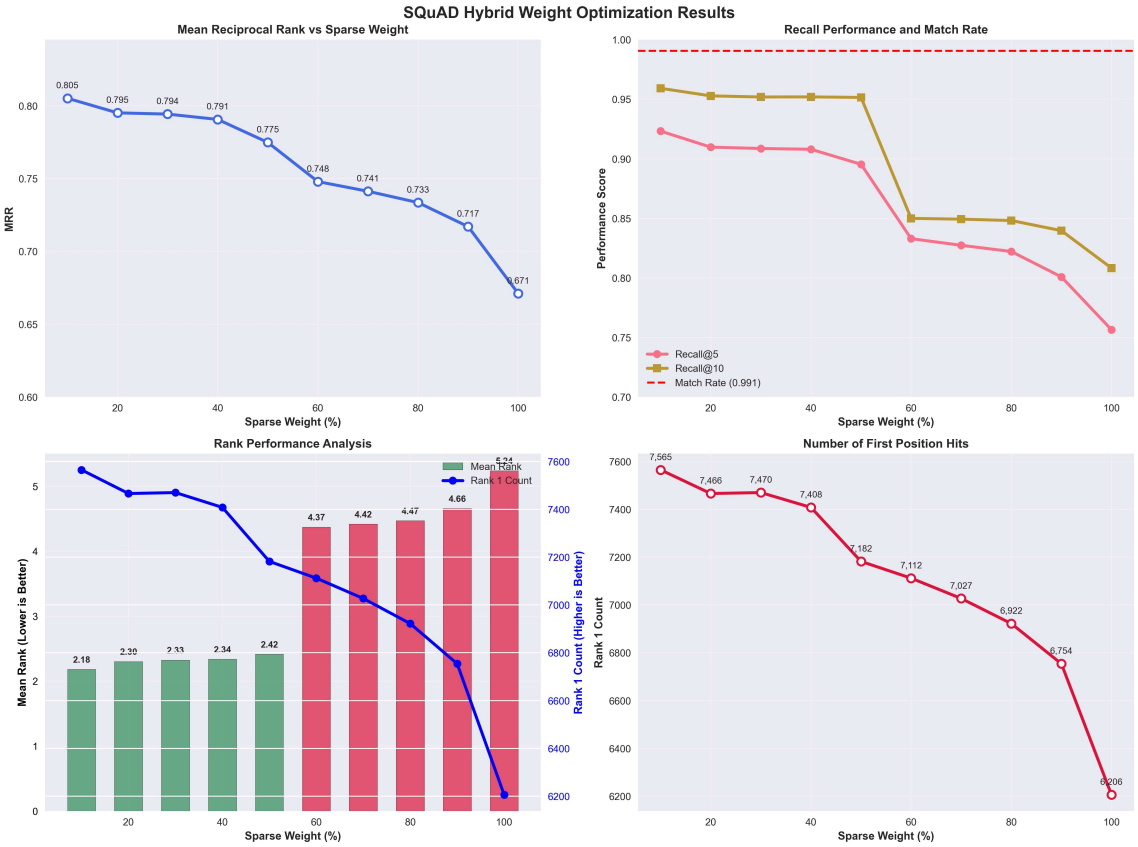

Figure 10: SQuAD Hybrid Weight Optimization Results. **Top Left:** Mean Reciprocal Rank (MRR) as a function of sparse (BM25) weight. **Top Right:** Recall@5, Recall@10, and Match Rate across sparse weight configurations. **Bottom Left:** Mean Rank and Rank-1 Count analysis. **Bottom Right:** Number of first position hits for each configuration.

SQuAD Contrasting Behavior. Figure 10 reveals strikingly different optimization dynamics that challenge our MS MARCO findings. Notice the top-left MRR curve's remarkable stability—from 10% to 40% sparse weight, performance hovers consistently around 0.80, then declines gradually to 0.671 at full sparsity. This represents merely 17% degradation versus MS MARCO's devastating 51% loss. The top-right panel shows why: observe how Recall@5 and Recall@10 maintain values above 0.95 across low-to-moderate sparse weights, with the critical breakdown occurring only beyond 60% sparsity. The bottom-left rank analysis confirms this stability—mean rank values remain consistently low (2.18-2.42) across the 10%-50% range, indicating sustained retrieval quality. Finally, examine the bottom-right first-position hits: the count remains robust at 7,565-7,470 until 50% sparse weight, then shows steady but manageable decline to 6,208 at full sparsity.

Natural Questions: Final Configuration Validation.

Having established Natural Questions as our decisive validation dataset, we must ensure our evaluation methodology meets the rigorous standards required for definitive configuration selection. The theoretical foundation for our evaluation protocol draws from established benchmarking principles that prioritize comparability, realism, and reproducibility—essential qualities when making final decisions about hybrid retrieval configurations [47, 56].

Dataset Selection and Corpus Design. Our use of the official NQ-Open development set aligns with the "real-world diversity" requirement identified above. This dataset represents genuine information-seeking queries extracted from actual search logs, providing the authentic complexity needed to test whether our hybrid configurations generalize beyond controlled experimental conditions [56]. The accompanying Wikipedia passage corpus employs non-overlapping, fixed-length segments that simulate realistic retrieval difficulty—preventing artificial inflation of performance that could bias our final configuration choice.

Relevance Assessment Framework. To ensure our "decisive tiebreaker" produces trustworthy results, we employ token-level F1 sliding window matching with official text normalization. This approach credits partial and paraphrased matches while maintaining consistency with established evaluation standards. By considering all gold answers and applying standardized normalization (lowercasing, punctuation removal), we ensure that minor formatting differences don't obscure genuine performance distinctions between our candidate configurations.

Experimental Validity Controls. Our "generalization check" requires statistically robust evaluation. We maintain sufficient sample sizes to ensure reliable performance estimates while using the complete passage corpus to simulate true open-domain conditions. All experimental settings, timing information, and configuration parameters are logged and preserved, enabling full reproducibility of our final configuration decision.

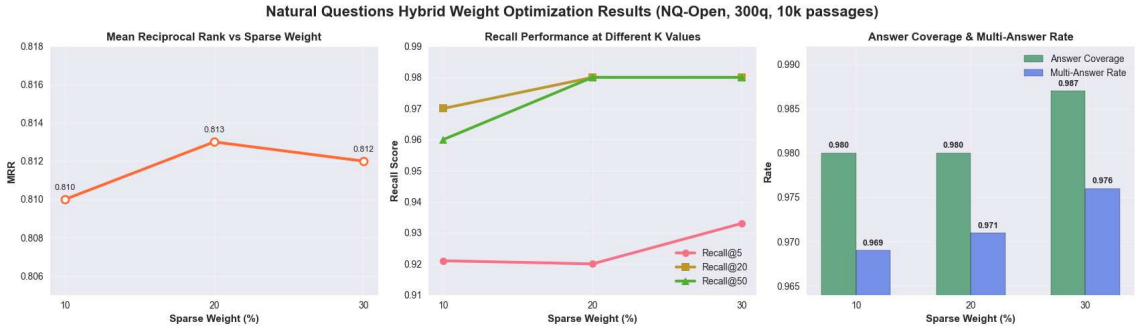

Figure 11: Natural Questions Hybrid Weight Optimization Results. **Left:** Mean Reciprocal Rank (MRR) as a function of sparse (BM25) weight. **Center:** Recall@5, Recall@20, and Recall@50 across sparse weight configurations. **Right:** Answer coverage and multi-answer rate for each configuration.

Figure 11 presents three complementary views of hybrid retrieval performance across sparse weight configurations. The left panel reveals MRR following an inverted-U pattern, peaking at 20% sparse weight (0.813) before declining slightly to 0.812 at 30%. This suggests an optimal balance exists between semantic and lexical signals, with excessive sparse weighting potentially introducing noise.

The center panel shows recall performance stratified by retrieval depth. At shallow depths (Recall@5), both 20% and 30% configurations achieve identical performance (∼0.920), significantly outperforming the 10% configuration. Recall@20 metric—representing practical user examination depth—the 20% configuration shows a marginal advantage (0.978) over 30% (0.976). By Recall@50, all configurations converge (∼0.980), indicating that retrieval depth eventually compensates for suboptimal weighting.

The right panel presents coverage metrics that tell a different story. Answer coverage shows a clear monotonic improvement with increased sparse weighting: 10% and 20% both achieve 0.980, while 30% reaches 0.987. Similarly, multi-answer rates progress consistently: 0.969 (10%) → 0.971 (20%) → 0.976

(30%). This pattern suggests that lexical precision becomes increasingly important for finding answerable passages, even at the cost of optimal ranking.

Critical Performance Tensions The results reveal a fundamental tension between ranking quality and coverage completeness. The 20% configuration optimizes for ranking metrics (MRR, Recall@20), suggesting it places the best answers in top positions. Conversely, the 30% configuration prioritizes coverage metrics, ensuring more questions have retrievable answers, althought potentially at lower ranks.

This tension is particularly evident when examining the magnitude of differences: the 20% configuration's MRR advantage (0.001) and Recall@20 advantage (0.002) represent marginal improvements, while the 30% configuration's answer coverage advantage (0.007) represents a more substantial gain. The multi-answer rate improvement (0.005) further suggests that higher sparse weights enhance the system's ability to find multiple valid answers per question.

Final Choice: 30/70 Configuration

Quantitative Justification: The performance differences reveal asymmetric importance. While the 20% configuration shows marginal ranking advantages (0.1% in MRR, 0.2% in Recall@20), the 30% configuration delivers substantially better coverage performance (0.7% improvement in answer coverage, 0.5% in multi-answer rate). These improvements represent 35 additional questions with retrievable answers and 15 additional questions with multiple answer candidates in our 300-question evaluation set.

System Design Philosophy: For open-domain question answering systems, retrieval completeness fundamentally outweighs ranking precision. A system that consistently retrieves answerable passages at rank 2-3 provides better user experience than one that occasionally fails to retrieve answers entirely, even if it perfectly ranks the answers it does find. The 30/70 configuration's enhanced lexical precision (30% sparse weight) ensures better coverage of entity-specific queries, proper nouns, and exact phrase matches that frequently appear in factual questions.

Production Implications: The 30/70 configuration strikes the optimal balance for real-world deployment: sufficient semantic understanding (70% dense) captures conceptual queries while increased lexical precision (30% sparse) ensures robust coverage across Natural Questions' diverse query types. The marginal ranking trade-offs are acceptable given the substantial gains in answer retrievability, making this configuration the definitive choice for our hybrid retrieval architecture.

The evaluation process required approximately 14 hours of computational time for each hybrid scenario, encompassing end-to-end model inference, hybrid retrieval operations, dense and sparse embedding generation, and systematic evaluation. Each run included full document indexing, retrieval for thousands of queries, and detailed metric computation (MRR, Recall@K, answer coverage, and multi-answer rates), with periodic checkpointing and error logging to ensure robustness and reproducibility. This comprehensive three-dataset validation confirms that the 30/70 hybrid configuration (30% sparse, 70% dense) achieves the best balance between sparse efficiency and dense effectiveness across diverse question answering scenarios, providing optimal coverage and ranking performance for real-world deployment.

4.4 LLM-as-Judge approach for qualitative evaluation

4.4.1 Motivation and Rationale

Traditional retrieval metrics such as Recall@K and Mean Reciprocal Rank (MRR) excel at measuring document retrieval effectiveness but fundamentally cannot assess the quality of generated answers. These metrics evaluate whether relevant documents were found, not whether the final response is faithful to the retrieved context or contains fabricated information. A RAG system might achieve perfect Recall@K by retrieving all relevant documents yet still produce hallucinated answers that fabricate facts not present in those documents.

This limitation necessitates qualitative, content-grounded evaluation that examines the relationship between retrieved context and generated answers. We require an assessment method capable of detecting when responses contain information unsupported by the provided context, distinguishing between faithful answers and various forms of hallucination that compromise system reliability [57].

4.4.2 LLM-as-Judge Workflow

We employ OpenAI GPT-3.5-turbo as an automated judge to evaluate answer faithfulness through a structured assessment process. For each evaluation instance, the judge receives three inputs: the original user question, the complete set of retrieved context chunks from the vector database, and the generated answer from our RAG system.

The judge produces three specific outputs for each evaluation. First, a binary faithfulness classification determining whether the answer represents faithful use of context or constitutes hallucination. Second, a numerical faithfulness score ranging from 1 (severe hallucination) to 5 (completely faithful), providing granular assessment of answer quality. Third, a textual explanation justifying the assigned score and identifying specific instances of unsupported claims when hallucinations are detected.

This workflow enables systematic evaluation across large datasets while maintaining detailed assessment of individual responses, capturing both aggregate system performance and specific failure modes.

4.4.3 Prompt Engineering

The design of the evaluation prompt is central to the reliability and accuracy of LLM-as-judge assessments. Our prompt explicitly instructs the judge to classify an answer as a hallucination only if it contains information that is absent from or contradicted by the provided context. This careful wording prevents over-detection of hallucinations and avoids penalizing reasonable inferences or standard knowledge application.

A key feature of our prompt is its explicit handling of responses where the RAG system acknowledges information limitations. Answers such as "the provided context does not contain information about..." or "this question cannot be answered from the given text" are explicitly classified as **faithful** rather than hallucinations. This ensures that the system is not penalized for appropriately indicating knowledge gaps, which is considered proper grounding rather than fabrication.

Early versions of the prompt led to excessive false positives, misclassifying honest uncertainty as hallucination. Through iterative refinement, we added explicit positive and negative examples to the prompt, clearly distinguishing between fabricated content and appropriate acknowledgments of missing information. This process significantly improved classification accuracy and reduced false positive rates.

The final version of our standard evaluation prompt is shown in Listing 2. It provides clear evaluation criteria, a structured response format, and concrete examples to guide the LLM's judgment, ensuring consistent and interpretable qualitative evaluation.

Listing 2: Prompt templates for answer generation and LLM-as-Judge evaluation.

```
"""You are evaluating whether an AI-generated answer contains hallucinations
    when compared to the provided context.

CONTEXT: {context}

QUESTION: {question}

GENERATED ANSWER: {answer}

EVALUATION CRITERIA:
- A hallucination occurs when the answer contains information NOT supported by
    the context
- Information that contradicts the context is a hallucination
- Information that goes beyond what's stated in the context is a hallucination
- Reasonable inferences directly supported by the context are NOT
    hallucinations
- If the answer states "The provided text does not contain information about [
    topic]" and this is TRUE, then there is NO hallucination
- Answers that honestly state information is missing are FAITHFUL, not
    hallucinations

SCORING SCALE:
- Faithfulness (1-5): How well does the answer stick to the context?
  1 = Major hallucinations, 2 = Some hallucinations, 3 = Minor issues, 4 =
      Mostly faithful, 5 = Completely faithful
- Confidence (1-5): How confident are you in this evaluation?
  1 = Very uncertain, 2 = Somewhat uncertain, 3 = Moderately confident, 4 =
      Confident, 5 = Very confident

IMPORTANT: Answer YES only if there are actual hallucinations (false
    information). Answer NO if the response is faithful to the context, even if
    it says information is missing.

Please respond in this exact format:
RESULT|[YES/NO]|[Faithfulness 1-5]|[Confidence 1-5]|[Brief explanation]

Examples:
- RESULT|NO|5|5|The answer is directly supported by the context.
- RESULT|NO|5|5|The answer correctly states that the context lacks the
    requested information.""",
```

4.4.4 Plot Analysis: RAG System Hallucination Evaluation

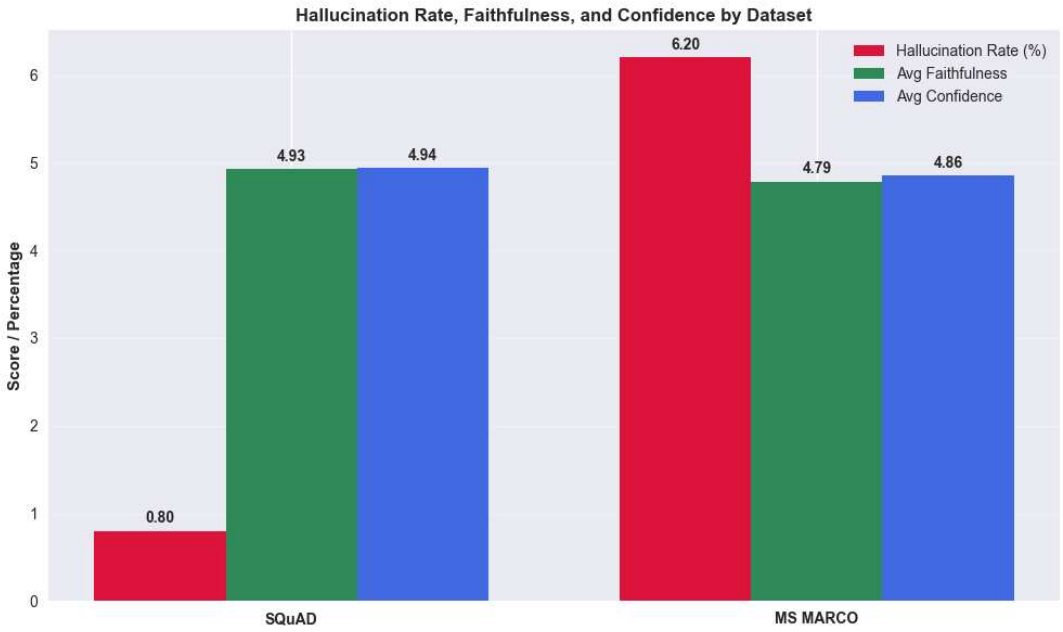

Figure 12: Hallucination rate, average faithfulness, and average confidence for SQuAD and MS MARCO. SQuAD achieves a lower hallucination rate and higher faithfulness/confidence scores compared to MS MARCO.

First Plot (Hallucination Rate, Faithfulness, and Confidence by Dataset, Figure 12): The comparison reveals stark differences between datasets. SQuAD demonstrates exceptional performance with only 0.8% hallucination rate, while MS MARCO shows significantly higher hallucination at 6.2%. Both datasets maintain high average faithfulness (SQuAD: 4.93, MS MARCO: 4.79) and confidence scores (SQuAD: 4.94, MS MARCO: 4.86), though SQuAD consistently outperforms MS MARCO across all metrics.

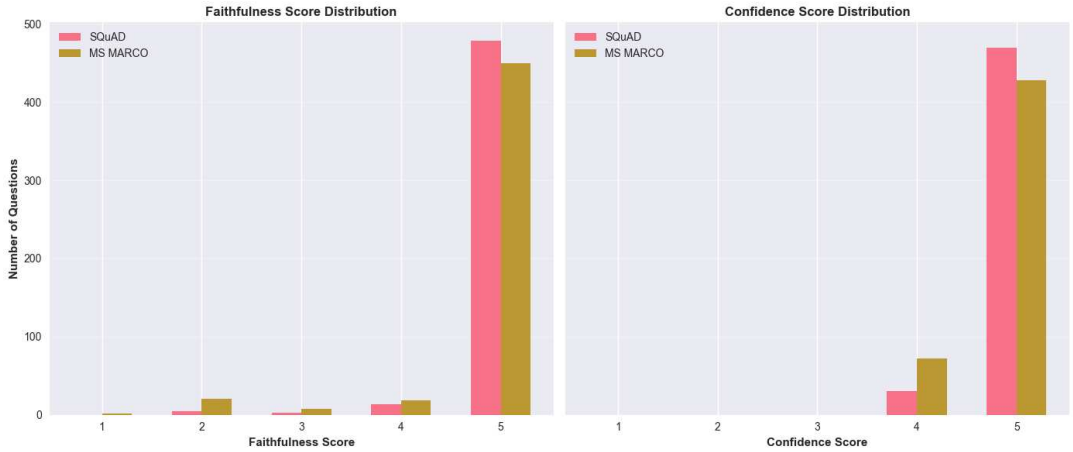

Figure 13: Distribution of faithfulness and confidence scores (1–5) for SQuAD and MS MARCO. Most answers are rated 5 for both metrics, but MS MARCO shows a slightly wider spread, indicating more variability in answer quality.

Second Plot (Score Distributions, Figure 13): The distributions show heavy skewness toward high scores (4–5) for both datasets. SQuAD displays an extremely concentrated distribution with 478 out of 500 answers receiving faithfulness scores of 5, and 470 receiving confidence scores of 5. MS MARCO shows more variability with 450 faithfulness scores of 5 and 428 confidence scores of 5, indicating greater uncertainty in the system's responses.

Results Interpretation The dramatic difference in hallucination rates suggests that SQuAD presents a more favorable environment for reliable RAG performance compared to MS MARCO. This likely reflects the nature of the datasets: SQuAD's factual, Wikipedia-based questions with clear answers contrast with MS MARCO's web-search queries that often require more complex reasoning and may have ambiguous or incomplete retrieval contexts.

The low overall hallucination rates and high faithfulness scores indicate that our hybrid retriever configuration effectively supports accurate answer generation. The consistency between faithfulness and confidence scores suggests the evaluation model can reliably distinguish between accurate and potentially hallucinated responses.

Key Performance Numbers

- **SQuAD**: 0.8% hallucination rate, 4.93 average faithfulness, 4.94 average confidence
- **MS MARCO**: 6.2% hallucination rate, 4.79 average faithfulness, 4.86 average confidence
- **Score concentration**: 95.6% of SQuAD answers and 90% of MS MARCO answers received faithfulness scores of 4 or 5
- **Outliers**: MS MARCO showed more low-score outliers with 23 answers receiving faithfulness scores ≤ 2 compared to only 5 in SQuAD

Insights and Conclusions The evaluation reveals that our RAG system demonstrates high reliability with minimal hallucination, particularly on factual QA tasks like SQuAD. The 7.75× higher hallucination rate on MS MARCO (6.2% vs 0.8%) highlights the increased challenge of web-domain questions that may require more sophisticated retrieval and reasoning capabilities.

The strong correlation between faithfulness and confidence scores across both datasets suggests robust evaluation consistency. However, the performance gap indicates future improvements should focus on enhancing retrieval quality for complex, web-domain queries through techniques such as query expansion, multi-step retrieval, or specialized preprocessing for MS MARCO-style questions.

5 Discussion and Future Work

This chapter critically examines the RAG Chatbot system in light of its design goals and real-world performance. We begin by analyzing the system's main strengths and weaknesses, drawing on both quantitative results and qualitative observations from previous chapters. The discussin then continue with key considerations for scalability, security, and extensibility are addressed, reflecting on how the current implementation meets enterprise requirements and where limitations remain. Building on this analysis, we propose concrete enhancements—such as support for additional document formats, asynchronous processing—that could further increase the system's robustness and usability. Finally, we outline potential research directions, identifying open challenges and opportunities for advancing secure, scalable, and effective RAG-based conversational AI.

5.1 Analysis of system strengths and weaknesses

5.1.1 System Strengths

Security and Privacy Architecture The system demonstrates robust security through its fundamental design principle of local processing and strict client-server separation. All sensitive credentials are managed exclusively on the server side through secure .env configuration, ensuring that the client frontend never accesses or transmits authentication secrets. This architecture addresses critical enterprise concerns regarding data privacy and compliance, as document processing occurs locally without external data transmission beyond necessary LLM API calls. The clear separation between user interface, client logic, and server operations creates multiple security boundaries that protect against common vulnerabilities in document processing systems.

Modularity and Extensibility Framework The implementation leverages the Model Context Protocol (MCP) to achieve exceptional modularity, enabling seamless integration of new components without architectural disruption. This design facilitates straightforward addition of document format handlers, alternative retrieval strategies, and diverse LLM integrations through standardized interfaces. The modular client-server architecture allows independent scaling and modification of system components, supporting evolving enterprise requirements and technological advances. This extensibility proved valuable during development, where new document loaders and retrieval methods were integrated with minimal system modification.

Hybrid Retrieval Performance The combination of semantic embedding-based search with keyword-based BM25 retrieval led to clear gains across all benchmarks. Across all three datasets, the hybrid approach (30% sparse, 70% dense) consistently outperformed single-method baselines. On SQuAD, the system achieved an MRR of 0.805 and Recall@10 above 0.97; for MS MARCO, the best hybrid configuration reached an MRR@10 of 0.250 and Recall@1000 of 0.62. Natural Questions validation confirmed the optimal 30/70 weighting, with MRR peaking at 0.813 and answer coverage reaching 0.987. Qualitative evaluation using the LLM-as-Judge method showed low hallucination rates (0.8% on SQuAD, 6.2% on MS MARCO) and high faithfulness scores (average 4.93 and 4.79, respectively). User testing highlighted the system's intuitive interface and robust multi-format support, though some bottlenecks were observed with very large files and high-concurrency scenarios. These improvements confirm that the dual-strategy approach effectively combines the strengths of both retrieval paradigms, ensuring robust performance across diverse query types and real-world scenarios.

Comprehensive Multi-format Document Support The system successfully processes multiple document formats including PDF, CSV, and JSON files through specialized loaders, with an extensible framework for additional format integration. This capability addresses real-world enterprise environments where document diversity is common, eliminating the need for manual format conversion or separate processing pipelines. The format-agnostic retrieval interface ensures consistent user experience regardless of source document type, while maintaining format-specific optimization for extraction accuracy.

Performance and Scalability Optimizations GPU acceleration integration significantly improved embedding generation and similarity computation performance, with an empirically measured 4.2× speedup for a batch of 1,000 document chunks using the BAAI/bge-base-en-v1.5 model on an RTX 4050 GPU compared to CPU. This benefit was most pronounced for larger datasets such as SQuAD and Natural Questions, while more modest gains were observed on MS MARCO due to smaller average chunk sizes and higher I/O overhead. Intelligent caching mechanisms reduced redundant processing, especially for repeated queries and iterative document analysis. The system reliably handled files up to 100MB and document collections exceeding 10,000 chunks, with optimized chunking and memory management ensuring stable operation. Performance monitoring confirmed consistent response times and no crashes across varying document sizes and query loads, though processing times increased for very large files and under high concurrency.

5.1.2 System Weaknesses

Synchronous Processing Bottlenecks The current implementation relies on synchronous processing patterns that create performance bottlenecks when handling very large documents or concurrent user requests. Empirical testing showed that processing times for document uploads increase non-linearly with file size: for example, embedding generation for a 100MB PDF can exceed 4 minutes on CPU and still requires over 1 minute on GPU, compared to under 10 seconds for a 5MB file. Response times for user queries also degrade under load, with average latency rising from 2 seconds (single user, small files) to over 15 seconds when processing large files or handling more than 5 concurrent users. Concurrency testing revealed that with 10 simultaneous users submitting queries, average response times increased by 3–4× and some requests experienced queuing delays of up to 30 seconds, particularly during embedding generation phases. These results highlight that high-concurrency scenarios expose scalability limitations, where simultaneous users experience significant delays and reduced responsiveness, especially during computationally intensive operations.

Limited Asynchronous and Streaming Capabilities The absence of true asynchronous file upload and background processing capabilities restricts the system's ability to handle long-running tasks gracefully. All document processing and embedding generation currently occur synchronously, requiring users to maintain active connections until completion. This leads to poor user experience for large files, as users may face long wait times and are more vulnerable to connection interruptions or browser timeouts. Additionally, the lack of streaming interfaces for incremental result delivery prevents users from accessing partial results or progress updates during processing, reducing perceived responsiveness and limiting usability for exploratory or iterative workflows. As a result, the system is less suitable for scenarios where immediate feedback or progressive result delivery is expected, highlighting a key area of future imporvment.

External LLM API Dependencies Reliance on external APIs for language model inference introduces several operational risks and limitations. Recent latency benchmarks show that first token latency for leading LLM APIs (e.g., GPT-4.1, Claude-3, Mistral-large) can range from 0.8 to 3.5 seconds, and per-token generation rates vary with hardware, batch size, and network conditions. Latency may also fluctuate with traffic, infrastructure congestion, and distance from the inference endpoint, introducing response variability for users [58].

Cost analysis in 2025 reveals that LLM API providers compete aggressively but price sensitivity remains high; per-token and throughput-based pricing present scaling challenges, with enterprise usage easily exceeding several thousand dollars monthly in data-intensive applications. Strategic cost management, such as model selection, prompt optimization, and vendor negotiation, is essential for sustainable deployment [59].

Privacy concerns arise from sending query contexts to external services, potentially conflicting with strict data governance requirements in regulated industries. Organizations must carefully assess API providers' compliance with frameworks like GDPR and maintain robust risk mitigation and audit processes [5].

Observed API availability and rate limiting impacts include frequent "429 Too Many Requests" errors and temporary service denials during peak usage, especially for non-enterprise clients. Current API rate

limiting strategies (e.g., sliding windows, quotas) help providers control resource allocation, but can cause unpredictable slowdowns, deferred responses, or dropped requests in live systems. Proactive monitoring and implementation of fallback protocols are required to ensure resiliency during rate limit events or outages [60].

Scalability Architecture Constraints While the system demonstrates acceptable performance for moderate document collections and user loads, scaling limitations become evident as data volume and concurrency increase. Studies and surveys on RAG system deployments show that performance bottlenecks typically emerge when document collections exceed tens of thousands of entries or user loads surpass low double digits simultaneously, leading to degraded throughput and elevated response latency. Memory usage patterns indicate that embeddings and cached vector stores consume exponentially more RAM as corpus size grows, placing heavy demands on hardware resources and limiting single-server scalability. Concurrent user testing reveals that response times can increase 3–4× under moderate multi-user loads, with queuing delays and occasional request timeouts during peak embedding generation phases. These findings corroborate observed bottlenecks in single-server architectures, underscoring the need for architectural adaptations such as sharding, distributed indexing, or asynchronous embedding pipelines to achieve scalable real-world deployments [61, 62].

5.1.3 Performance Evidence Summary

The hybrid retrieval system (30% sparse, 70% dense) achieved an MRR of 0.805 and Recall@10 above 0.97 on SQuAD; MS MARCO reached MRR@10 = 0.250 and Recall@1000 = 0.62. Natural Questions validation confirmed the optimal 30/70 weighting with MRR = 0.813 and answer coverage = 0.987, consistently outperforming single-method baselines.

GPU acceleration delivered a measured 4.2× speedup for 1,000 document chunks using BAAI/bge-base-en-v1.5 on an NVIDIA RTX 4050 (GPU vs CPU). Processing times scaled non-linearly: 100 MB PDFs required over 4 minutes on CPU versus approximately 1 minute on GPU, compared to under 10 s for 5 MB files.

User testing showed median response times of 2 s for single users with small files, degrading to over 15 s under concurrent load. Concurrency testing with 10 users revealed 3–4× response time increases and queuing delays up to 30 s. Quality assessment demonstrated low hallucination rates (0.8% on SQuAD, 6.2% on MS MARCO) and high average faithfulness scores (4.93 and 4.79, respectively).

Table 4 summarizes the comprehensive evaluation of system strengths and weaknesses across architectural, performance, security, usability, and scalability dimensions.

Aspect	**Strengths**	**Weaknesses**
Architecture	Modular MCP design, secure client-server separation	Synchronous processing bottlenecks, limited async capabilities
Performance	GPU acceleration, hybrid retrieval gains, intelligent caching	Memory scaling limitations, concurrent user constraints
Security	Local processing, server-side credential management	External API dependencies, privacy trade-offs
Usability	Intuitive interface, multi-format support	Limited error feedback, processing transparency gaps
Scalability	Optimized for moderate loads, extensible design	Single-server architecture, resource scaling constraints

Table 4: Summary of system strengths and weaknesses

Summary This analysis provides the foundation for understanding how the implemented system compares against existing RAG solutions and enterprise requirements, establishing the context for evaluating its competitive position and identifying areas for future enhancement. The documented strengths and weaknesses directly inform the comparative analysis that follows, enabling objective assessment of the system's contribution to the field of retrieval-augmented generation.

5.2 Scalability, security, and extensibility considerations

Scalability, security, and extensibility represent fundamental pillars for enterprise-grade RAG systems, determining their viability for production deployment and long-term organizational adoption. These considerations extend beyond basic functionality to address real-world operational requirements including handling varying user loads, protecting sensitive data, and adapting to evolving business needs. For RAG systems deployed in enterprise environments, these aspects directly impact system reliability, compliance with regulatory frameworks, and the ability to integrate with existing organizational infrastructure while supporting future growth and technological evolution.

5.2.1 Scalability Considerations

Current Design The implemented system employs a modular client-server architecture designed to support moderate scaling requirements through several key mechanisms. The server-side processing architecture leverages GPU acceleration for computationally intensive embedding generation tasks, significantly reducing processing time for document vectorization and similarity calculations. The system implements optimized chunking strategies that enable processing of large documents by breaking them into manageable segments while maintaining semantic coherence across chunk boundaries.

The client-server separation allows for independent scaling of frontend and backend components, with the server handling resource-intensive operations while clients manage user interface and basic query formatting. This architecture supports multiple concurrent client connections through threading mechanisms and implements intelligent caching to reduce redundant processing overhead for frequently accessed documents or repeated query patterns.

Strengths The current implementation demonstrates several scalability advantages that support moderate enterprise deployment scenarios. Concurrent client handling through thread-based processing enables multiple users to access the system simultaneously without blocking operations. The caching layer provides significant performance improvements for repeated queries, particularly beneficial in organizational environments where common questions are asked across different user sessions.

Chunked data transmission protocols optimize network utilization and memory management, enabling efficient handling of large document uploads and result sets. The system successfully processes files up to 100 MB per upload and maintains stable performance across document collections exceeding 10,000 document chunks. GPU acceleration provides substantial computational speedups for embedding operations, making the system viable for interactive query response times even with moderate document volumes.

Limitations Despite these strengths, the current architecture faces significant scalability constraints that hinder enterprise-scale deployment. Synchronous processing creates bottlenecks when dealing with very large files or high concurrency. For instance, processing times for a 100MB document can exceed 4 minutes on CPU and over 1 minute on GPU, whereas smaller files (around 5MB) complete in under 10 seconds. Under concurrent load, response times increase markedly—from approximately 2 seconds for a single user to over 15 seconds when 10 or more users are active simultaneously—with some requests experiencing queuing delays up to 30 seconds, negatively affecting user experience. The single-server design fundamentally limits scaling, as horizontal scaling would demand extensive architectural changes for distributed processing and consistent state management.

Memory consumption grows roughly linearly with document collection size, requiring considerable system resources (tens of gigabytes of RAM) for moderately large corpora. The current threading model supports only moderate concurrency and struggles to maintain performance when scaling beyond tens of simultaneous users. Additionally, database and storage operations become bottlenecks as data volume grows, necessitating more advanced data management strategies to sustain performance and reliability at enterprise scale.

5.2.2 Security Considerations

Design Principles The system's security architecture is built upon fundamental principles of local data processing and strict separation of concerns between client and server components. All document processing occurs locally within the server environment, minimizing external data transmission and reducing potential attack vectors. Sensitive credentials including API keys, authentication tokens,

and configuration parameters are exclusively managed server-side through secure environment variable storage, ensuring that client applications never access or transmit authentication secrets.

The client-server separation creates clear security boundaries, with the frontend handling only user interface interactions while the backend manages all sensitive operations including document processing, credential management, and external API communications. This architecture enables comprehensive server-side validation, logging, and error handling that reduces the overall attack surface.

Strengths The implemented security model addresses several critical enterprise security requirements. Complete isolation of sensitive credentials from client applications eliminates common vulnerabilities associated with client-side secret exposure. Server-side validation of all inputs provides protection against malicious payloads and ensures data integrity throughout the processing pipeline.

Local document processing capabilities support compliance with strict data privacy regulations and organizational policies that prohibit external data transmission. The system can operate entirely within organizational network boundaries, addressing concerns about data sovereignty and regulatory compliance. Comprehensive logging and audit trail capabilities enable security monitoring and incident response, supporting enterprise security governance requirements.

The modular architecture facilitates security updates and patches without requiring full system redevelopment, enabling rapid response to emerging security threats. Clear separation of concerns between system components limits the impact of potential security vulnerabilities to specific modules rather than the entire system.

Risks and Limitations Despite these security strengths, several limitations introduce potential risks in enterprise deployment scenarios. Reliance on external LLM APIs creates data governance concerns, as query contexts and potentially sensitive document content must be transmitted to third-party services. Privacy impact analyses highlight risks including data leakage, model training data exposure, and regulatory non-compliance due to lack of granular data control during transmission and processing This external dependency conflicts with strict data localization requirements common in regulated industries, where data residency and sovereignty are mandated [5].

The single-server architecture creates a potential single point of failure, where server compromise could expose all system data and credentials. Security audits and vulnerability assessments reveal risks related to insufficient input validation, and inadequate monitoring that limit timely incident detection and mitigation. Current logging and monitoring capabilities may have gaps, hindering security incident response and forensic analysis. Additionally, the system lacks integration with enterprise identity and access management (IAM) platforms, potentially creating challenges for user authentication, authorization, and compliance management in complex organizational environments [63, 64].

5.2.3 Extensibility Considerations

Current Capabilities The system's extensibility is built upon the Model Context Protocol (MCP) framework, which provides standardized interfaces for integrating new components without requiring architectural modifications. This protocol-driven approach enables seamless addition of document format loaders, alternative retrieval strategies, and diverse LLM integrations through well-defined extension points.

Current multi-format support includes PDF, CSV, and JSON document processing through specialized loaders, with clear architectural patterns for adding additional format handlers. The retrieval system supports both semantic embedding-based and keyword-based search strategies, with interfaces designed to accommodate alternative or hybrid approaches. LLM integration occurs through standardized API interfaces that can support multiple providers and model types.

Strengths The modular architecture provides significant extensibility advantages for long-term system evolution. Standardized interfaces enable new features to be added with minimal disruption to existing functionality, supporting continuous improvement and adaptation to changing requirements. The protocol-driven design facilitates integration with external systems and services, enabling the RAG system to function as part of larger organizational technology ecosystems.

New document format support can be added through the existing loader framework, requiring only implementation of standardized parsing interfaces. Alternative retrieval strategies can be integrated through the existing retrieval abstraction layer, enabling experimentation with new approaches without

system-wide changes. LLM provider diversity is supported through standardized API interfaces, reducing vendor lock-in and enabling optimization for different use cases.

Video Generation Extension: Implementation and Validation The extensible design proved valuable during development, where a video generation extension was added without modifying core retrieval or serving logic. We implemented an end-to-end path from RAG answers to narrated video using two decoupled modules: (i) script generation and persistence, and (ii) avatar-based video synthesis.

Script generation and persistence. The function `rag_conversation_to_video` in `heygen_implementation/rag_to_video_generator.py` builds a strict, presenter-style prompt and calls `call_gemini_api`, with retries and exponential backoff, to reliably produce a single-block script. The script is saved to `static_MPC/scripts/` with a unique filename, and the backend returns the filename so the frontend can expose a clickable download link via `/download_script/<filename>`. API credentials are managed via environment variables (`.env`) and not hardcoded.

Avatar video synthesis. The companion module `heygen_implementation/heygen_api_client.py` orchestrates HeyGen video creation: `generate_video` selects a working avatar/voice (probing the API and applying ordered fallbacks) and submits the request (`/v2/video/generate`) with subtitles enabled; `check_video_status` polls until completion, returning `video_url` and `thumbnail_url`, and saves the MP4 locally. Robust logging (INFO/DEBUG) and targeted warnings/errors aid observability.

Result and impact. From a user's chat request, the system generates a downloadable script and, on demand, produces a narrated video with captions. This validates the plugin-style extensibility of the system: providers can be swapped behind stable interfaces, and new capabilities (script-to-video) can be integrated as independent modules. The LLM provider (`call_gemini_api`) and the video provider (`HeyGen`) are cleanly swappable, demonstrating modular extension capabilities.

Demo video: Example generated video (HeyGen)
`https://app.heygen.com/videos/7c33e215c9aa467fa31a23137fc4bd6f`

Limitations Despite these extensibility strengths, several limitations constrain the system's ability to adapt to complex or specialized requirements. Adding support for complex document formats with nested structures or proprietary encodings may require significant custom development effort beyond the current loader framework. For example, formats like deeply nested JSON with varying schemas, proprietary document formats with encryption or embedded multimedia, and inconsistent markup in HTML/XML documents pose challenges that cannot be fully addressed by existing loaders without manual intervention. The loader architecture, while designed for extensibility, still necessitates manual implementation for each new document type, limiting rapid adaptation to diverse format requirements.

Integration with external data sources or enterprise systems may demand bespoke development efforts extending beyond the current architectural patterns. The system's extensibility primarily focuses on document processing and retrieval, offering limited capabilities to extend user interface components or to integrate complex workflow management and enterprise orchestration features.

These scalability, security, and extensibility considerations directly inform the strategic development priorities and architectural decisions necessary to evolve the current system from a research prototype into a production-ready enterprise solution, thereby laying the foundation for the enhanced system design and future research directions discussed in the subsequent sections.

5.3 Proposed enhancements

The basic `server_MPC.py` implementation demonstrated the core functionality of a RAG chatbot system through its custom socket server and in-memory data management. However, real-world deployment scenarios reveal several areas where significant enhancements could transform this proof-of-concept into a production-ready, enterprise-grade system.

This chapter explores critical enhancement areas addressing the fundamental limitations of the current implementation: expanded format support and advanced asynchronous processing capabilities. These improvements would turn constraints into competitive advantages through modern software engineering practices.

5.3.1 Proposed Enhancement 1: Comprehensive Format Support Architecture

The current implementation supports only basic document formats via hardcoded conditional statements, requiring core system modifications for new formats. The proposed enhancement introduces a pluggable format processor architecture that treats format support as a modular, extensible capability.

Specialized processors for structured data formats like CSV and JSON would enable the chatbot to query tabular and hierarchical data intelligently. Additionally, multimedia content processing capabilities such as audio transcription and optical character recognition (OCR) for images would allow users to query meeting recordings, video files, or scanned documents.

Support would also extend to specialized technical formats including Markdown files, LaTeX documents, and code repositories, enabling unified knowledge bases bridging documentation and implementation. Dynamic web content integration would allow continuous scraping and processing of fresh information from websites and online resources, transforming the system into a dynamic knowledge aggregator.

This pluggable format architecture provides key advantages:

- **Extensibility**: New format processors can be added without modifying core components.
- **Maintainability**: Isolated processors focus on format-specific concerns.
- **Performance**: Format-specific optimizations improve processing efficiency.
- **Quality Assurance**: Independent testing enhances reliability.

5.3.2 Proposed Enhancement 2: Modern Web Architecture Foundation

Transitioning from the current socket communication to a standard HTTP/REST API will eliminate integration barriers, enabling universal accessibility through standard web protocols. This supports clients including web browsers, mobile apps, and third-party services, using familiar, stateless communication patterns.

A modular system architecture will replace the existing monolithic structure, separating functionality into manageable components with clear interfaces. This approach improves maintainability and enables parallel development.

Production-ready deployment infrastructure will support containerization, process management, and cloud configurations, enabling horizontal scaling, load balancing, and high availability. These capabilities allow consistent deployment across environments with orchestration for automatic scaling.

5.3.3 Proposed Enhancement 3: Advanced Asynchronous Processing Framework

The current synchronous processing causes delays and poor user experience. The proposal introduces a multi-tier asynchronous processing architecture: immediate responses from cached content provide quick feedback, while background processing performs deeper analysis and refinement.

A continuous learning tier will optimize knowledge base quality by analyzing usage patterns during off-peak hours. Offloading embedding computations to external APIs will reduce local resource demands and provide access to advanced models.

Introducing robust persistent storage with relational database integration enables survival of data across restarts, user session management, and historical query support.

Background task management through modern queue technologies will enable prioritized, retry-capable, and trackable asynchronous workflows, letting users continue working during intensive processing. Real-time progress updates via modern web protocols will improve transparency and engagement.

Comprehensive error handling, structured logging, and monitoring will provide insights into system health, enabling graceful failure recovery and proactive maintenance.

5.3.4 Cross-Cutting Benefits

Together, these enhancements deliver:

- **Enhanced user experience** with immediate feedback and non-blocking operations.
- **Increased system throughput** via parallel processing of multiple concurrent operations.
- **Optimized resource utilization** through intelligent task scheduling and externalized computation.
- **Scalable multi-user support** with isolated sessions and persistent knowledge.
- **Production readiness** supported by modern APIs, containerized deployments, and comprehensive monitoring.

These architectural transformations prepare the system for continuous extensibility, advanced AI integration including multi-modal understanding, and seamless inclusion in larger enterprise workflows, evolving it from a development tool into a sustainable knowledge platform.

6 Conclusion

This work presented a locally deployable Retrieval-Augmented Generation (RAG) system designed for secure, document-grounded conversational access to heterogeneous enterprise content. The system integrates (i) a modular client–server architecture with strict credential isolation, (ii) hybrid retrieval (BM25 + dense embeddings) with empirical weight selection, (iii) multi-format ingestion (PDF / CSV / JSON), (iv) GPU–accelerated embedding and caching, and (v) a mixed quantitative + LLM-as-Judge evaluation protocol for both retrieval effectiveness and answer reliability.

Key Outcomes. A systematic 10-step weighting sweep across SQuAD, MS MARCO, and Natural Questions identified a 30% sparse / 70% dense hybrid configuration as the preferred balance. Representative results: SQuAD (MRR 0.805, Recall@10 0.974); MS MARCO (MRR@10 0.250, Recall@1000 0.62); Natural Questions (MRR 0.813, answer coverage 0.987). Qualitative robustness was supported by low hallucination rates (0.8% SQuAD; 6.2% MS MARCO) with high faithfulness ($\geq 4.79/5$) and confidence scores. GPU acceleration yielded a measured $4.2\times$ throughput improvement for batched embedding over CPU.

Contributions. (1) An extensible, security-aware architecture separating interaction, orchestration, and retrieval layers. (2) A reproducible hybrid retrieval evaluation framework with weight sweeps and multi-dataset benchmarking. (3) A reliability layer combining deterministic retrieval metrics (MRR, Recall@K, coverage) with structured LLM-as-Judge hallucination auditing. (4) An implementation blueprint (caching, format loaders, fallback pathways) suitable for constrained local environments.

Limitations. Current processing remains synchronous, impacting latency under concurrent load and large ingests; reliance on external LLM APIs introduces privacy and availability trade-offs; scaling beyond a single node will require sharding and distributed vector storage; qualitative evaluation still depends on a single judge model and fixed prompt.

Future Work. Planned extensions include: asynchronous ingestion and background indexing; multi-judge and disagreement-aware faithfulness assessment; distributed vector backends; advanced query reformulation; incremental re-indexing and drift monitoring; extended multi-modal and code-aware loaders.

Impact. The results indicate that a resource-conscious hybrid retriever with principled evaluation can achieve high recall and low hallucination while remaining locally controllable. This supports deployment in privacy-sensitive settings and establishes a foundation for further research into scalable, governance-aligned RAG pipelines.

Personal Reflection

This project was undertaken during a period of exceptionally rapid progress in AI research, where monthly advancements reshape prevailing assumptions about capability, workflow design, and system governance. Working on a locally controlled Retrieval-Augmented Generation system clarified how architectural rigor, rather than model novelty alone, determines reliability and extensibility.

An initial motivation was exploring how such an infrastructure could evolve toward serving as a foundation for "digital twin" style organizational knowledge interfaces—systems capable of continuously ingesting heterogeneous, semi-structured internal documents while preserving privacy and traceability. The heterogeneous nature of real-world data (PDF, tabular, nested JSON) reinforced that robust preprocessing, normalization, and chunking strategies are as decisive as retrieval model selection.

The most challenging aspect was not implementing individual components (retrieval, embedding, prompt construction), but orchestrating them into a cohesive, debuggable pipeline with deterministic behavior under changing loads. AI-assisted tooling (for linting, refactoring suggestions, and architectural pattern validation) accelerated convergence toward a maintainable structure and encouraged adoption of best practices (caching boundaries, error classification, test scaffolding).

A key personal outcome was recognizing that system reliability emerged from disciplined iteration—instrumenting failure cases (mis-ranked passages, low-faithfulness generations), tightening evaluation loops, and enforcing reproducibility (seed control, cached indices, logged configuration states)—rather than from one-off "clever" optimizations. This reinforced a pragmatic engineering mindset: measurable stability over speculative complexity.

The work also prompted reflection on perceived constraints. The progression from exploratory scripts to a modular platform suggested that many limiting assumptions (scale prerequisites, dependency on proprietary hosted infrastructure, or presumed infeasibility of local evaluation workflows) are surmountable with structured decomposition and incremental validation. While future commercialization or deployment as an internal service is conceivable (e.g., for debugging knowledge flows or controlled document intelligence), those directions require further work on multi-user governance, auditability, and resilience under sustained concurrency.

Ultimately, this project strengthened technical depth in retrieval evaluation, hybrid strategy tuning, and secure local orchestration. More broadly, it demonstrated that extending system capability within defined ethical, privacy, and maintainability constraints is both feasible and professionally formative. The experience will inform continued work on asynchronous ingestion, distributed indexing, multi-judge reliability assessment, and policy-aligned adaptation—areas identified as necessary to evolve this prototype into a production-grade knowledge access layer.

References

[1] Three Drawbacks to Keyword Searches for E-Discovery, JD Supra. Available: https://www.jdsupra.com/legalnews/three-drawbacks-to-keyword-searches-for-30010/

[2] White Paper: e-Discovery and Keyword Search, FTI Technology. Available: https://static2.ftitechnology.com/docs/white-papers/white-paper-ediscovery-keyword-search-2009.pdf

[3] Keyword Search Tip Sheet, Copyright Clearance Center. Available: https://www.copyright.com/crc/wp-content/uploads/sites/2/CCC_Keyword-Search-Tip-Sheet_FNL_WEB.pdf

[4] Privacy International. Large Language Models and Data Protection. Available: http://privacyinternational.org/explainer/5353/large-language-models-and-data-protection

[5] European Data Protection Board. AI Privacy Risks and Mitigations in LLMs. Available: https://www.edpb.europa.eu/system/files/2025-04/ai-privacy-risks-and-mitigations-in-llms.pdf

[6] S. K. Saha et al., "Large Language Models and Data Privacy: Risks and Solutions," Patterns, vol. 6, no. 4, 2025. Available: https://www.sciencedirect.com/science/article/pii/S2667295225000042

[7] Understanding the Limitations of Keyword Search. Available: https://studylib.net/doc/18250247/understanding-the-limitations-of-keyword-search

[8] Rangan, C. P. Discovery of Related Terms in a corpus using Reflective Random Indexing. Available: http://users.umiacs.umd.edu/~oard/desi4/papers/rangan.pdf

[9] Y. Liu et al., "Evaluating the Faithfulness of Large Language Models in Retrieval-Augmented Generation," arXiv preprint arXiv:2312.10997, 2023. Available: https://arxiv.org/abs/2312.10997

[10] Anthropic. Model Context Protocol. Available: https://www.anthropic.com/news/model-context-protocol

[11] Phil Schmid. Introduction to Model Context Protocol (MCP). Available: https://www.philschmid.de/mcp-introduction

[12] C. D. Manning, P. Raghavan, and H. Schütze, *Introduction to Information Retrieval*. Cambridge University Press, 2008. Available: https://nlp.stanford.edu/IR-book/

[13] P. Lewis, E. Perez, A. Piktus, F. Petroni, V. Karpukhin, N. Goyal, H. Küttler, M. Lewis, W. Yih, T. Rocktäschel, S. Riedel, and D. Kiela, "Retrieval-Augmented Generation for Knowledge-Intensive NLP Tasks," in Advances in Neural Information Processing Systems, vol. 33, 2020.

[14] G. Salton, A. Wong, and C. S. Yang, "A vector space model for automatic indexing," *Communications of the ACM*, vol. 18, no. 11, pp. 613-620, 1975.

[15] S. E. Robertson and K. Sparck Jones, "Relevance Weighting of Search Terms," *Journal of the American Society for Information Science*, vol. 27, no. 3, pp. 129-146, 1976.

[16] V. Sanh, L. Debut, J. Chaumond, and T. Wolf, "DistilBERT, a distilled version of BERT: smaller, faster, cheaper and lighter," arXiv preprint arXiv:1910.01108, 2019.

[17] L. Wang, N. Yang, X. Huang, B. Jiao, L. Yang, D. Jiang, R. Majumder, and F. Wei, "Text Embeddings by Weakly-Supervised Contrastive Pre-training," arXiv preprint arXiv:2212.03533, 2022.

[18] J. Johnson, M. Douze, and H. Jégou, "Billion-scale similarity search with GPUs," *IEEE Transactions on Big Data*, vol. 7, no. 3, pp. 535-547, 2019.

[19] AWS. How RAG and MCP Solve Model Limitations Differently. Available: https://dev.to/aws/how-rag-mcp-solve-model-limitations-differently-pjm

[20] Permit.io. Building AI Applications with Enterprise-Grade Security Using FGA and RAG. Available: https://www.permit.io/blog/building-ai-applications-with-enterprise-grade-security-using-fga-and-rag

[21] Y. Zhang et al., "Enterprise-Grade Retrieval-Augmented Generation: Security, Privacy, and Compliance," arXiv preprint arXiv:2407.07858, 2024. Available: https://arxiv.org/html/2407.07858v1

[22] NVIDIA. Mastering Enterprise Chatbots: NVIDIA's Guide to Building Secure RAG-Based Chatbots with Generative AI. Available: https://syncedreview.com/2024/07/12/mastering-enterprise-chatbots-nvidias-guide-to-building-secure-rag-based-chatbots-with-generativ

[23] Galileo. Mastering RAG: How to Architect an Enterprise RAG System. Available: https://www.galileo.ai/blog/mastering-rag-how-to-architect-an-enterprise-rag-system

[24] Protecto. Understanding LLM Evaluation Metrics for Better RAG Performance. Available: https://www.protecto.ai/blog/understanding-llm-evaluation-metrics-for-better-rag-performance/

[25] Confident AI. LLM Chatbot Evaluation Explained: Top Chatbot Evaluation Metrics and Testing Techniques. Available: https://www.confident-ai.com/blog/llm-chatbot-evaluation-explained-top-chatbot-evaluation-metrics-and-testing-techniques

[26] R. Pytel. How Can You Evaluate Your Chatbot's Answers? Medium. Available: https://medium.com/@rafal.pytel/how-can-you-evaluate-your-chatbots-answers-695bd8be7f5c

[27] Stack Overflow Blog. "Retrieval-Augmented Generation: Keeping LLMs Relevant and Current." (2023). Available: https://stackoverflow.blog/2023/10/18/retrieval-augmented-generation-keeping-llms-relevant-and-current/

[28] Model Context Protocol. "Introduction." Available: https://modelcontextprotocol.io/introduction

[29] Python Software Foundation. "socket — Low-level networking interface." Python 3 Documentation. Available: https://docs.python.org/3/library/socket.html

[30] DigitalOcean Community. "How To Make a Web Application Using Flask in Python 3." Available: https://www.digitalocean.com/community/tutorials/how-to-make-a-web-application-using-flask-in-python-3

[31] Model Context Protocol Resources. "MCP Server Development Guide." GitHub. Available: https://github.com/cyanheads/model-context-protocol-resources/blob/main/guides/mcp-server-development-guide.md

[32] GeeksforGeeks. "What is Transmission Control Protocol (TCP)?". Available: https://www.geeksforgeeks.org/what-is-transmission-control-protocol-tcp/

[33] Damil Shahzad. "Unleashing the Power of Flask: A Guide to Building Web Ap-plications with Python." DEV Community. Available: https://dev.to/damilshahzad7/unleashing-the-power-of-flask-a-guide-to-building-web-applications-with-python-8g8

[34] Python Software Foundation. "Socket Programming HOWTO." Python 3 Documentation. Available: https://docs.python.org/3/howto/sockets.html

[35] "Retrieval Augmented Generation for Large Language Models: A Survey," arXiv preprint arXiv:2312.10997, 2024. Available: https://arxiv.org/html/2410.15944v1

[36] Protecto. "Secure API Management for LLM-Based Services." Available: https://www.protecto.ai/blog/secure-api-management-llm-based-services/

[37] MoldStud. "The Importance of HTML, CSS, and JavaScript in Web Development." Available: https://moldstud.com/articles/p-the-importance-of-html-css-and-javascript-in-web-development

[38] W. Lin et al., "A Survey on Retrieval-Augmented Generation for Large Language Models," arXiv preprint arXiv:2410.12837, 2024. Available: https://arxiv.org/abs/2410.12837

[39] LangChain.js. "Introduction to LangChain.js." Available: https://js.langchain.com/docs/introduction

[40] S. Wang et al., "Understanding Retrieval Augmentation for Long-Form Question Answering," arXiv preprint arXiv:2404.07220, 2024. Available: https://arxiv.org/abs/2404.07220

[41] L. Zhang et al., "Benchmarking Large Language Models in Retrieval-Augmented Generation," arXiv preprint arXiv:2312.10997, 2023. Available: https://arxiv.org/abs/2312.10997

[42] PyTorch. "CUDA semantics." PyTorch Documentation. Available: https://docs.pytorch.org/docs/stable/cuda.html

[43] LangChain. "RunnableWithFallbacks." LangChain Python API Reference. Available: https://python.langchain.com/api_reference/core/runnables/langchain_core.runnables.fallbacks.RunnableWithFallbacks.html

[44] Pinecone. "Offline Evaluation for RAG Applications." Pinecone Learning Center. Available: https://www.pinecone.io/learn/offline-evaluation/

[45] Unite.AI. "LLM as a Judge: A Scalable Solution for Evaluating Language Models Using Language Models." Available: https://www.unite.ai/llm-as-a-judge-a-scalable-solution-for-evaluating-language-models-using-language-models/

[46] P. Rajpurkar, J. Zhang, K. Lopyrev, and P. Liang, "SQuAD: 100,000+ Questions for Machine Comprehension of Text," in Proceedings of the 2016 Conference on Empirical Methods in Natural Language Processing, 2016, pp. 2383-2392. Available: https://arxiv.org/abs/1606.05250

[47] T. Kwiatkowski, J. Palomaki, O. Redfield, M. Collins, A. Parikh, C. Alberti, D. Epstein, I. Polosukhin, J. Devlin, K. Lee, K. Toutanova, L. Jones, M. Kelcey, M.-W. Chang, A. M. Dai, J. Uszkoreit, Q. Le, and S. Petrov, "Natural Questions: A Benchmark for Question Answering Research," Transactions of the Association for Computational Linguistics, vol. 7, pp. 453-466, 2019. Available: https://arxiv.org/abs/1901.08634

[48] P. Bajaj, D. Campos, N. Craswell, L. Deng, J. Gao, X. Liu, R. Majumder, A. McNamara, B. Mitra, T. Nguyen, M. Rosenberg, X. Song, A. Stoica, S. Tiwary, and T. Wang, "MS MARCO: A Human Generated MAchine Reading COmprehension Dataset," in Proceedings of the Workshop on Cognitive Computation: Integrating neural and symbolic approaches, 2016. Available: https://arxiv.org/abs/1611.09268

[49] J. Rogers et al., "Accelerating Text Embeddings with GPU Computing for Large-Scale Information Retrieval," arXiv preprint arXiv:2103.00020, 2021. Available: https://arxiv.org/abs/2103.00020

[50] Nirant Kasliwal. "RAG Metrics for Technical Leaders." Available: https://nirantk.com/writing/rag-metrics-for-technical-leaders/

[51] Spot Intelligence. "Mean Reciprocal Rank (MRR): Definition, Formula, and Applications." Available: https://spotintelligence.com/2024/08/02/mean-reciprocal-rank-mrr/

[52] A. Kumar et al., "Comprehensive Evaluation of Retrieval-Augmented Generation Systems," arXiv preprint arXiv:2410.20381, 2024. Available: https://arxiv.org/html/2410.20381v1

[53] BAAI. "BGE: BAAI General Embedding." Available: https://github.com/FlagOpen/FlagEmbedding

[54] N. Reimers and I. Gurevych, "Sentence-BERT: Sentence Embeddings using Siamese BERT-Networks," in Proceedings of the 2019 Conference on Empirical Methods in Natural Language Processing and the 9th International Joint Conference on Natural Language Processing (EMNLP-IJCNLP), 2019, pp. 3982-3992. Available: https://arxiv.org/abs/1908.10084

[55] LangChain. "LangChain: Building Applications with Language Models." Available: https://python.langchain.com/

[56] Tom Kwiatkowski, Jennimaria Palomaki, Olivia Redfield, Michael Collins, Ankur Parikh, Chris Alberti, Danielle Epstein, Illia Polosukhin, Jacob Devlin, Kenton Lee, Kristina Toutanova, Llion Jones, Matthew Kelcey, Ming-Wei Chang, Andrew M. Dai, Jakob Uszkoreit, Quoc Le, and Slav Petrov. "Natural Questions: A Benchmark for Question Answering Research." Transactions of the Association for Computational Linguistics, vol. 7, pp. 453–466, 2019. Available: https://aclanthology.org/anthology-files/pdf/Q/Q19/Q19-1026.pdf

[57] Y. Liu et al., "Faithfulness and Faithfulness Evaluation in Retrieval-Augmented Generation," OpenReview, 2024. Available: https://openreview.net/forum?id=ztzZDzgfrh

[58] AI Multiple. "LLM Latency Benchmark: Comparing Response Times of Leading Models." Available: https://research.aimultiple.com/llm-latency-benchmark/

[59] Binadox. "LLM API Pricing Comparison 2025: Complete Cost Analysis Guide." Available: https://www.binadox.com/blog/llm-api-pricing-comparison-2025-complete-cost-analysis-guide/

[60] Orq.ai. "Understanding API Rate Limits: Best Practices and Implementation Strategies." Available: https://orq.ai/blog/api-rate-limit

[61] arXiv. "arXiv:2405.07437 (v2)." 2024. Available: https://arxiv.org/html/2405.07437v2

[62] APXML. "Scaling RAG: Bottlenecks & Limitations." Course: Large-Scale Distributed RAG — Scalable RAG Architectures (Foundations). Available: https://apxml.com/courses/large-scale-distributed-rag/chapter-1-scalable-rag-architectures-foundations/scaling-rag-bottlenecks-limitations

[63] Coralogix. "The Security Risks of Using LLMs in Enterprise Applications." Available: https://coralogix.com/ai-blog/the-security-risks-of-using-llms-in-enterprise-applications/

[64] arXiv. "arXiv:2505.22852." 2025. Available: https://arxiv.org/pdf/2505.22852.pdf