Retrieval Augmented Generation (RAG) for Space Mission Design - A Space Mission Design Assistant

by

Emil Ares

BA, University of Cambridge

Submitted to the School of Aerospace, Transport and Manufacturing
in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE IN ASTRONAUTICS AND SPACE ENGINEERING

at

CRANFIELD UNIVERSITY

September 2025

Authored by: Emil Ares
School of Aerospace, Transport and Manufacturing
August 28, 2025

Certified by: Dr Nicola Garzaniti
Lecturer in Space Engineering, IRP Supervisor

IRP MEMBERS

IRP Supervisor

Dr Nicola Garzaniti
Lecturer in Space Engineering
Centre for Autonomous and Cyberphysical Systems

IRP Members

Emil Ares
Author
Student No. 454676

Retrieval Augmented Generation (RAG) for Space Mission Design - A Space Mission Design Assistant

by

Emil Ares

Submitted to the School of Aerospace, Transport and Manufacturing
on August 28, 2025 in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE IN ASTRONAUTICS AND SPACE ENGINEERING

ABSTRACT

Early-phase space mission design is characterised by vast design tradespaces and significant uncertainty, where effective retrieval of historical knowledge is crucial yet often inefficient, even with the adoption of Model-Based Systems Engineering (MBSE). This paper presents a novel Retrieval-Augmented Generation (RAG) framework to streamline this process by leveraging a Large Language Model (LLM) as an intelligent design assistant.

The system integrates a curated knowledge base derived from the European Space Agency (ESA)'s eoPortal archive with a LlamaIndex-based RAG pipeline. The methodology involved developing a high-throughput scraper (achieving a $> 4\times$ speedup and 99.9% data acquisition success), robust data preprocessing (including Markdown-based structured data conversion), and a comprehensive evaluation via a parameter sweep against a synthetically generated benchmark. Results demonstrate a highly effective system with high retrieval accuracy (Mean Average Precision (MAP@k) ≈ 0.95) and strong generation quality (F1-score ≈ 0.75). The optimal configuration (top_k=5, similarity_threshold=0.5, temperature=0.1) provides faithful and relevant answers to complex technical queries within acceptable latency (median 2.0 seconds), verified through qualitative analysis of a Streamlit prototype. This framework significantly enhances data-driven decision-making in early-phase space mission analysis, laying a robust foundation for future, deeper integration with MBSE workflows.

IRP supervisor: Dr Nicola Garzaniti

Title: Lecturer in Space Engineering

Acknowledgments

First and foremost, I would like to thank my family for the encouragement, patience, and support have been the strength of this project. For believing in me, above all when I could not believe in myself, I thank you.

To my friends, thank you for the countless conversations, study sessions, cooperation and guidance.

A special thanks goes to Dr. Nicola Garzaniti for their invaluable guidance and mentorship.

I would also like to acknowledge the role of Large Language Models in this research, not only as a core component of the RAG assistant but also as an indispensable tool in generating the synthetic dataset and performing the LLM-as-a-judge evaluation tasks.

Contents

Acronyms and Abbreviations

ANN Approximate Nearest Neighbours. 15, 16, 17, 34

BM25 Best Matching 25. 11, 12, 14

ECSS European Cooperation for Space Standardisation. 3, 64

EM Exact Match. 23, 79

EO Earth Observation. 1, 4, 30

ESA European Space Agency. II, X, 1, 2, 6, 29, 30, 35, 45, 63, 67, 71

FMEA Failure Modes and Effects Analysis. 6

FTA Fault Tree Analysis. 6

HNSW Hierarchical Navigable Small World. 15, 16, 75, 76

IDF Inverse Document Frequency. 11

ISS International Space Station. 9

JPL Jet Propulsion Laboratory. 6

LLM Large Language Model. II, IX, X, 1, 2, 4, 6, 8, 9, 10, 11, 13, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 26, 27, 28, 32, 33, 34, 35, 37, 38, 39, 40, 59, 60, 62, 63, 64, 65, 67, 76, 77, 78, 79

MAP@k Mean Average Precision. II, X, 14, 15, 39, 56, 57, 60, 61, 62, 63, 67, 79

MBSE Model-Based Systems Engineering. II, 4, 6, 7, 8, 9, 27, 28, 63, 64, 65, 67, 68

MRR Mean Reciprocal Rank. X, 15, 39, 57, 60, 79

NASA National Aeronautics and Space Administration. X, 6, 8, 9, 45

NLP Natural Language Processing. 40

RAG Retrieval-Augmented Generation. II, IX, X, XII, 1, 2, 3, 4, 5, 6, 9, 10, 11, 13, 15, 16, 17, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30, 32, 33, 35, 37, 38, 39, 40, 41, 44, 45, 47, 48, 49, 50, 51, 52, 55, 56, 60, 61, 63, 64, 65, 66, 67, 68, 78

RAGAS Retrieval Augmented Generation Assessment Scores. X, 24, 40, 60, 79

RLHF Reinforcement Learning from Human Feedback. 65

SysML Systems Modelling Language. 3, 6, 7, 8, 9, 28, 64

TF Term Frequency. 11

TF-IDF Term Frequency–Inverse Document Frequency. 11, 12, 13

UML Unified Modeling Language. 7

V&V Verification & Validation. 7

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Background

Early-phase design of space missions involves exploring a vast tradespace of possible configurations and parameters. Designers must make numerous decisions (e.g. orbit altitude, instrument payload, launch vehicle) in the conceptual phase, with limited information and under uncertainty. The number of alternatives can be enormous, and evaluating all options exhaustively is infeasible. This complexity is compounded by the need to balance competing objectives (such as performance, cost, and risk) with incomplete knowledge of how past missions succeeded or failed. Consequently, engineers often rely on prior mission data, experience, and heuristics to prune the tradespace to a manageable size.

Leveraging historical knowledge is crucial in this context. Many organisations maintain archives of past missions – for example, the European Space Agency (ESA)'s eoPortal directory provides detailed articles on hundreds of Earth Observation (EO) satellite missions from various agencies as well as other space missions [1]. Such repositories contain valuable lessons and technical specifications that could guide new designs. In practice, however, retrieving specific insights from large document archives is time-consuming. Designers may manually search reports or online databases, which is inefficient and risks missing relevant information. There is a clear opportunity to improve knowledge reuse during mission design by connecting engineers with the right information at the right time [2]. An intelligent assistant that can surface pertinent details from past missions would help reduce the viable design tradespace by ruling out infeasible or suboptimal options early and present key tradeoff parameters that engineers need to consider.

Recent advances in artificial intelligence provide a potential solution. In particular, Retrieval-Augmented Generation (RAG) has emerged as a powerful technique for combining Large Language Models (LLMs) with domain knowledge bases [3]. RAG involves using an LLM (such as GPT-4o) to generate text, but augmenting its input with relevant documents fetched

from an external corpus (non-parametric memory). By grounding the generative model with factual references, RAG can produce more informed and accurate responses [3]. This approach addresses a key limitation of stand-alone LLMs, which often cannot recall specific technical facts stored outside their trained parameters [3][4, p. 1]. In 2020, Lewis et al. demonstrated that a RAG model (a sequence-to-sequence generator backed by a dense vector index of Wikipedia) outperformed standard models on knowledge-intensive tasks [4]. The ability to cite sources and reduce hallucinations makes RAG attractive for high-stakes domains like aerospace, where trust and correctness are paramount.

Given these motivations, this project explores a RAG-based design assistant for early-phase space missions. The assistant integrates a state-of-the-art LLM with a custom knowledge base of previously flown space mission descriptions. By asking the assistant questions (e.g. "What orbit regimes have been used for SAR imaging satellites?"), a designer can quickly retrieve and summarise relevant prior mission data. This targeted guidance helps narrow the tradespace. For instance, if the knowledge base shows that certain orbits or technologies consistently led to issues for similar missions, those options can be pruned from consideration. The ultimate vision is that such a tool will accelerate the concurrent engineering process, allowing teams to iterate on mission concepts more efficiently and with greater confidence in the historical basis of their decisions.

1.2 Aim of the Project

This Individual Research Project aims to develop and evaluate a Retrieval-Augmented Generation (RAG) design assistant that supports early-phase space mission design by providing relevant information from past missions, thereby reducing the effective tradespace. In essence, the project seeks to harness Large Language Models (LLMs), in combination with a curated aerospace knowledge base, to guide mission engineers toward more informed decisions at the conceptual design stage. The assistant will be implemented using OpenAI's GPT-4o architecture for text generation and an OpenAI embedding model for information retrieval, backed by a vector database (ChromaDB) and the LlamaIndex framework. By integrating these technologies, the system should be capable of answering mission design queries with factual evidence from prior missions, ultimately helping to eliminate impractical options early and focus on promising design choices.

1.3 Objectives

To achieve the above aim, the project is structured around the following objectives

- **Knowledge Base Construction**

  Compile a domain-specific textual corpus of space mission knowledge, primarily from ESA's eoPortal mission archive (with entries covering hundreds of past, current, and

planned missions [1]), and prepare it for use in the assistant. This involves data scraping, cleaning, and organisation of mission descriptions and technical details.

- **RAG Pipeline Implementation**

  Design and implement a Retrieval-Augmented Generation (RAG) pipeline. This includes generating semantic embeddings of the knowledge base documents, indexing them in a vector database, and integrating with a generative model (GPT-4o-based) such that the model can condition its outputs on retrieved documents.

- **System Architecture & Integration**

  Develop a coherent system architecture using LlamaIndex and ChromaDB that ties together the components – document parser, embedding generator, vector index, retriever, and language model – into a working design assistant. Ensure that queries trigger the end-to-end process of retrieval and generation seamlessly.

- **Prototype Deployment**

  Deploy the assistant in an accessible form (e.g. an interactive console application or simple web interface). The deployment should demonstrate how a user (mission designer) can input questions and receive answers with supporting context from the knowledge base.

- **Evaluation of Efficacy**

  Define and execute an evaluation framework to assess the assistant's performance. This may include testing the accuracy and relevance of retrieved information, the correctness and clarity of generated answers, and the system's impact on reducing the tradespace (qualitatively, through case studies or example scenarios). Identify limitations and areas for improvement.

- **Scope for Expansion**

  Investigate the integration of additional knowledge sources such as European Cooperation for Space Standardisation (ECSS) standards and SysML documentation into the assistant. While full integration is left as future work, we should outline how the inclusion of standards and requirements documentation would enhance the assistant's capabilities (for example, by informing design decisions with established best practices and constraints).

1.4 Thesis Structure

This thesis is organised into five chapters, each addressing a key aspect of the research project.

Chapter 1 provides an introduction to the research, outlining the background and motivation for developing an intelligent design assistant for space missions. It defines the project's aim, specific objectives, and discusses the scope and limitations of the work.

Chapter 2 presents a comprehensive literature review. It begins by examining the current landscape of space mission design, focusing on Model-Based Systems Engineering (MBSE). It then explores how Large Language Models (LLMs), and specifically the Retrieval-Augmented Generation (RAG) framework, can address the existing challenges in knowledge retrieval, thereby establishing the theoretical foundation for this project.

Chapter 3 details the methodology employed to build and evaluate the RAG assistant. This includes the processes for knowledge base construction through web scraping and data preprocessing, the implementation of the RAG pipeline using LlamaIndex, the deployment of a user-facing prototype, and the design of a robust evaluation framework.

Chapter 4 presents the results of the project. It analyses the performance of the data acquisition process, provides a detailed characterisation of the resulting knowledge base, and presents both qualitative and quantitative evaluations of the RAG assistant's performance. The chapter culminates in a multi-objective parameter optimisation to identify the ideal system configuration.

Chapter 5 concludes the thesis by summarising the key findings and achievements of the project. It reflects on how the objectives were met and discusses promising avenues for future work, outlining a roadmap for deeper integration of AI tools into systems engineering workflows.

Finally, the appendices provide supplementary materials, including an executive summary, ethical clearance documentation, detailed descriptions of the code, and additional evaluation results.

1.5 Scope and Limitations

This project focuses on early-phase mission design assistance with the knowledge base scoped to missions in the eoPortal archive, which consists primarily of Earth Observation (EO) missions (699 EO missions versus 393 other space missions). The current implementation covers textual descriptions of satellite missions (mission objectives, instruments, orbits, launch dates, etc.), and includes less about other mission types (e.g. communications, human spaceflight) as well as less detailed numerical databases. The rationale is that EO missions provide a rich and relatively homogeneous set of past cases to inform new mission concepts. However, this scope means the assistant may be less effective if queries fall outside the EO context or require knowledge not present in the archive.

There are a few limitations to acknowledge, namely, the completeness of the knowledge base,

real-time updating, and the scope of guidance.

- **Knowledge base completeness:** The eoPortal-derived corpus, while extensive, is not exhaustive. It contains in-depth articles on many missions, but some very recent missions or highly classified projects might be absent. Moreover, information on each mission is limited to what the eoPortal articles cover – typically high-level technical and operational details. If a query asks for data not captured in those articles (for example, extremely specific design calculations or mission anomalies), the assistant may not find a relevant answer.

- **Real-Time Updating:** The knowledge base is essentially static as of the time of scraping. Any missions launched or updated after the data collection will not be included, unless the database is manually refreshed. The assistant does not have an active internet search capability; it only knows what is in the vector index. In a dynamic mission design environment, this means the tool could lag behind the latest developments or announcements until the knowledge base is updated.

- **Scope of Guidance:** The assistant is intended to support human designers, not to autonomously design missions. It can provide relevant facts, comparisons, and suggestions (for instance, listing missions similar to a proposed concept or highlighting typical values for certain subsystems), but it does not perform simulations or detailed engineering analysis. The ultimate design decisions must still be made by experts, who use the assistant's outputs as one input among many. The tool currently cannot evaluate the feasibility of a novel mission concept beyond the context of past data; it only draws parallels and highlights information from those past examples.

1.6 Summary

In summary, this project demonstrates a proof-of-concept design assistant within a constrained domain. The limitations listed are recognised and provide directions for improvement. Despite these constraints, the core idea – using a RAG approach to cut down the early mission tradespace – is explored in depth, and the results offer insight into how such an assistant can be implemented and what value it can provide.

1.7 Code & Data Availability

The Space Mission Design Assistant codebase is publicly available on GitHub.

See Appendix E for quick-start instructions and structure (the link above contains the same instructions). Appendix C provides a more detailed breakdown of the important scripts, their inputs, methodologies, outputs and justifications. Appendix C is also used to supplement our methodology, Chapter 3.

Chapter 2

Current State of the Art & Literature

2.1 The Modern Landscape of Space Mission Design

Space missions inherently push the limits of complexity and have recently driven the shift from document-driven practices to Model-Based Systems Engineering (MBSE). In the past five years, MBSE has increasingly become standard in agencies and industry to manage system complexity and maintain a single source of truth for designs [5, p. 2]. For example, the European Space Agency (ESA) Agenda 2025 explicitly aims for full digitisation of project engineering via MBSE and even 'digital twins' of spacecraft (see Fig. 2.1) [6, p. 12]. Likewise, the National Aeronautics and Space Administration (NASA) has been standardising MBSE practices for integrating early safety requirements and analysis, including Reliability Block Diagrams, Failure Modes and Effects Analysis (FMEA) and Fault Tree Analysis (FTA) [5, p. 2]. And notably, its Jet Propulsion Laboratory (JPL) has applied MBSE on the Europa Clipper mission from its outset, using Systems Modelling Language (SysML) models to manage mass and power budgets and track requirements, yielding improved requirements management, better cross-team communication and reduced errors [7, p. 1]. Other case studies, such as ESA's EagleEye Earth observation mission, demonstrate that MBSE can significantly enhance efficiency in complex projects [5].

Concurrently, advances in AI, particularly Large Language Models (LLMs) such as OpenAI's ChatGPT and Google's Gemini models, are opening new frontiers in space systems engineering. The convergence of MBSE and AI promises to boost engineering productivity by automating tedious tasks and enabling more rapid, informed design iterations. This literature review will first explore the current paradigm of MBSE, then examine how LLMs, and specifically Retrieval-Augmented Generation (RAG), can address its remaining challenges in knowledge integration and accessibility.

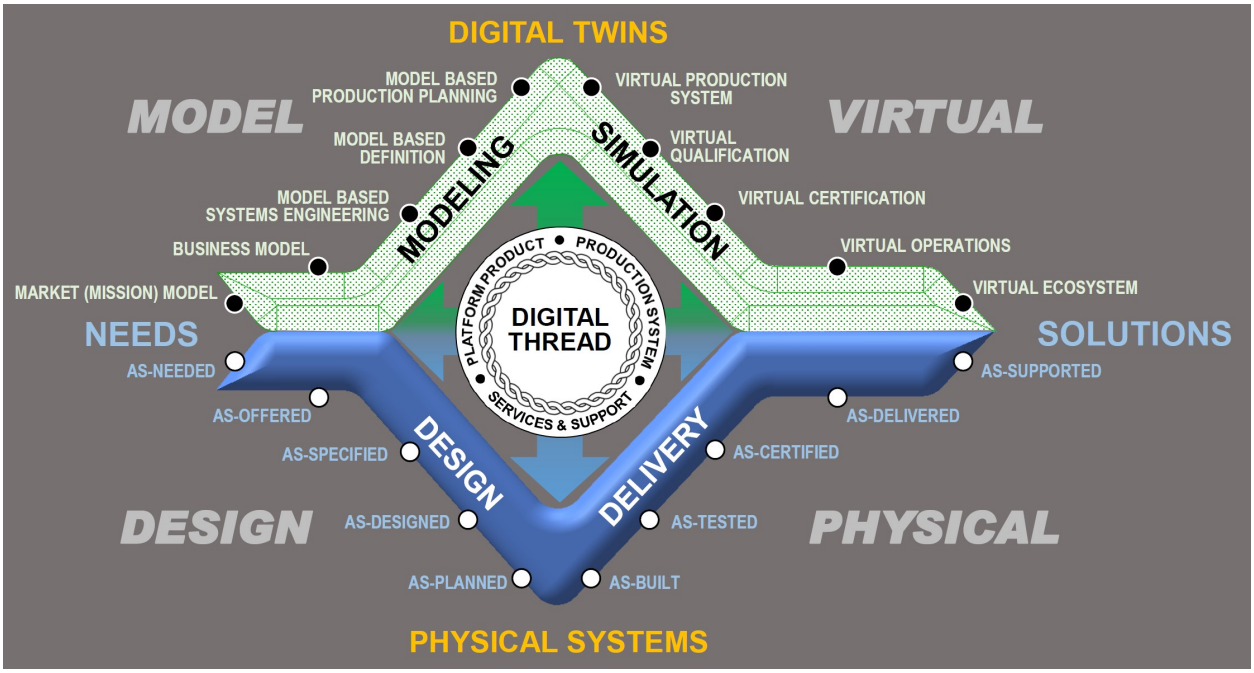

Figure 2.1: Overview of a 'digital twin' model. The diamond's top half is the physical system's virtual representation. The diamond centre represents the 'digital thread', which links the model to the design of the physical system. The bottom half represents the traditional V-Model [8, p. 6]

2.2 Model-Based Systems Engineering (MBSE): The Current Paradigm

MBSE refers to the formalised application of modelling to support system requirements, design, analysis, verification and validation throughout the life cycle. It is now widely regarded as state-of-the-art for complex mission development, becoming the primary method for developing complex systems in the space and aerospace sectors [5, p. 2]. Traditional document-based approaches struggle with consistency and traceability; MBSE addresses this by capturing the design in a rigorous model that ensures consistent communication across interdisciplinary teams [5, p. 2]. By integrating requirements, design, and analysis in a model, inconsistencies are exposed earlier, making Verification & Validation (V&V) more systematic.

2.2.1 Systems Modelling Languages and Tools

The primary language for MBSE in space missions is Systems Modelling Language (SysML), a general-purpose language derived from Unified Modeling Language (UML) that supports the specification, analysis, and design of complex systems [9]. Its richness allows SysML to represent most aspects of a space mission in an integrated model. SysML is typically implemented in tools like Cameo Systems Modeller or Capella, supported by an ecosystem of simulation tools and model repositories like OpenMBEE [10].

2.2.2 Challenges and Opportunities in MBSE

Despite its benefits, MBSE adoption presents challenges. There is a steep learning curve for engineers, and integrating the necessary suite of tools is non-trivial [7, p. 1]. As models scale, they can become difficult to manage, and the semantic flexibility of languages like SysML can lead to misinterpretation without strict conventions.

Most importantly, while MBSE provides a structured framework for system design, it does not inherently solve the challenge of leveraging the vast corpus of unstructured knowledge from past missions. Decades of design reports, lessons-learned documents, test results, and technical standards exist outside the formal model. An engineer working within an MBSE tool still needs to manually search these external archives to inform their design decisions, a process that is time consuming and risks missing critical information. This gap between the structured model and the unstructured sea of historical knowledge presents a clear opportunity for enhancement through AI.

2.3 Bridging the Gap with Large Language Models (LLMs)

LLMs, with their ability to understand and generate human-like text, are emerging as a key technology to bridge the gap between formal models and unstructured domain knowledge. Several promising applications in the space mission domain are currently being researched:

Generative MBSE Architecture Synthesis

Researchers have started to use LLMs to assist in the architectural design of space systems, for example, by coupling a GPT-3 model with the Capella MBSE tool to generate elements of a spacecraft architecture from textual requirements [11]. This suggests LLMs could become interactive assistants for mission architects, though the capability is still nascent.

Autonomous Mission Operations

Exploratory work is underway to investigate an LLM's ability to operate a spacecraft, with GPT-4 being used to control a simulated spacecraft by interpreting telemetry and issuing commands [12]. However, applying this to real flight-critical systems remains a distant prospect.

Documentation Q&A and Requirements Engineering

A more immediate and robust application is using LLMs for intelligent documentation analysis. Space missions generate enormous quantities of text, and LLMs excel at ingesting and retrieving information from these large databases. NASA's development of NASA-GPT, an internal chatbot for querying technical reports, exemplifies this use case [13]. Similarly, LLMs can be trained to classify and extract requirements from specification documents [14].

Among these candidate use-cases, a documentation Q&A assistant powered by the Retrieval-Augmented Generation (RAG) approach is the most mature and attractive. It directly addresses the knowledge retrieval challenge identified in MBSE workflows. As demonstrated by NASA and by contractors on the International Space Station (ISS) [15], this approach turns an LLM into an intelligent search and Q&A tool that can surface critical information from authoritative programme knowledge. It represents a practical and powerful first step, forming a natural stepping-stone to more advanced capabilities like automated SysML code generation.

2.4 A Focused Solution: Retrieval-Augmented Generation (RAG)

Having identified a RAG-powered assistant as a direct and impactful way to enhance early-phase mission design, this section provides a detailed review of the theory, architecture, production and evaluation considerations of RAG systems and Large Language Models (LLMs).

2.4.1 RAG Pipeline Overview

Retrieval-Augmented Generation (RAG) is a technique that pairs LLMs with external knowledge sources to overcome key limitations of standalone LLMs [4, p. 1][16, p. 1]. Traditional LLMs store vast knowledge in their parameters but struggle with hallucinations (making up facts) and outdated information once deployed [16, p. 1][17]. RAG addresses this by equipping an LLM with a non-parametric memory – typically a database of documents – that can be queried in real time for relevant information [4, p. 1]. In essence, RAG "synergistically merges the LLM's intrinsic knowledge with external databases" to produce responses that are up-to-date, factual, and verifiable [16, p. 1]. This approach has proved especially powerful in knowledge-intensive tasks. For instance, the original 2020 RAG model (which combined a seq2seq generator [18] with a dense vector index of Wikipedia) achieved state-of-the-art results on open-domain question answering, outperforming fine-tuned parametric models and retrieve-and-read pipelines [4]. RAG generations were observed to be more specific and factual than those from an LLM without retrieval [4, p. 1], underscoring the benefit of grounding outputs in external evidence.

At the core of RAG is a retrieval pipeline that injects relevant context into the LLM's input. Fig. 2.2 illustrates a typical RAG architecture, consisting of an offline ingestion stage and an online query-response stage. During ingestion, a domain-specific corpus is processed. Documents are chunked (split into passages), and each chunk is encoded into a high-dimensional embedding vector that captures its semantic content [19]. These vectors are stored in a specialised vector database (or index) that enables fast similarity search [19]. Common implementations use cosine similarity or dot-product in the embedding space to

measure text similarity – documents whose vectors have a small angle (high cosine similarity) with the query vector are considered most relevant [20]. The choice of similarity metric typically aligns with the embedding model's training; for instance, models trained with a cosine objective yield the best results if cosine similarity is used for retrieval [20], as opposed to keyword search, which just searches for exactly matching words and not similar meaning.

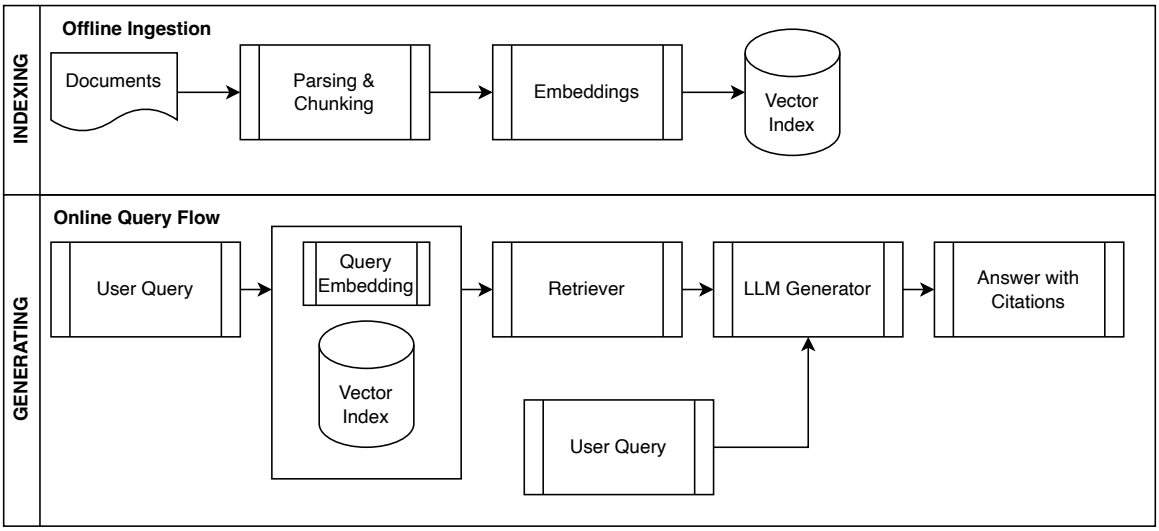

Figure 2.2: Overview of a RAG pipeline architecture, showing the offline document ingestion (data sources are chunked, embedded, and indexed in a vector database) and the online query flow (a user query is embedded, top-k similar documents are retrieved, and passed as context to the LLM which generates the final answer). The RAG system thus consists of a retriever (knowledge search) and a generator (LLM) working in tandem.

When a user poses a query, the system embeds the query into the same vector space and retrieves the top-k[1] document chunks most similar to the query embedding [21][22]. These retrieved texts (context passages) are then integrated into the LLM's prompt, usually by concatenating them with the user query and a suitable prompt template [22]. Finally, the generator (the LLM, e.g. GPT-4o) processes this augmented prompt and produces a response that draws on the provided context. Formally, if q is the query and $D = d_1, \ldots, d_k$ are the retrieved documents, the LLM generates an answer A approximating $A = \text{LLM}(q, d_1 \ldots d_k)$. By conditioning on external text d_i, the LLM is grounded and less likely to fill knowledge gaps with fabricated information [4, p. 10][21]. The quality of the final answer thus heavily depends on the retriever fetching relevant and sufficient information [21]. Effectively, "the overall generation is only as strong as its weakest component – retriever or generator" [21]. This has driven the development of robust retrieval techniques (e.g. hybrid keyword and semantic search, iterative retrieval) to ensure the LLM has the right context [23]. In early-phase

[1]Given a set of items that each have a score (e.g. similarity to a query), top-k means return the k items with the highest scores.

space mission design, such a RAG-based assistant can rapidly surface design options and constraints from prior missions, effectively serving as an intelligent knowledge concierge. Instead of manually combing through archives, an engineer's natural-language query (e.g. "What orbits have been used for SAR imaging satellites?") can be answered with relevant facts from mission databases, allowing quick elimination of infeasible concepts and substantiation of design decisions.

2.4.2 Foundational Retrieval Theory

A retriever in a RAG system is responsible for finding relevant knowledge snippets that can answer the user's query. Its performance is critical – if the retriever fails to fetch the right information, the LLM has little chance of producing a good answer [24]. Modern retrieval techniques can be broadly divided into keyword-based search and semantic search, each with strengths and limitations. In practice, RAG systems often employ a combination of both, possibly with metadata filters to scope the search.

Keyword Search

Keyword search is a method that retrieves documents by matching words in the prompt with words in the documents [25]. It effectively relies on direct keyword matching using techniques like Term Frequency–Inverse Document Frequency (TF-IDF) and Best Matching 25 (BM25)[2].

In TF-IDF, documents are scored by Term Frequency (TF) of query words weighted by Inverse Document Frequency (IDF) to downweight common terms. The two components are defined below as TF and IDF, where t is the query term, d is the document and D is the corpus:

TF: Counts how often a term occurs within a document. Higher counts indicate that the term is more prominent and likely more relevant to the document's subject [26].

$$\mathrm{TF}(t,d) = \frac{\text{Number of times term } t \text{ appears in document } d}{\text{Total number of terms in document } d} = \frac{f_{t,d}}{|d|} \tag{2.1}$$

IDF: Assigns a lower weight to terms common across many documents and a higher weight to rare terms. Rare terms are more likely to be meaningful and specific to the topic [26].

$$\mathrm{IDF}(t,D) = \log\left(\frac{\text{Total number of documents in corpus } D}{\text{Number of documents containing term } t}\right) = \log\left(\frac{N}{\mathrm{df}_t}\right) \tag{2.2}$$

This should make intuitive sense. The expression inside the log of Eq. 2.2 suggests its value

[2]The name "BM25" comes from it being the 25th variant in a series of ranking functions developed by its creators.

(score) increases the rarer the word; however, we don't want to reward rare words overly, so we take the log.

The TF-IDF similarity score is just the product of Eq. 2.1 and Eq. 2.2

$$\text{TF-IDF}(t, d, D) \;=\; \text{TF}(t, d) \cdot \text{IDF}(t, D) \tag{2.3}$$

TF-IDF is powerful in that it differentiates between rare and commonly used words, and is a standard baseline for the performance of keyword retrieval. Top-scoring documents contain many query keywords, especially those that are rare across the entire corpus. While TF-IDF is a classic keyword search method, modern systems often adopt a refined variant known as BM25.

BM25 is a more advanced bag-of-words ranking function that builds on TF-IDF by saturating term importance and normalising for document length [27]. Eq. 2.1 and Eq. 2.2 are refined as follows:

$$\text{TF}_{\text{BM25}}(t, d) = \frac{f_{t,d}}{f_{t,d} + k_1(1 - b + b\frac{|d|}{avgdl})} \tag{2.4}$$

where k_1 is a parameter that controls the saturation (usually set between 1.2 and 2.0 [28]) and b is a parameter that controls the influence of document length (typically set to 0.75 [28]). $avgdl$ is the average document length in the corpus.

$$\text{IDF}_{\text{BM25}}(t) = \log\left(\frac{N - n_t + 0.5}{n_t + 0.5}\right) \tag{2.5}$$

where n_t is the number of documents containing term t.

BM25 down-weights matches in longer documents; $b \in [0, 1]$ controls how much. At $b = 0$ there is no length normalisation; at $b = 1$ the normalisation uses the ratio $|d|/\text{avgdl}$ directly (e.g., if $|d| = 2\,\text{avgdl}$, the length factor doubles).

Finally, the BM25 score for a document d with respect to query q is [28]:

$$\text{BM25}(q, d) = \sum_{t \in q} IDF(t) \cdot TF(t, d) \tag{2.6}$$

Intuitively, BM25 rewards documents that contain many occurrences of the query words, but with diminishing returns, and prefers documents that are not overly long. It also gives us two parameters to customise (fine-tune) as required, thereby enhancing its adaptability. This

means it tends to perform better than TF-IDF in production retrievers (including LlamaIndex which we use in Chapter 3).

Overall, keyword search methods are fast and precise for keyword-heavy queries, and they don't require model training. However, they struggle when the query uses different wording (synonyms) from the document or for conceptual queries, because they only consider exact or stemmed word matches.

Semantic Search

Semantic retrieval addresses the above issues by encoding text into high-dimensional vector embeddings, such that similar meanings lie close together in the vector space. Relevance is computed by a vector similarity score (typically cosine similarity or dot-product) rather than literal word overlap. For example, the cosine similarity between a query vector $\mathbf{q}$ and a document vector $\mathbf{d}$ is $\cos(\mathbf{q}, \mathbf{d}) = \dfrac{\mathbf{q} \cdot \mathbf{d}}{|\mathbf{q}||\mathbf{d}|}$. Documents whose embedding has a small angle to the query vector (high cosine value) are considered semantically relevant [29]. A crucial point is to align the similarity metric with the embedding model's training objective [30]. For instance, if the embedding model was trained with a cosine similarity loss, using cosine for retrieval will yield more accurate results than, say, Euclidean distance [30]. Semantic search is powerful at handling paraphrasing and conceptual similarity (e.g. query "thermal control device" can retrieve a passage about "radiators" even if that keyword isn't in the query). Its downside is that it requires a trained model and can sometimes retrieve passages that are topically similar but not specific to the exact question, especially if the model wasn't fine-tuned for the domain.

Metadata filtering

Metadata filtering is an important enhancement to both keyword and semantic search. In production RAG applications, each document or chunk often has structured metadata (e.g. document type, date, author, mission phase). The retriever can apply filters so that only chunks meeting certain criteria are searched. For example, one might restrict retrieval to documents from 2020 or a specific subsystem. This improves precision by narrowing the search space. Metadata filtering can be invoked explicitly (when the user or system adds filter conditions) or implicitly through query parsing. In the latter, the system analyses the query to infer filters – e.g. if a query mentions "Hayabusa2 mission 2014", the retriever can automatically apply filters for mission name Hayabusa2 and year 2014. Advanced RAG implementations use an LLM-based query parser to extract such intent and conditions from the query and translate them into structured search filters [31]. This "self-query" approach allows dynamic creation of metadata filters and even selection of the appropriate retrieval method based on the query [31].

Hybrid Search

Because each retrieval method has unique strengths, hybrid retrieval strategies are often used to maximise recall. A common approach is to run both a keyword search (e.g. BM25) and a vector search in parallel, then merge the results [31]. The merging can simply take the union of top-k results from each, or apply learned weighting and sorting by a combined relevance score [31]. Hybrid retrieval helps in cases where pure semantic search might miss a highly relevant document that happens to use different wording, or where pure keyword search might miss a relevant document that doesn't contain the exact query terms. Studies have shown that hybrid retrieval can substantially boost retrieval accuracy [31].

Evaluating Retrieval Quality

Once a retriever is operational, the key question is whether it's effective (search quality). We can measure latency, throughput, and resource usage, but the most important metric is search quality - how well it finds relevant documents.

There are a few ways to answer this question, but in essence, we need three quality metrics. Namely, the specific query being evaluated, the ranked results (documents returned in ranked order) and the ground truth (all documents labelled as relevant or irrelevant) [32]. The most widely used retriever quality assessors are precision (Precision@k, fraction of the top-k results that are relevant) and recall (Recall@k, the fraction of queries for which a relevant document appears in the top-k results) [32]. Put simply, they are:

$$\text{Precision} = \frac{\text{Relevant Retrieved}}{\text{Total Retrieved}} \tag{2.7}$$

and

$$\text{Recall} = \frac{\text{Relevant Retrieved}}{\text{Total Relevant}} \tag{2.8}$$

Fundamentally, Precision (Eq. 2.7) penalises a retriever for returning irrelevant documents [32] and can be thought of intuitively as capturing how trustworthy the results are. Recall (Eq. 2.8) penalises a retriever for leaving out relevant documents [32] and intuitively measures how comprehensive a retriever is. The only way to get 100% precision and 100% recall is to return exactly the relevant documents (no extras, no misses). In practice, we judge quality on the top k results. Increasing k usually raises recall but lowers precision, so we report metrics at a fixed cutoff, like Precision@k and Recall@k, computed over the top-k items the retriever ranks highest. The top five, two or one are used when we want stricter standards.

However, generally, when both precision and recall are important (e.g. the system returns many passages to the user, such as between top five and top fifteen), metrics like Mean Average Precision (MAP@k) are used to give a holistic view of the system [24].

$$\text{MAP} = \frac{\text{Sum precisions for relevant docs only}}{\text{Number of relevant docs}} \tag{2.9}$$

Eq. 2.9 rewards systems that rank relevant documents near the top. If an irrelevant document appears early in the ranking, it lowers the precision of every relevant document that follows, reducing the overall score. A high MAP@k value, therefore, shows that relevant documents are consistently being placed high in the results [32].

Another widely used metric is Mean Reciprocal Rank (MRR), which is sensitive to where the first relevant result appears in the ranking. For instance, if the correct document is ranked 1st for one query (reciprocal rank = 1), and 5th for another (reciprocal rank = 1/5), the mean of these is the MRR. High MRR means relevant info tends to appear at top ranks, which correlates with easier answer extraction by the LLM. It reflects how soon (on average) we can find a relevant document in the retriever's ranking. It also emphasises the importance of including at a minimum one relevant item as high in the rankings as possible [32] - MRR is usually calculated over many queries.

To summarise, with many metrics available, it's useful to understand how they complement each other. Recall@k is the most fundamental measure for retrievers, since it reflects the core goal - finding relevant documents. Precision and MAP@k go further by checking how many irrelevant documents are included and how well the relevant ones are ranked. MRR is more specialised, focusing on whether relevant results appear right at the top. Taken together, these metrics both evaluate a retriever's overall performance and show whether system changes are making an impact.

2.4.3 Information Retrieval with Vector Databases

With the foundational theory explained, we need to move to production. Traditional relational databases can handle many retrieval techniques, but they struggle to scale once you need to search millions or billions of documents, especially for the vector operations required in semantic search. At that point, it's more efficient to use a vector database, which is designed to store and search large volumes of vectors. Because of this, vector databases have become closely associated with RAG systems [33].

Approximate Nearest Neighbours (ANN)

Implementing semantic search at scale requires efficient Approximate Nearest Neighbours (ANN) algorithms and specialised data stores, since brute-force search through millions of embeddings is too slow. Vector databases (like ChromaDB, FAISS, Annoy, Milvus, Weaviate, etc.) are purpose-built to store high-dimensional embeddings and support fast similarity search with ANN methods. These systems index vectors in a way that accelerates retrieval, usually by trading off a tiny bit of accuracy for massive speed-ups. One popular approach, used by Facebook's FAISS library, is the Hierarchical Navigable Small World (HNSW) graph

[4, p. 4]. HNSW organises vectors as nodes in a graph where edges connect close neighbours; search traverses this graph greedily from a few entry points to find near matches in sub-linear time. Other ANN techniques include product quantisation (compressing vectors into compact codes and only comparing in a coarse quantised space) and inverted file indices (which cluster the vector space and only search within relevant clusters). The details of these algorithms are beyond our scope, but the key is that the vector index can be tuned for a balance between speed and recall – e.g. one can adjust the search depth in HNSWs or the number of clusters scanned in a quantisation index [4, p. 4]. In mission engineering terms, this means a well-configured vector DB can instantly search a knowledge base of millions of mission documents and retrieve the few most semantically relevant snippets, all within perhaps tens of milliseconds.

Chunking

Equally important is how the data is chunked and indexed. Documents must be split into passages (chunks) of an appropriate size for indexing. If chunks are too large, the embeddings may dilute important details; too small, and a single fact might be split across multiple chunks. A common strategy is to chunk by paragraphs or sections, with a token length limit (but for many missions it's better to chunk larger, e.g 1000-2000 tokens for integrity) and perhaps a sliding window overlap to avoid losing context at boundaries [34]. Chunking enables the retriever to pull back just the relevant paragraph rather than an entire document [22], which not only improves relevance but also reduces the prompt size (and thus LLM processing cost). More advanced chunking techniques include using semantic chunking – for instance, leveraging document structure (headings, XML/JSON tags) to split along logical units, or even using an LLM to decide split points (e.g. ensure a requirement statement and its rationale stay in the same chunk). Regardless of method, each chunk inherits metadata from its source (document title, date, etc.) so that this context is not lost after splitting [34].

Overall, when designing a RAG system, the aim isn't to chase the most advanced chunking method available. The real goal is to understand the options, judge which fit your data best, and weigh their costs and benefits before deciding whether to use them.

Query Parsing

A key step in production RAG systems is cleaning up user prompts. Since users typically phrase queries conversationally, these inputs often make poor search queries. Instead of sending them directly to the vector database, the retriever can parse the prompt, extract its intent, and rewrite or transform it for better retrieval. Many query parsing methods exist, but the most common and effective is simply using an LLM to rewrite the query before passing it to the retriever [35]. The rewritten query strips out unnecessary details, resolves ambiguities, and can even introduce domain-specific terminology to improve matching in the knowledge base. While it's worth iterating on the rewriting strategy, the gains in retrieval quality typically outweigh the extra cost of making an LLM call for each prompt [35].

More advanced techniques also exist, such as Named Entity Recognition, which identifies specific categories in a query - such as people, locations, dates, or even fictional characters [35]. These extracted entities can then guide the retriever, either by improving the vector search itself or by supporting metadata-based filtering later in the pipeline. Overall, query parsing is better done simply via an LLM as more complex techniques won't necessarily yield better results and are more complicated to run.

Cross-encoders, ColBERT and Reranking

Once chunks are embedded and stored in the vector DB, the query workflow involves multiple stages of refinement. First, we parse the query. Second, the query embedding is used to retrieve a set of top-k candidates via the ANN index. Often k is set a bit higher (e.g. $k = 50$) than the final number of passages we plan to feed the LLM, to ensure we have enough material to work with [4]. This initial candidate set can then be re-ranked by more precise but slower models before we pick the top few to pass to the LLM. Two popular re-ranking approaches are cross-encoders and ColBERT (a late-interaction bi-encoder) [36]. A cross-encoder takes the query and a candidate passage together as input to a transformer model and outputs a relevance score (usually via a classification head) [37]. Because the cross-encoder sees the full query–passage interaction (all tokens together), it can make very fine-grained relevance judgments – often outperforming the initial bi-encoder retriever. However, it is computationally expensive, since a BERT-sized model must run for each candidate pair. In practice, one might re-rank only the top 50 or 100 passages with a cross-encoder. By contrast, ColBERT (Collective Late Interaction BERT) offers a middle ground. In ColBERT, the query and documents are encoded independently but without condensing into a single vector – instead, each token produces an embedding [38]. During matching, ColBERT performs a late interaction: it compares the query token embeddings with document token embeddings (usually via dot products) and aggregates the scores (e.g. taking maximum similarities for each query term) [38]. This token-level matching retains more nuance than a single-vector comparison, yet is far faster than running a full cross-encoder for each document. In essence, ColBERT "bridges" bi-encoders and cross-encoders, achieving nearly the accuracy of the latter while maintaining much of the efficiency of the former [38]. Empirical studies have shown that ColBERT and similar multi-vector retrievers can significantly improve answer recall in RAG pipelines without incurring prohibitive costs [38].

In summary, a high-performing RAG retrieval system may combine multiple layers. An ANN vector search for recall, a keyword search for precision on rare keywords, and a reranker for fine-grained sorting [24]. The retriever also interfaces with other components: query parsing (to apply any filters or transformations before search), caching of past results (to avoid repeat work on popular queries), and fallbacks (for instance, if the semantic search finds nothing with high confidence, the system might try a broader keyword search). All of these measures ensure that, by the time the query and retrieved texts reach the LLM, the model is primed with the most relevant and correct information available.

2.4.4 LLMs and Text Generation

LLMs typically builds on the Transformer architecture (see Fig. 2.3), which uses multi-head self-attention to capture long-range dependencies in text [39, p. 3]. In a Transformer-based LLM, input text is first tokenised and embedded (with positional encodings), then passed through layers that allow each token to attend to others regardless of their distance [39]. Notably, decoder-only LLMs (such as GPT series) employ a causal mask in self-attention so that each position can only attend to earlier positions, ensuring that during generation, a token only depends on previously generated tokens [39]. This autoregressive setup lets the model generate text one token at a time. At each step, the decoder produces a probability distribution over the vocabulary for the next token given all prior tokens. The highest-probability token can be taken as output, or a sampling strategy can be applied to select a likely token, which is then appended to the sequence and fed back into the model for the next step. This iterative decoding process continues until an end-of-sequence token or length limit is reached.

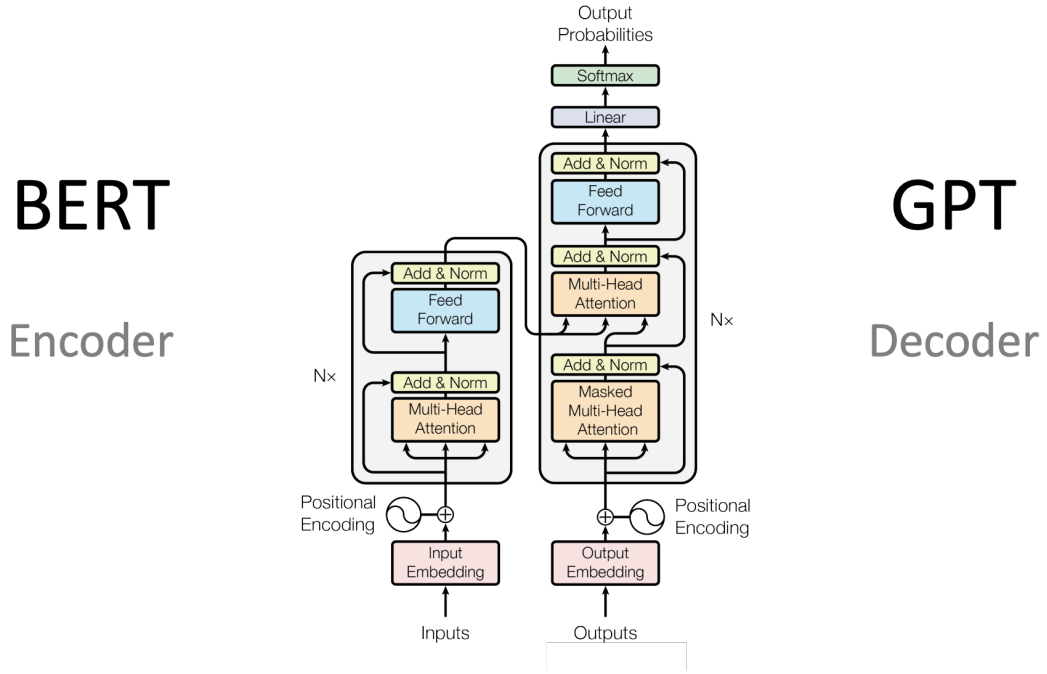

Figure 2.3: Transformer architecture, with an encoder (left) and a decoder (right), each composed of stacked self-attention and feed-forward layers. The decoder's self-attention is masked to allow autoregressive text generation.(Source: [40][39, p. 3])

Decoding Strategies and Sampling Methods

How the next token is chosen from the model's probability distribution has a major impact on the quality of generated text. A naive approach is greedy decoding, which always picks the highest-probability token at each step. Greedy decoding (or its batch variant, beam search) is deterministic and often yields grammatically correct but generic and repetitive text

[41]. Maximum-likelihood decoding methods like beam search have been observed to produce degenerate outputs – e.g. bland, over-repetitive passages – even with advanced models [41]. To avoid such issues, stochastic sampling methods inject randomness in a controlled way:

- **Top-k sampling:** Instead of considering the entire vocabulary, the model restricts its choice to the top k tokens (those with highest probability) and samples the next token from this subset. This method, introduced in early neural text generators, preserves some spontaneity while avoiding low-probability choices [41]. Capping the candidate list to k prevents the model from picking extremely unlikely tokens that could derail coherence. However, a fixed k can be suboptimal if the tail of the distribution is long or if the model's confidence varies – a large k may still include implausible tokens, whereas a small k might become too restrictive.

- **Top-p sampling:** Rather than a fixed token count, nucleus sampling dynamically chooses the smallest set of top tokens whose cumulative probability exceeds a threshold p [41][42]. The model then samples from this "nucleus" of probable tokens. This approach truncates the long tail of low-probability tokens (which is often "unreliable" or noise [41]) while adaptively widening the sample space when the distribution is uncertain [42]. By sampling from the top-p portion of the distribution, nucleus sampling maintains diversity without sacrificing fluency – it was shown to yield text that is as diverse as human writing while avoiding the incoherence of pure random sampling [41]. In fact, [41] found nucleus sampling to be the most effective decoding strategy for long-form generation, outperforming both top-k and deterministic methods in human evaluations of quality and diversity

- **Temperature:** A control parameter is the temperature T, which smooths or sharpens the probability distribution before sampling. In practice, the logits (pre-softmax scores) are divided by T; $T < 1$ makes the model more confident (increasing the gap between top probabilities and the rest), leading to more deterministic, higher-probability outputs, whereas $T > 1$ flattens the distribution, encouraging more random choices. Adjusting T allows tuning the creativity of the output. However, temperature alone does not eliminate low-probability outliers in the tail [41] - e.g. a very high T can still sample absurd tokens – so it is often used in combination with top-k or top-p truncation. The common practice in open-ended text generation is to first apply a top-k or top-p filter to ensure only likely tokens are considered, and then sample among them with an appropriate temperature to balance coherence and creativity.

In summary, careful decoding is crucial for LLM-generated text. Greedy or beam search outputs may be high-probability but can lack diversity (a phenomenon dubbed text degeneration when models get stuck in loops [41]), while purely random sampling can produce incoherent results. Methods like top-k and nucleus sampling aim to hit a sweet spot: they avoid the pitfall of always picking the most likely token (which can lead to dull, repetitious text) by allowing variability, but also avoid the pitfall of truly random generation by focusing

on the model's plausible continuations [41]. Empirical studies confirm that these strategies can dramatically affect output quality even from the same model [41]. Tuning decoding parameters is therefore an important part of using LLMs effectively, especially in applications like RAG where both factual accuracy and readability are desired.

LLM Model Choice – Capability vs. Latency and Cost

The choice of LLM can significantly influence text generation performance, response time, and deployment cost. Larger models generally produce more coherent and accurate text, but they are slower and more expensive to run. For example, GPT-4 series (OpenAI's state-of-the-art models) are renowned for their capability and fluency, consistently achieving top results on language tasks. However, they are massive proprietary models, only accessible via paid API, incur substantial computational costs, and typically have higher latency per query due to their size [43]. Moreover, being closed-source, they cannot be run on local hardware, and subtle changes between API versions can affect their outputs [43].

In mission-critical settings like space engineering, such drawbacks translate to higher operational costs and potential integration hurdles. On the other hand, smaller open-source models (on the order of billions of parameters, vs. GPT-4's hundreds of billions) offer a different trade-off. Models like Mistral 7B (a 7-billion-parameter Transformer released in 2023) sacrifice some raw capability but are optimised for efficiency. The Mistral 7B model leverages architectural improvements – e.g. grouped-query attention for faster inference and sliding window attention for long contexts – to outperform larger models (it beats the 13B-parameter LLaMA-2 on many benchmarks) while running with lower memory and latency requirements [43]. Such a model can be deployed on modest hardware, making it attractive for applications that require quick responses or on-board processing. In terms of text generation quality, open models have rapidly advanced but are still behind closed-source models. However, this narrowing gap suggests that for many applications, especially those with tight latency or budget constraints, a smaller model with domain-specific tuning can be an effective choice. Ultimately, the model selection involves balancing capability (larger models generally yield more accurate and detailed outputs) against latency and cost (smaller models are cheaper to run and respond faster). In a space engineering RAG system, this means a designer could use a powerful model like GPT-4o to maximise answer quality when time and expense permit, but opt for an optimised lightweight model for real-time assistance or on-device deployment where resources are limited. In our case, where mission design is critical (as well as safety), a more powerful model is preferable.

Prompt (Query) Engineering for RAG

In RAG, the user's query is first used to fetch relevant documents, and those documents are then concatenated with the query as context for the LLM [23]. This architecture changes how prompt engineering works compared to a standalone LLM. Certain prompting tricks that work in direct LLM usage (e.g. elaborate role-playing instructions like "You are an expert

in X...") may be less effective or even detrimental in RAG, since the system's first step is to use the prompt for retrieval of documents [44]. Prompt engineering in RAG, therefore, focuses on guiding the LLM to utilise the retrieved context faithfully and produce accurate, grounded answers.

System Prompts and Role Instructions

In a chat-based LLM, prompts are structured as a series of messages with roles such as system, user, and assistant. The system message is a crucial component that provides high-level instructions influencing the model's overall behaviour. For example, the system prompt can instruct the model to adopt a specific tone or to only use the retrieved documents when answering questions [44]. In the context of RAG, a system prompt often explicitly tells the LLM to base its answers solely on the provided context and perhaps to cite sources. This helps prevent the model from relying on outdated parametric knowledge or hallucinating by anchoring it to the retrieved evidence. In practice, these system instructions can be quite extensive – for instance, real-world chatbots often include long system prompts detailing the model's knowledge cutoff date, the current date, desired answer format, tone, and steps to follow. Developers have considerable flexibility to shape the LLM's behaviour in this way, and spending time to refine the system prompt can significantly improve the consistency and quality of responses in a RAG system.

Alongside the system message, the prompt template for a RAG application typically inserts other elements in a fixed structure. A common template might begin with the system instructions, followed by a recap of relevant prior conversation (for multi-turn dialogues), then the retrieved documents (often as a formatted list or quotes), and finally the latest user question. Organising the prompt in this structured way ensures the model knows what information is context (e.g. "Here are some documents on the topic") versus what is the user's query. This approach has two benefits. First, it makes it easier to experiment with or adjust different parts of the prompt (e.g. one can easily swap in a new system instruction or change how many documents are provided), and second, it helps the model clearly distinguish user intent from supporting information. Overall, a well-designed prompt template tailored for RAG is the foundation for high-quality, context-aware answers [45].

In-Context Learning (Few-Shot Prompting)

Beyond basic structuring of the prompt, one powerful prompt engineering technique is in-context learning, also known as few-shot prompting. In-context learning refers to providing a few examples of question-answer pairs or other tasks within the prompt itself, so that the model can infer the desired format or style of the answer. By showing the model one example (one-shot) or multiple examples (few-shot), the prompt effectively teaches the model how to respond. For instance, if building a customer service chatbot, the prompt might include a couple of sample customer queries and the high-quality responses that a helpful agent would give. The LLM then uses these as patterns when generating its answer to a new query.

In RAG systems, few-shot prompting can be combined with retrieval to dynamically supply relevant examples. Instead of hard-coding static examples in the prompt, the system could retrieve past Q&A pairs from a knowledge base that are similar to the user's query. Those pairs are then inserted into the prompt as demonstrations. This strategy grounds the model not just in relevant factual documents but also in the style of reasoning or answering that has been effective for similar questions. Essentially, the RAG retriever can fetch illustrative examples in addition to factual context. Research has shown that selecting good in-context examples can notably improve performance [46]. However, one must be mindful of the prompt length – each example consumes tokens from the context window. In practice, few-shot examples are most useful if the task requires a particular format or reasoning style that the LLM wouldn't reliably adopt with an empty prompt. If the model is already well instruction-tuned for general conversations, simple zero-shot instructions (with no examples) might suffice for many RAG applications.

Step-by-Step Reasoning and Chain-of-Thought

Another category of prompt engineering techniques focuses on encouraging the LLM to reason through the problem step-by-step before giving a final answer. This can be done implicitly or explicitly. An explicit method is to prompt the model with a phrase like "Let's think step by step" or to provide an example where the solution is derived through a series of intermediate steps (often called chain-of-thought prompting). Such approaches have been found to significantly improve performance on complex reasoning tasks [47]. For example, Wei et al. (2022) showed that by including a few worked-out reasoning examples in the prompt, even very large models dramatically improved their accuracy on math word problems and logical reasoning benchmarks [47].

There are variations on this theme. One is chain-of-thought with self-consistency, where the model is prompted to generate multiple solution paths and then the most consistent answer is chosen, as a way to reduce reasoning errors. Another is "scratchpad" prompting, where the prompt explicitly delineates a section for the model's reasoning (e.g. a special token or format that the model knows is for thinking) and a section for the final answer. All these techniques aim to leverage the LLM's ability to perform multi-step reasoning when guided properly by the prompt.

Managing Context Window and Multi-Turn Conversations

Prompt engineering must also account for the context window limit of the model. The context window includes the entire prompt (system message, examples, retrieved text, conversation history) and the model's generated reply. RAG by design can quickly stuff a prompt with a lot of text. This can approach the limit, especially in a multi-turn dialogue where previous exchanges are carried along.

To handle this, prompt engineering for RAG employs context management techniques,

sometimes called context pruning. The simplest approach is to include only the most recent conversation turns and the newly retrieved documents when building the prompt for the next user query. Older conversation history can be dropped or summarised. For instance, one might always retain the last N question-answer pairs and omit anything older. If maintaining some long-term context is important, another strategy is to have a running summary of the dialogue. Each time the user asks a new question, the system can inject a summary of the relevant past discussion into the prompt, in place of all the raw prior messages. This keeps the context short while preserving key information. Empirically, this approach works well to remind the model of prior context without overloading it [48].

In summary, prompt engineering in RAG is about finding the right balance between guidance and brevity. A well-crafted system prompt sets the stage for the LLM's behaviour (especially to ensure it uses retrieved knowledge correctly), and additional techniques like in-context examples or chain-of-thought prompting can be employed when they demonstrably improve results. Because RAG already supplies factual grounding via retrieval, the prompt's job is largely to integrate that grounding into a coherent answer. Ultimately, prompt engineering remains something of an art [46].

Handling Hallucinations

Hallucinations are fluent but factually incorrect statements produced by an LLM. They arise because autoregressive models optimise for probable continuations of text rather than truth, so plausible yet false completions can occur, even more so when prompts encourage helpfulness [49]. RAG mitigates (but does not eliminate) this by grounding generation in retrieved evidence [4][50].

A practical strategy in RAG to handle hallucinations is to constrain generation to evidence: Use a system prompt that instructs the model to answer only from the provided context, to cite sources, and to abstain when support is insufficient [4][49][50].

There exist more advanced techniques that are more time-consuming but out of the scope of this project. We expect the user to always double-check information and sources when interacting with the RAG system.

Evaluating LLM Performance

We have already explained how to evaluate a retriever in Subsection 2.4.2. We now aim to understand the performance of our generation (LLM).

To evaluate the LLM's answer itself, classic metrics and task-specific metrics are used. For factoid question answering tasks (where a ground-truth answer is known), Exact Match (EM) accuracy and F1-score (overlap of answer tokens) are common. EM is a strict metric. It's 1 only if the generated answer exactly matches the reference answer [4, p. 4]. F1 allows partial credit by measuring the precision and recall of the answer tokens versus the reference (useful

if there are slight wording differences). For open-ended generation or summarisation tasks, metrics like ROUGE or BLEU compare n-grams of the generated text to reference texts. However, these surface-level metrics often miss the mark for factual correctness and usefulness in RAG. A model could get a high ROUGE score by regurgitating reference phrasing without truly understanding the content, or conversely, it might correctly answer a question with a phrasing different from the reference and be unfairly penalised. Thus, there is increasing emphasis on faithfulness metrics – measuring whether the generated answer is grounded in the retrieved evidence and free of hallucinations. One approach is to have human evaluators judge an answer for factual correctness and support. For example, the RAG-50K benchmark introduced by Gao et al. (2024) [16] has human-rated scores for answer factuality. Human evaluation is the gold standard, but it is costly and not scalable for iterative development.

An emerging alternative is LLM-based evaluation, where an automated "judge" model rates the answer along dimensions like relevance and faithfulness [51]. Frameworks such as Retrieval Augmented Generation Assessment Scores (RAGAS) use prompting techniques to have an LLM read the query, the retrieved context, and the answer, and output scores [51]. LlamaIndex has its own inbuilt set of evaluation metrics that are easy to deploy and test [52].

The goal is to achieve a form of retrieval precision – not returning any unsupported statements – and answer recall – covering all key points of the correct answer. In summary, generation evaluation in RAG goes beyond fluency or grammar (which modern LLMs handle well) to focus on answer correctness and grounding.

2.4.5 RAG Systems in Production

Logging, Monitoring and Observability

Deploying RAG systems at scale requires robust logging and monitoring infrastructure. In production, every query and response should be logged and traceable, enabling engineers to audit system behaviour and diagnose issues. Advanced observability tools go beyond basic health metrics, allowing practitioners to trace data flows, analyse historical outputs, and detect anomalies in real-time [53]. Monitoring for RAG must track not only infrastructure health and latency, but also domain-specific metrics like retrieval success rates, response fidelity, and data drift as the underlying knowledge sources evolve [53]. This comprehensive operational visibility ensures the RAG system remains reliable and accountable over time.

Continuous Evaluation Pipelines

Production RAG systems benefit from custom evaluation pipelines to continuously assess performance in live settings. Standard metrics alone are often insufficient – practitioners develop domain-specific tests and feedback loops to measure both retrieval relevance and generation accuracy on an ongoing basis [53]. For example, organisations assemble specialised test queries with known answers and use synthetic query generation to probe corner cases,

thereby detecting regressions early [53]. Automated evaluation frameworks (as discussed in) can help quantify retrieval correctness and answer faithfulness without relying solely on slow human annotation. By integrating these evaluation results into the deployment workflow, teams can rapidly refine the retriever or generator when quality deviates, ensuring the live RAG system maintains high accuracy and usefulness.

Latency vs. Quality Trade-offs

Online RAG applications must balance response latency against output quality. Users expect quick answers, so production systems often impose tight latency budgets. Achieving low latency may involve using smaller or distilled language models and retrieving only the most relevant few documents, at the expense of some drop in generation richness. Conversely, larger models and more extensive retrieval yield higher answer quality but incur higher latency and cost. Finding the optimal operating point is a key engineering decision – typically choosing the smallest model and minimal context that meets the task's quality requirements. Latency and throughput are tracked as first-class metrics in production, alongside accuracy metrics, to ensure the system remains responsive under real-world load [53]. Engineers may also deploy caching, asynchronous processing, or multi-stage ranking (retrieve then rerank) to improve perceived latency without sacrificing too much quality. Ultimately, production RAG design involves carefully trading off model complexity and retrieval depth to achieve a cost-performance balance aligned with service level objectives.

Security and Compliance Considerations

Production RAG systems in enterprise and regulated domains must adhere to strict security and privacy requirements. Integrating external knowledge bases and user queries introduces new risks. For example, the system could inadvertently leak sensitive proprietary data or be manipulated via malicious inputs [54]. Threats specific to RAG pipelines – such as prompt injection attacks (where retrieved content tricks the model into unintended behaviour) or data poisoning (corrupting the indexed knowledge with false information) – have been documented and need mitigation [54]. Robust RAG deployments, therefore, include safeguards like content filtering, query sanitisation, and verification of retrieved sources. Fine-grained access controls and audit trails are implemented to ensure compliance with data protection regulations.

Multimodal RAG

An emerging frontier for RAG systems is multimodal retrieval and generation, where the system can handle not just text but also images, charts, or other data types as both input and retrieved context. Real-world enterprise data (e.g. in aerospace or healthcare) is often inherently multimodal, combining text with sensor data, diagrams, or tables [53]. Early multimodal RAG solutions typically convert all inputs into text (for example, by captioning images or serialising tables) so that a language model can process them, although this conversion can discard some information [53]. Research prototypes show that integrating

heterogeneous data sources can improve performance on tasks like anomaly detection, where combining visual and textual evidence yields richer insights than either alone [53]. While still in its infancy, multimodal RAG is expected to grow in importance, enabling systems that, for instance, retrieve relevant diagrams or satellite images alongside documents and have the generative model explain them together. This development will further broaden the applicability of RAG systems across domains that demand a holistic understanding of diverse data formats.

2.4.6 RAG vs. Fine-Tuning vs. Prompt Engineering

An important question is how RAG compares to other methods of specialising LLMs, namely fine-tuning and prompt engineering. Fine-tuning entails updating an LLMs's weights on domain-specific data, while prompt engineering means crafting optimal prompts (possibly with few-shot examples) to achieve better performance from a fixed model. Each approach has strengths and trade-offs [23][55].

Fine-tuning excels at teaching an LLMs domain-specific terminology and style, and generally yields higher base accuracy on tasks within that domain [22]. However, it requires a large curated dataset, substantial compute, and must be redone to incorporate any new knowledge, which is impractical for rapidly evolving information [22]. Prompt engineering is lightweight but has limited ability to correct factual mistakes or add new knowledge beyond what the model already knows. RAGs, by contrast, is ideal when "up-to-date external knowledge" is needed and one wants to minimise changes to the model itself [55]. By simply updating the document index, a RAG system can immediately take into account new data (e.g. the latest mission results or standards) without retraining the RAG [55]. This makes RAG very agile for quickly changing knowledge challenges, such as keeping up with incremental documentation updates and newly published lessons learned. Moreover, RAG provides traceability – the assistant can cite its sources, which is vital for high-stakes engineering domains. Fine-tuned models, in contrast, act as black boxes with no built-in mechanism to indicate where a fact originated.

Fig. 2.4 visually compares RAG with other adaptation methods. We see that RAG occupies the regime of high external knowledge, low model adaptation, whereas pure fine-tuning occupies the opposite regime. In practice, these techniques can be combined (a fine-tuned LLM can be used within a RAG loop), but the consensus is that RAG offers a faster, more lightweight route to inject new knowledge and achieve factual accuracy in LLM responses [23][55]. This aligns with the needs of early mission design, where information is continuously evolving and decisions must be justified with evidence.

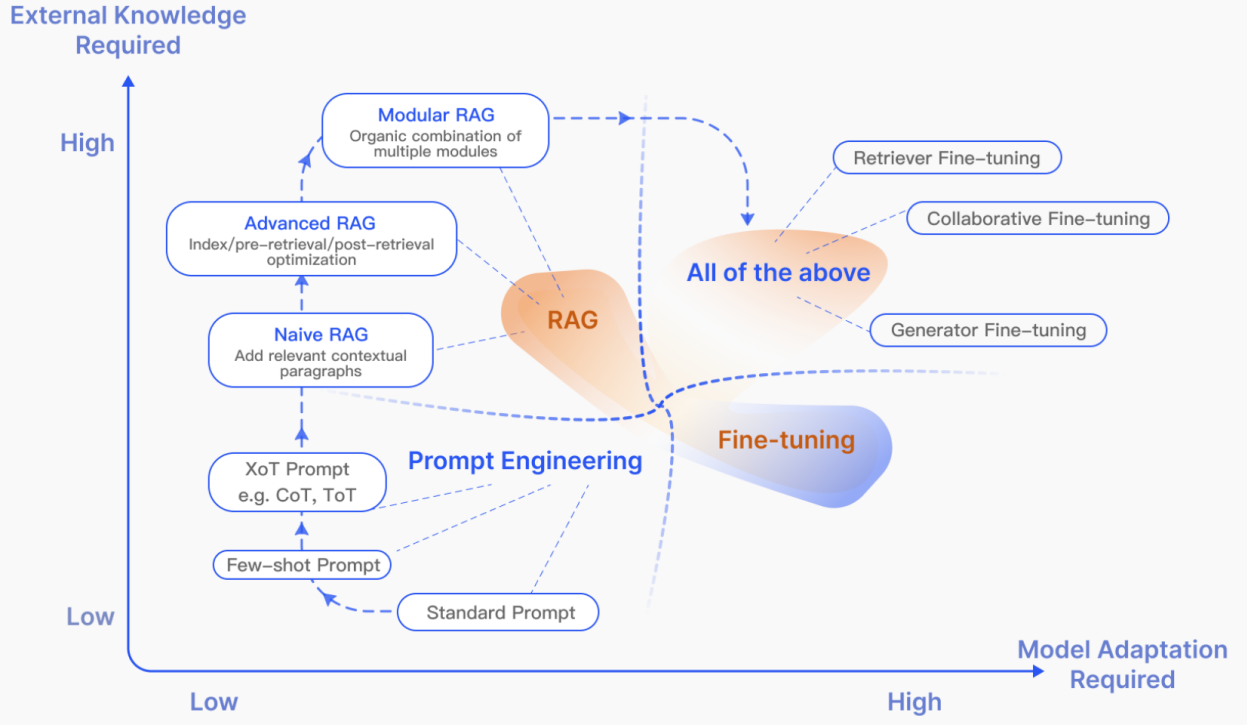

Figure 2.4: Comparison of RAG with other model optimisation methods (prompt engineering, fine-tuning) along two axes. "External Knowledge Required" and "Model Adaptation Required." RAG is most advantageous in scenarios demanding high knowledge freshness and minimal changes to the LLM, whereas fine-tuning suits scenarios with static knowledge but requiring specialised adaptation [23]. (Adapted from Gao et al. 2024 [16, p. 7])

Finally, it should be noted that implementing a RAG system involves engineering considerations beyond the core algorithm. The retrieval component can be enhanced with caching, re-ranking of results, and filtering to avoid irrelevant data. The LLM prompting strategy must handle the context window limitations (e.g. truncating or compressing retrieved text if too long) [23]. Despite these complexities, RAG's modular nature – decoupling knowledge storage from the model – makes it appealing for systems engineering applications. Finally, as Section 2.2 will show, this approach is also a promising complement to classical Model-Based Systems Engineering methods for early mission design.

2.5 Conclusion: Synthesising RAG and MBSE for Future Systems Engineering

This chapter has charted a course from the high-level challenges of modern space mission design to a specific, actionable technological solution. We began by establishing MBSE as the current industry standard for managing system complexity, while also identifying a critical gap - the disconnect between structured models and the vast repository of unstructured historical knowledge.

We then reviewed how LLMs are poised to bridge this gap, concluding that a Documentation Q&A assistant powered by RAG offers the most immediate and impactful application. This approach directly addresses the knowledge retrieval needs of systems engineers, allowing them to make more informed decisions early in the design phase. The detailed exploration of RAG architecture and theory underscores its suitability for this task, offering factual, traceable, and up-to-date information retrieval.

Ultimately, this literature review establishes the foundation for this research project. By developing a RAG-based design assistant, we are creating a foundational tool for knowledge management that complements existing MBSE practices. This assistant serves as a vital first step, creating a pathway for future work where such AI tools could be more deeply integrated into the systems engineering lifecycle, potentially assisting with the direct generation and validation of SysML models.

Chapter 3

Methodology

3.1 Overall Approach & System Architecture

The methodology for this project is designed to systematically address the aim of developing and evaluating a RAG design assistant for early-phase space mission analysis. The approach is structured into four primary phases, which are detailed in the subsequent sections of this chapter:

1. **Knowledge Base Construction:** Sourcing and preprocessing a domain-specific corpus of space mission documentation.
2. **RAG Pipeline Implementation:** Building the core retrieval and generation system using the LlamaIndex framework [56].
3. **Prototype Deployment:** Creating a functional user interface for interaction and testing.
4. **Evaluation Framework:** Defining and executing a robust set of quantitative and qualitative metrics to assess the system's performance.

The end-to-end workflow, from data ingestion to user interaction, is encapsulated in the system architecture shown in Fig. 3.1. This architecture distinguishes between the offline processes required for knowledge base preparation and the online methods that occur in real-time during a user query (whilst following the typical RAG architecture [57]).

The offline stage begins with the acquisition of mission data from ESA's eoPortal [1]. This raw data is preprocessed into a clean, structured format suitable for ingestion and analysis. The LlamaIndex framework then orchestrates the core ingestion pipeline: documents are loaded, parsed into smaller text chunks (nodes), and converted into high-dimensional vector embeddings using an OpenAI embedding model [58]. These embeddings are then indexed and stored in a persistent ChromaDB vector database [59], which serves as the system's

long-term memory or ‘knowledge base’.

The online stage is initiated when a user submits a query through the prototype interface. The same embedding model converts the user’s query into a vector. This query vector is used to perform a similarity search against the indexed vectors in ChromaDB, retrieving the top-k most semantically relevant text chunks. These retrieved chunks, which serve as factual context, are then combined with the original user query into an augmented prompt. Finally, this prompt is passed to the OpenAI GPT-4o model, which synthesises a coherent, context-grounded answer for the user. The following sections will provide a detailed account of the implementation of each component within this architecture.

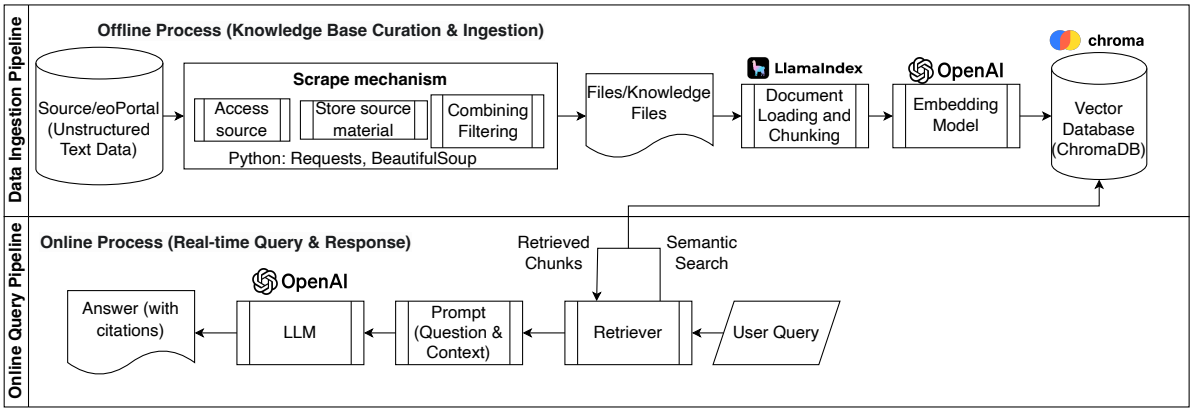

Figure 3.1: Illustrative high-level RAG architecture.

3.2 Knowledge Base Construction

The foundation of any effective RAG system is a high-quality, domain-specific knowledge base. This section outlines the acquisition and preprocessing steps taken to build the corpus for the space mission design assistant, aligning with the project’s first objective.

3.2.1 Data Acquisition

The primary (and exclusive) data source for the knowledge base is the European Space Agency (ESA) eoPortal mission directory [1]. This source was chosen for its trusted provenance (operated by ESA), structured and detailed content, and high-quality data relevant to our domain. The repository contains detailed articles on hundreds of Earth Observation (EO) and other satellite missions, providing rich, unstructured text covering mission objectives, instrument payloads, orbital parameters, and operational status, making it ideal for domain-specific Q&A. This is a single, curated source that reduces contradictions and makes citation auditing straightforward. Depth and cleanliness matter more than breadth at this stage, and simplify error analysis. Another important reason why we have chosen one data source is repeatability. A single source with a simple structure makes scraping, parsing, and later re-ingestion repeatable.

To acquire this data, a custom web scraping solution was developed through an iterative process. An initial, methodical scraper (`Slow_Scraper.py`) was built to ensure data quality and handle the complexities of the dynamic web platform. Subsequently, this scraper was re-engineered for high-throughput performance to enable the rapid collection of the entire corpus (`Fast_Scraper.py`) - see Appendix C.1 for a detailed breakdown of both methods.

Initial Methodical Approach: The Slow Scraper

The first iteration, `Slow_Scraper.py`, was designed with a primary focus on robustness and data integrity. It incorporated several key features:

- **Sitemap Discovery:** To ensure comprehensive coverage, the scraper first parsed the site's `sitemap.xml`, providing a direct and efficient method for discovering all mission page URLs. This is also better than employing a brute-force crawl, which is inefficient and can place unnecessary load on the host server. The scraper utilises a more targeted discovery strategy.
- **Dual-Mode Fetching:** Acknowledging that the target website uses JavaScript for content rendering and cookie consent, a dual-mode strategy was implemented. While simple pages could be fetched with the fast `requests` library, the primary mode utilised `Selenium` to control a headless Chrome browser. This allowed for full page rendering and programmatic interaction with the cookie consent banner, which was critical for accessing the underlying mission data.
- **Sequential Processing Bottleneck:** A key characteristic of this initial approach was its processing model. Selenium WebDriver instances are not inherently thread-safe, meaning a single browser instance cannot be reliably controlled by multiple parallel threads. To guarantee stability and prevent race conditions, the Selenium-based scraping was executed sequentially. While this method produced high-quality data, it introduced a significant performance bottleneck, making the process of downloading over 1,000 documents prohibitively time-consuming.

Performance Optimisation: The Fast Scraper

The primary objective of the second iteration, `Fast_Scraper.py`, was to overcome the sequential processing bottleneck of the initial design while retaining its robustness. The key architectural improvement was the implementation of a thread-local driver architecture.

- **Parallel JavaScript Rendering:** Instead of using a single, shared Selenium instance, the fast scraper was engineered to instantiate a new, independent headless browser for each worker thread in its parallel processing pool (`ThreadPoolExecutor`). This allowed for true concurrent JavaScript rendering, as each thread operated on its own sandboxed browser instance without interference. A `finally` block was used to ensure each browser was properly closed after its task, preventing memory leaks.

- **Enhanced Robustness:** Further improvements were added, including more sophisticated handling of HTTP redirects (which could otherwise lead to duplicate content under different filenames) and a manifest-checking system that allows the scraper to be stopped and resumed without re-downloading existing files.

This iterative engineering process resulted in a final data acquisition tool that was not only capable of handling the complexities of the target website but could also do so with an order-of-magnitude increase in speed, as will be shown in the Results chapter (see Section 4.2).

3.2.2 Data Preprocessing and Structuring for RAG

The raw data acquired from the scraping process, while comprehensive, is not in an optimal format for direct ingestion into a RAG pipeline. A crucial intermediate step of preprocessing and structuring was performed to enrich the corpus, enhance data quality, and format the content for effective retrieval. This process was done via the Python script, `prepare_rag_data.py`.

The primary goals of this preprocessing phase were to first consolidate mission-related information, second, to convert structured data into an LLM friendly format, and finally, to embed essential metadata directly into the documents. The script executed the following key operations (see Appendix C.2 for a detailed breakdown):

1. **Metadata Handling:** The script first parsed the scraper's manifest file to create a master metadata file (`mission_metadata.csv`). This file aggregates key attributes for each mission, such as its title, source URL, text length, and the number of extracted tables and images. This centralised metadata provides a high-level overview of the corpus and is useful for contextualising the documents (see also Metadata Filtering in Section 2.4.2 for more information on why this step is needed/useful).

2. **Data Quality Filtering:** A critical quality control measure was implemented to ensure the integrity of the knowledge base. Any mission document containing fewer than a predefined minimum number of characters (`MIN_TEXT_CHARS = 800`) was filtered out. This step removes error pages that contain no substantive information, preventing the RAG system from ingesting low-value content that could otherwise lead to poor-quality retrievals.

3. **Structured Data Conversion:** A key challenge in RAG is making structured data, such as tables, accessible to a language model. To address this, a novel approach was taken: all tables extracted from the mission pages were programmatically converted into Markdown format. This text-based representation of tables is highly readable by both humans and LLMs. By appending these Markdown tables to the main body of text, the structured data is seamlessly integrated into the unstructured corpus, making it directly indexable and searchable by the embedding model.

4. **Document Enrichment:** To ensure that every retrieved text chunk retains its core context, a standardised header was prepended to each mission document. This header contains the mission's title, its unique ID, and the source URL. This enrichment guarantees that even if a small, isolated chunk is retrieved during a query, the LLM will have the necessary metadata to identify its origin and context, which is important for generating accurate and traceable answers.

The output of this script is a "RAG-ready" directory containing a set of unified documents. Each document represents a single mission and contains an enriched, self-contained body of text that is optimised for the subsequent chunking, embedding, and indexing stages performed by LlamaIndex.

3.3 RAG Pipeline Implementation using LlamaIndex

With a clean and structured corpus of mission documents, the core RAG pipeline was implemented using the LlamaIndex framework [56]. The implementation was logically separated into two distinct scripts: an offline `indexing_pipeline.py` for building the knowledge base, and an online `query_pipeline.py` for handling user interaction.

3.3.1 Data Ingestion and Indexing

The ingestion process, defined in `indexing_pipeline.py`, transforms the preprocessed documents into a queryable vector index stored in ChromaDB [59]. This script was designed to create a unified and context-rich representation of the source data (see Appendix C.3 for a detailed breakdown).

1. **Document Loading and Enrichment:** The pipeline begins by loading each mission's processed JSON file into a LlamaIndex `Document` object. During this step, the rich metadata generated in the preprocessing phase (mission title, URL, whether the document contains tables or images, etc.) is attached to each document.

2. **Unified Content Representation and Chunking:** Instead of treating text and tables separately, the pipeline adopts a unified representation strategy.

    - **Content Inlining:** Before parsing, structured elements like tables and image descriptions are serialized and integrated directly into the document's main text content. Tables are converted into Markdown format, and image captions or alt-texts are appended. This ensures that the contextual relationship between the text and its associated tables or images is preserved within a single data stream. A cleaning function also removes navigational artifacts and redundant content to improve signal quality.

    - **Text Chunking:** This unified text content is then parsed using a `SentenceSplitter`.

A large `chunk_size` of 2000 tokens with a `chunk_overlap` of 100 tokens was chosen. This configuration is designed to keep large, self-contained sections of documents - including the newly inlined tables - intact within a single chunk. This reduces the risk of fracturing critical information and helps the retrieval system access complete context for synthesis.

3. **Embedding and Storage:** All generated nodes (text chunks) were converted into 3072-dimension vector embeddings using OpenAI's powerful `text-embedding-3-large` model. The resulting vectors were then indexed and stored in a persistent ChromaDB vector store, which uses Chroma's ANN algorithm to enable fast similarity search (see Subsection 2.4.3). This persistence is critical, as it allows the query engine to load the fully built index instantly without needing to re-run the expensive embedding process[1].

3.3.2 Query Engine Construction

The `query_pipeline.py` script constitutes part of the online component of the assistant (this script is called by the chatbot). It loads the persistent index from ChromaDB and constructs a sophisticated query engine designed for high-quality, traceable responses (see Appendix C.4 for a detailed breakdown).

1. **Multi-Step Retrieval Process:** The engine employs a multi-step retrieval process to ensure the context provided to the LLM is of the highest possible relevance and quality.
    - **Initial Retrieval:** A `VectorIndexRetriever` first fetches the top-k (initially k=5 but optimised in the results) most semantically similar nodes from the vector index based on the user's query.
    - **Quality Filtering:** The retrieved nodes are then passed through a `SimilarityPostprocessor`. This component acts as a quality gate, immediately filtering out any node that does not meet a predefined similarity threshold (initially 0.35 but optimised in the results). This step is crucial for reducing noise and preventing tangentially related information from impacting the final answer.
2. **Response Synthesis:** The filtered, high-relevance nodes are then passed to a `Response Synthesizer`. This component leverages the efficient `gpt-4o-mini` model[2] to generate a coherent, synthesised answer based on the provided context and the original user query. The temperature is set to 0.0 (initially but optimised later) to ensure deterministic, factual responses.

[1] If we wanted to add more mission data, we could embed it independently and add it to the ChromaDB file, allowing us to keep up to date with the space industry and keeping us relevant.

[2] The `gpt-4o-mini` model was used for the automated parameter sweep due to its speed and lower cost. The final deployed prototype utilizes the more powerful `GPT-o3` thinking model to ensure the highest quality responses for the user.

3. **Metadata-Aware Querying:** The architecture supports metadata-aware queries by applying `MetadataFilters` during the retrieval step. While nodes are not explicitly tagged by content type (e.g., 'table'), this feature allows the search to be constrained at the document level. For instance, a query can be narrowed to search only within documents that are flagged with `'has_tables': True`, focusing the retrieval on missions where structured data is known to exist.

Finally, to ensure the system is trustworthy and aligns with the needs of an engineering context, the query engine is designed to return the source nodes alongside the generated answer. This provides full traceability, enabling users to verify the source of any claim made by the assistant.

Overall, using only the eoPortal corpus ensures that all retrieved facts come from a vetted source (ESA's database) with consistent structure. This significantly reduces the risk of inaccuracies. Using OpenAI's embeddings and LLMs leverages their superior performance – our focus is on reliability and accuracy over model experimentation, so we use proven tools like `gpt-4o-mini` rather than optimising alternative embeddings or models (such as open-source ones). (Future work could experiment with domain-specific embeddings or open-source LLMs, but that is beyond our current scope.) The combination of LlamaIndex and ChromaDB lets us implement the RAG pipeline with minimal overhead, given LlamaIndex's convenient integration with vector stores and document chunking.

3.4 Prototype Deployment

To bridge the gap between the backend RAG pipeline and a functional user application, a prototype was developed using the Streamlit framework [60]. The primary objective of this prototype is to provide an intuitive, interactive interface for qualitative evaluation, allowing for a practical assessment of the RAG assistant's utility in a simulated user environment. The complete implementation is detailed in the `streamlit_chatbot.py` script (see Appendix C.5 for a detailed breakdown).

The choice of Streamlit was a deliberate methodological decision. As a Python-native library, it allows for rapid development of data-centric web applications without requiring expertise in traditional web development (HTML, CSS, JavaScript). This enabled the project to focus resources on the core RAG logic while still producing a professional and highly functional user interface for testing and demonstration.

Usage Run `streamlit run streamlit_chatbot.py`, enter an API key, then query (e.g., "Typical power for EO CubeSats?"). The app displays the answer, response time, and curated sources; logs can be saved.

3.4.1 User Interface and Architectural Design

The prototype's architecture is designed around user experience and the specific needs of a question-answering system. It leverages Streamlit's core components to create a robust and stateful chat application.

- **State and Session Management:** A critical feature of any interactive application is the ability to maintain state. The chatbot heavily utilises Streamlit's `session_state` object to persist the conversation history, session ID, and user settings across interactions. As Streamlit re-runs the script on every user input, this state management is fundamental to preserving the conversational context and providing a coherent user experience.

- **Two-Panel Layout:** The interface is organised into a standard and intuitive two-panel layout. A main panel is dedicated to the chat conversation, providing a clean and focused area for user interaction. A persistent sidebar contains all meta-functionality, including session information, settings (such as toggling source visibility), and example questions to guide the user. This separation of concerns is a standard UX pattern that prevents the main interface from becoming cluttered.

- **Secure API Key Handling:** To make the application portable and secure, it does not store any hardcoded API keys. Upon first launch, the user is prompted to enter their own OpenAI API key, which is then stored exclusively in the session state for the duration of their session. This approach ensures that sensitive credentials are never persisted to disk and that the user maintains full control over their API usage.

3.4.2 Features for an Engineering-Focused Assistant

Beyond a standard chat interface, several features were specifically implemented to align with the requirements of a technical, engineering-focused tool where traceability and verifiability are paramount.

1. **Traceability and Source Management:** As emphasised in the literature review, the ability to verify information is crucial in high-stakes domains like space mission design. To address this, the chatbot provides robust source management:

    - After generating a response, the assistant displays a list of the source documents that were used as context.

    - Each source is presented as a clickable hyperlink, taking the user directly to the original eoPortal page. This allows for immediate fact-checking and deeper exploration.

    - The relevance score of each source node is displayed, giving the user a quantitative measure of how closely that source matched their query.

- A deduplication algorithm is applied to the source list, ensuring that if multiple chunks are retrieved from the same mission document, only a single, clean link to that mission is presented. This enhances usability by preventing a cluttered and redundant list of sources.

2. **Session Logging for Analysis:** The prototype includes a feature to save the entire conversation history to a structured JSON file. Each log contains the full sequence of user prompts and assistant responses, along with metadata such as timestamps and the source nodes used for each answer. This capability is not just a convenience feature; it is a critical tool for the research methodology, enabling detailed post-session analysis of the assistant's performance, identification of common failure modes, and systematic evaluation of the user experience.

3. **Engine Transparency and Usability:** To aid both users and developers, the sidebar includes a set of usability features. Example questions are provided to onboard new users and demonstrate the intended scope of the assistant. For advanced analysis, an option to display the live query engine's statistics (such as the number of indexed chunks and the models in use) offers transparency into the system's configuration.

In summary, the Streamlit prototype is more than a simple interface; it is an integral part of the evaluation framework. Its design prioritises usability, and its feature set, with a particular emphasis on traceability and logging, is tailored to the needs of a reliable and verifiable engineering design assistant.

3.5 Evaluation Framework

3.5.1 Evaluation Introduction

Before diving into this section, it's important to take a step back and look at what we're trying to achieve.

- **Problem:** We need to evaluate our RAG system, but no standard test exists for asking questions about space missions. We also need to find the best settings (hyperparameters) for our system, but testing every combination is impractical.

- **Our Solution:** Three steps

    1. **Create a High-Quality Test** `generate_100_questions.py`: This solves the no test problem by using a fast and competent LLM (gpt-4o-mini) to act as a domain expert. We provide it with our actual mission documents (randomly) and ask it to generate 100 challenging, diverse questions and their ground-truth answers. This creates a synthetic but high-quality, domain-specific benchmark dataset.

    2. **Run a Systematic Experiment** `comprehensive_evaluation.py`: This solves

the best settings problem by performing a parameter sweep. The script systematically iterates through every possible combination of your chosen hyperparameters (e.g., different top_k values, similarity thresholds, temperatures, etc.). For each combination, it runs your RAG system against a random sample of 20 of these questions[3] and calculates a rich set of performance metrics.

3. **Analyse and Report the Results** `visualization_and_reporting.py`: This solves the "what does it all mean?" problem by generating a suite of plots and tables. These visualizations are designed to answer key questions like "How does changing top_k affect retrieval?" and "What is the trade-off between answer quality and response time?"

- This entire process, orchestrated by `run_evaluation_pipeline.py`, represents a robust, automated, and repeatable scientific experiment to validate and optimise the RAG system (the questions used for the evaluation tasks in Chapter 4 are provided on GitHub for reproducibility).

3.5.2 Step 1: Synthetic Dataset Generation

The foundation of any robust evaluation is a high-quality dataset of questions and ground-truth answers. As no such public benchmark exists for space mission design queries, a synthetic dataset was generated using the `generate_100_questions.py` script.

The methodological choice to use an LLM (`gpt-4o-mini`) for data generation was made to ensure three key characteristics:

1. **Domain Specificity:** By prompting the LLM with the actual text from a diverse, random sample of 20 mission documents from the knowledge base, the generated questions are guaranteed to be relevant and answerable within the system's corpus.

2. **Question Diversity:** To avoid testing only one type of query, a predefined distribution of question types was enforced. The generation process was guided to create a balanced set of 100 questions covering factual recall, technical specifications, comparison, analytical reasoning, and more. This ensures the RAG assistant is evaluated across a wide range of realistic user intents.

3. **Inclusion of Ground Truth:** For each generated question, the LLM was instructed to provide not only the correct answer but also the specific text excerpt from the source document that justifies it. This provides the essential ground truth for both the **retrieved context** and the final **generated answer**, enabling the independent evaluation of both retrieval and generation stages, a critical practice discussed in the literature review.

[3]Querying 100 questions with multiple combinations would take days, so 20 questions are chosen. 100 questions are generated in case a more detailed analysis is wanted to be done in the future.

The output of this step is a high-quality, structured JSON file containing 100 question-answer-context triplets, forming a repeatable and challenging benchmark for the subsequent evaluation stages.

3.5.3 Step 2: Comprehensive Evaluation and Parameter Sweep

With a benchmark dataset established, the `comprehensive_evaluation.py` script was executed to systematically measure the RAG assistant's performance across a wide range of hyperparameters. This parameter sweep is a crucial methodological step to move beyond a single performance score and instead understand the system's behaviour and trade-offs.

The script iterates through every combination of the parameters defined in a configuration file, including:

- **Retrieval Parameters:** 'top_k' (the number of documents to retrieve) and 'similarity_threshold' (the relevance cutoff).
- **Generation Parameters:** 'temperature' (the creativity of the LLM) and 'response_mode' (the LlamaIndex strategy for synthesising the final answer).

For each unique parameter combination, the script runs the query engine against a sample of the 100-question dataset and calculates a comprehensive suite of metrics. This approach was justified by the need to answer key engineering questions about the system's design: Is a higher 'top_k' always better? What is the optimal 'similarity_threshold' to balance recall and precision? How does 'temperature' affect factual accuracy?

3.5.4 Step 3: Evaluation Metrics and Reporting

The final stage of the pipeline, executed by `visualization_and_reporting.py`, is the calculation and visualization of a rich set of metrics designed to provide a multi-faceted view of system performance. The evaluation is logically divided into two main categories, reflecting the two core components of a RAG system.

Retrieval Performance Metrics

These metrics assess the retriever's ability to find the correct source documents, using the ground-truth contexts from the synthetic dataset. The key metrics include:

- **Recall@k and Precision@k:** These fundamental metrics, discussed in the literature review, measure the completeness and correctness of the retrieval set.
- **Mean Average Precision (MAP@k) and Mean Reciprocal Rank (MRR):** These more sophisticated metrics evaluate the ranking of the retrieved documents, rewarding the system for placing the most relevant documents at the top of the list.

Generation Performance Metrics

These metrics assess the quality of the final, user-facing answer. A hybrid approach combining traditional Natural Language Processing (NLP) metrics with modern, LLM-based evaluation was used for a comprehensive assessment:

- **Ground-Truth Based Metrics:** These include token-level **F1-Score** and **Exact Match**, which measure the lexical overlap with the ground-truth answer, and **ROUGE/BLEU** scores, which assess fluency.
- **LLM-as-a-Judge (LlamaIndex Evaluators & Retrieval Augmented Generation Assessment Scores (RAGAS) through LlamaIndex [61]):** To capture semantic meaning beyond word overlap, a powerful judge LLM (`gpt-4o`) was used to score the generated answers on qualitative axes. The key metrics, provided by LlamaIndex's (and RAGAS's) evaluation suite, are:
  - **Faithfulness:** Measures if the answer is factually consistent with the retrieved context (i.e., does not hallucinate).
  - **Answer Relevancy:** Measures if the answer directly addresses the user's question.
  - **Correctness:** Compares the generated answer against the ground-truth answer for semantic equivalence.

By systematically executing this three-stage pipeline, this evaluation framework produces a rich, multi-dimensional dataset of performance results. The final visualizations, detailed in the Results Chapter (Section 4.6), are then used to identify the optimal parameter configuration and to draw robust conclusions about the RAG assistant's capabilities and limitations.

Chapter 4

Results

4.1 Introduction

This chapter presents the comprehensive results of the research project, directly addressing the aims and objectives outlined in Chapter 1. The findings are structured to follow a logical progression, beginning with an evaluation of the data acquisition process, followed by a detailed analysis of the constructed knowledge base, and culminating in an assessment of the RAG assistant's performance.

The first section quantifies the effectiveness of the custom web scraper, validating the methodology chosen for data collection. The second section characterises the resulting knowledge base, analysing its scope, content, and quality to establish the foundation upon which the RAG system operates. The third and most critical section presents the quantitative and qualitative performance of the RAG assistant itself, using the evaluation framework defined in the methodology. Finally, a discussion synthesises these findings, interpreting their significance in the context of the project's goals and the state-of-the-art reviewed in Chapter 3.

4.2 Data Acquisition Performance and Optimisation

As established in the literature review, the performance of a RAG system is fundamentally dependent on the quality and comprehensiveness of its underlying knowledge base. This section validates the iterative data acquisition methodology, demonstrating the successful development of a high-throughput scraper by comparing its performance to an initial, more methodical version.

All plots are produced from the `KnowledgeBaseAnalysisComplete.py` script.

The summary of the scraping operation is presented in Table 4.1. A total of 1,096 mission

pages were identified and targeted for download. Of these, 1,095 were successfully downloaded on the first run and processed, resulting in an exceptional success rate of 99.9%. The remaining was successful on the script's second attempt (still within the same runtime). This result is significant as it validates the critical design choice of our fetching strategy. The Selenium-based JavaScript rendering was successful in handling cookie consent banners, which would have otherwise blocked the scraper and resulted in a failed download (only the cookie consent form would have been downloaded). The low failure rate demonstrates the robustness of this approach. The table also quantifies the scale of the resulting knowledge base, which suggests a rich, multi-modal dataset.

Metric	**Value**
Total Missions Scraped	1,096
Successful Downloads (First Shot)	1,095 (99.9%)
Total Tables Extracted	3,980
Total Images Catalogued	27,682
Total Text Characters	51,435,174
Total Data Size (GB)	0.27
Avg. Tables per Mission	3.6
Avg. Text Length (chars)	46,930

Table 4.1: Summary statistics of the completed data acquisition process, quantifying the total volume and high success rate of the data collection - using the fast scraper method.

The primary outcome of the scraper optimisation is illustrated in Fig. 4.1 (slow method) and Fig. 4.2 (fast method).

The initial, slow scraper, limited by its sequential processing of JavaScript-rendered pages, required approximately 1 hour and 45 minutes (105 minutes) to download the entire corpus, maintaining an average rate of approximately 10 missions per minute. In contrast, the optimised fast scraper, leveraging a parallel, thread-local driver architecture, completed the same task in approximately 25 minutes. Its scraping rate was significantly higher and more variable, averaging between 20-30 missions per minute, reflecting the efficiency of concurrent processing. This represents a greater than **4x increase in performance**, validating the engineering effort to overcome the initial bottleneck and enabling the rapid collection of the knowledge base.

The drop in Fig. 4.2 around 01:04 is due to the failed first-shot attempt at the 1 mission (see Table 4.1); the script retried the URL and succeeded. Comparing the other plots in the folders `Slow_KnowledgeBasePlots` and `Fast_KnowledgeBasePlots` confirms that the data

extracted is identical.

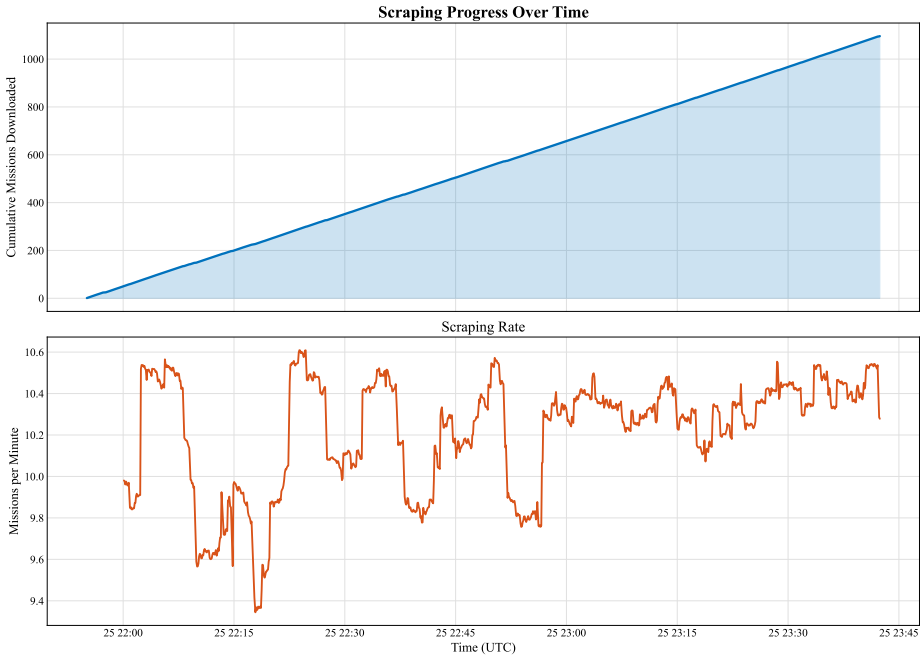

Figure 4.1: Slow scraping progress and rate throughout the data acquisition process. The top plot shows the steady accumulation of mission documents, while the bottom plot illustrates the stable download rate achieved through parallelisation. Time is expressed in UTC, with the 26 being the 26th of the month (August in this case), and 2200 being UTC.

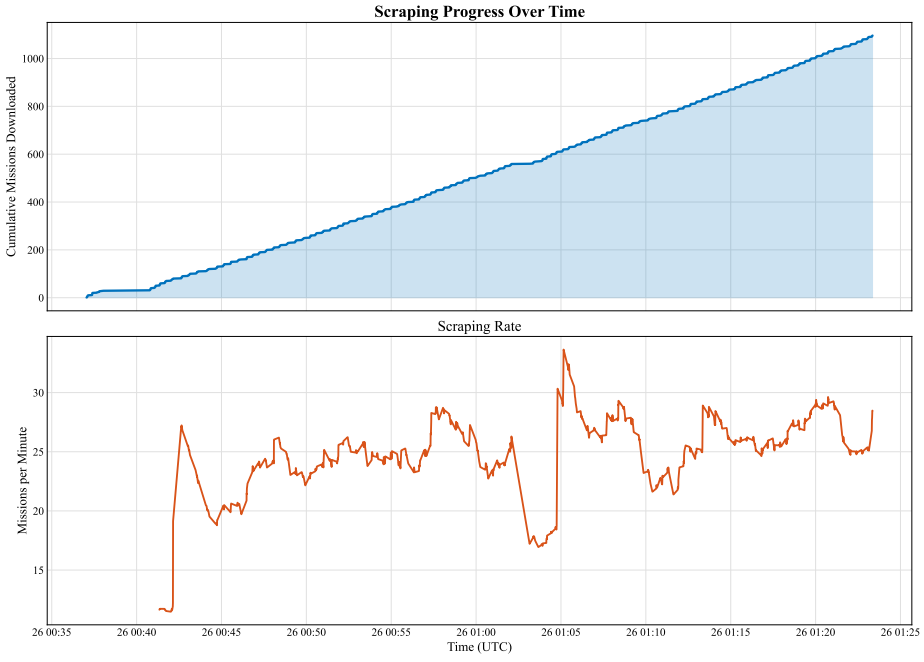

Figure 4.2: Fast scraping progress and rate throughout the data acquisition process.

4.3 Knowledge Base Analysis

An understanding of the corpus's contents, biases, and information density is important for interpreting the RAG assistant's subsequent performance.

This section provides a detailed analysis of the newly constructed knowledge base to evaluate its viability as a foundation for the RAG assistant. The following plots examine key characteristics of the corpus to confirm its suitability and relevance for the project's objectives. The analysis will assess the **temporal applicability** by examining mission launch decades, the **thematic scope** by identifying common terms and prevalent space agencies, and the overall **information richness** by quantifying the volume and diversity of the extracted content. Collectively, this evaluation serves to validate the dataset as a comprehensive and appropriate resource for a modern space mission design assistant.

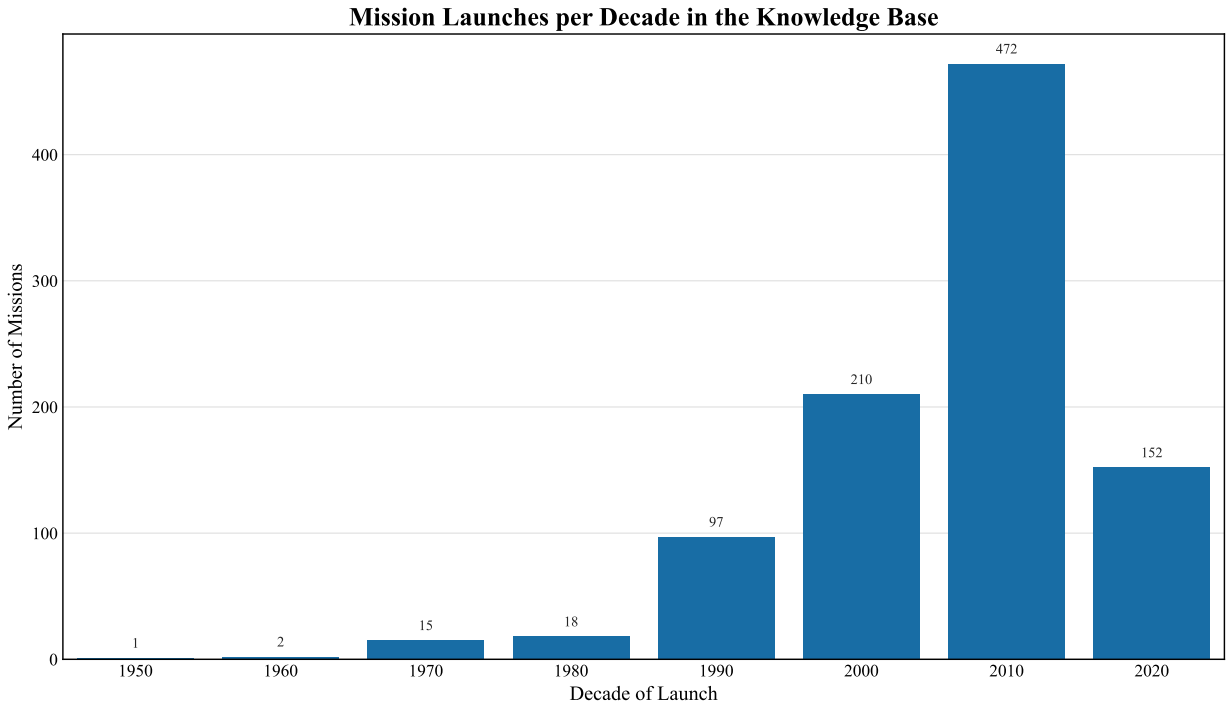

Figure 4.3: Distribution of missions in the knowledge base by their decade of launch. The plot shows a significant concentration of missions from the 1990s onwards.

The temporal distribution of the missions is shown in Fig. 4.3. The data is heavily weighted towards missions launched in the 2000s and, most significantly, the 2010s, which saw 472 missions launched. This aligns with the expansion of institutional and commercial space activities, particularly the rise of smaller satellites and constellations. This indicates that the knowledge base is rich with modern mission architectures and technologies, making it highly relevant for contemporary design tasks.

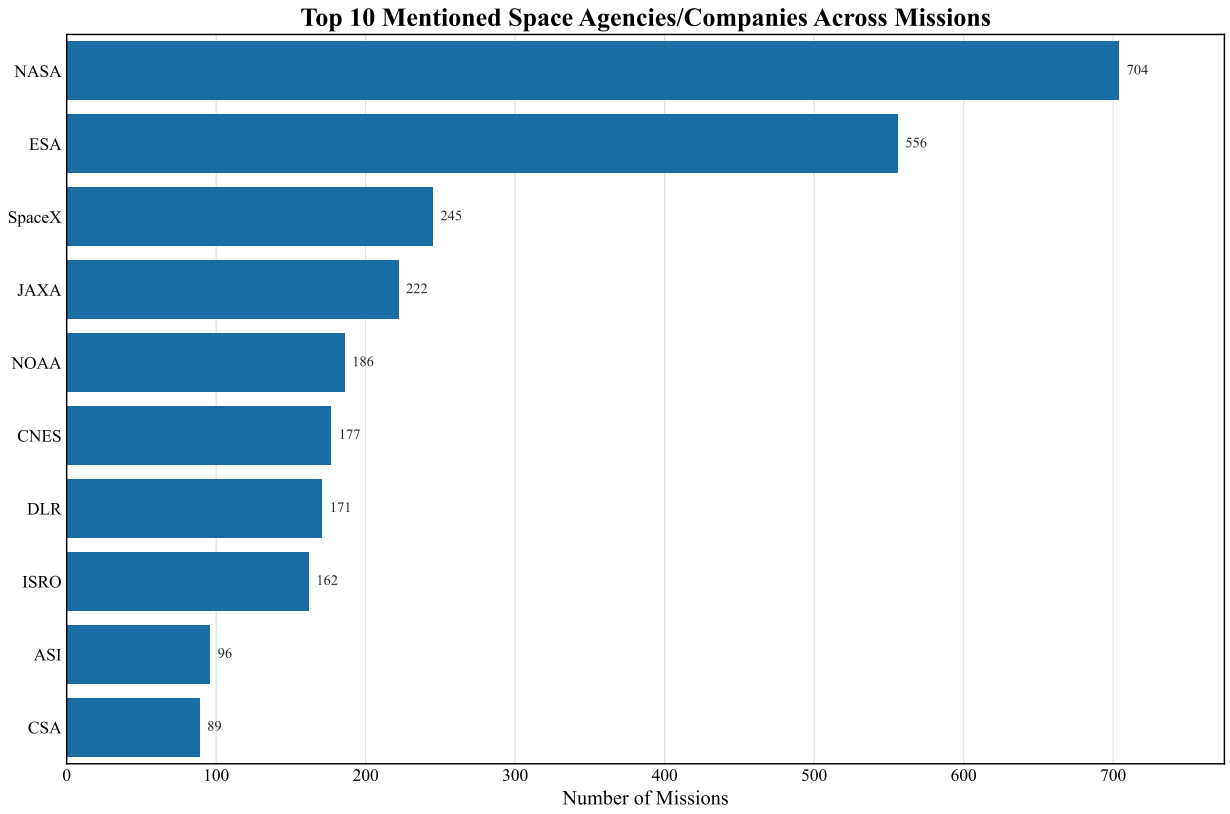

Figure 4.4: Frequency of the top 10 most mentioned space agencies or companies across all mission documents. The data highlights a strong prevalence of NASA and ESA missions.

Fig. 4.4 illustrates the distribution of missions by associated space agencies or companies. The corpus is predominantly composed of missions involving NASA (704 mentions) and ESA (556 mentions), which is expected given the eoPortal's focus. This inherent bias means the RAG assistant will perform as an expert on missions from these agencies but will have limited knowledge of others. The notable presence of SpaceX (245 mentions) confirms that the dataset includes substantial information on modern commercial launch and mission activities, further enhancing its relevance. We do have to take into consideration that SpaceX is a private company and may withhold a lot of sensitive information.

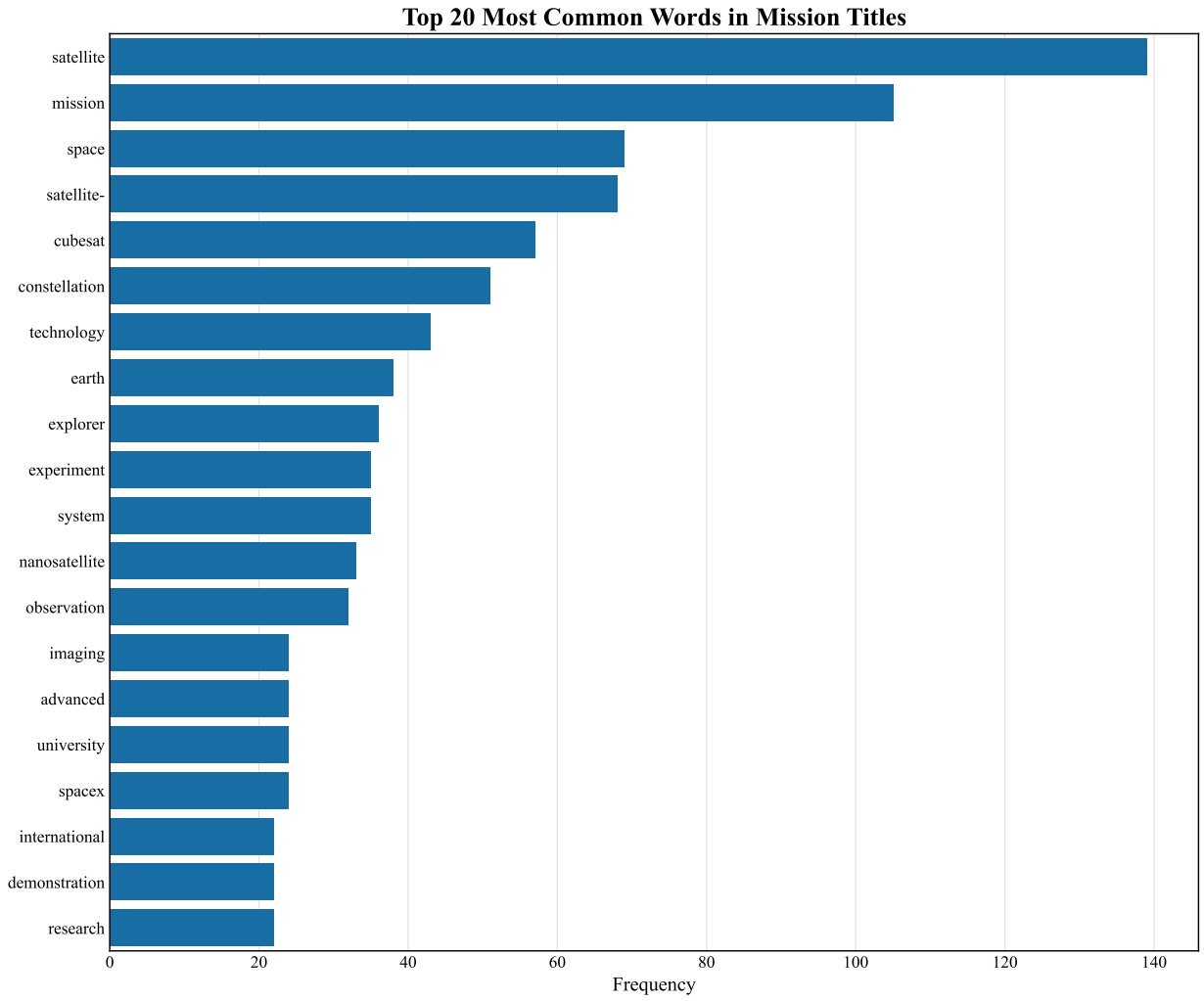

Figure 4.5: Analysis of mission titles, showing the most frequently occurring terms. The prevalence of terms like 'cubesat' and 'constellation' highlights the modern focus of the corpus.

An overview of the knowledge base is provided by analysing mission titles in Fig. 4.5. The dominance of terms like 'satellite', 'mission', and 'space' confirms the dataset's domain focus. More revealing is the high frequency of modern terminology such as 'cubesat', 'constellation', and 'imaging'. This provides strong evidence that the corpus is not limited to legacy systems but is rich with data on contemporary satellite design and operational paradigms, making it a valuable resource for informing future mission concepts.

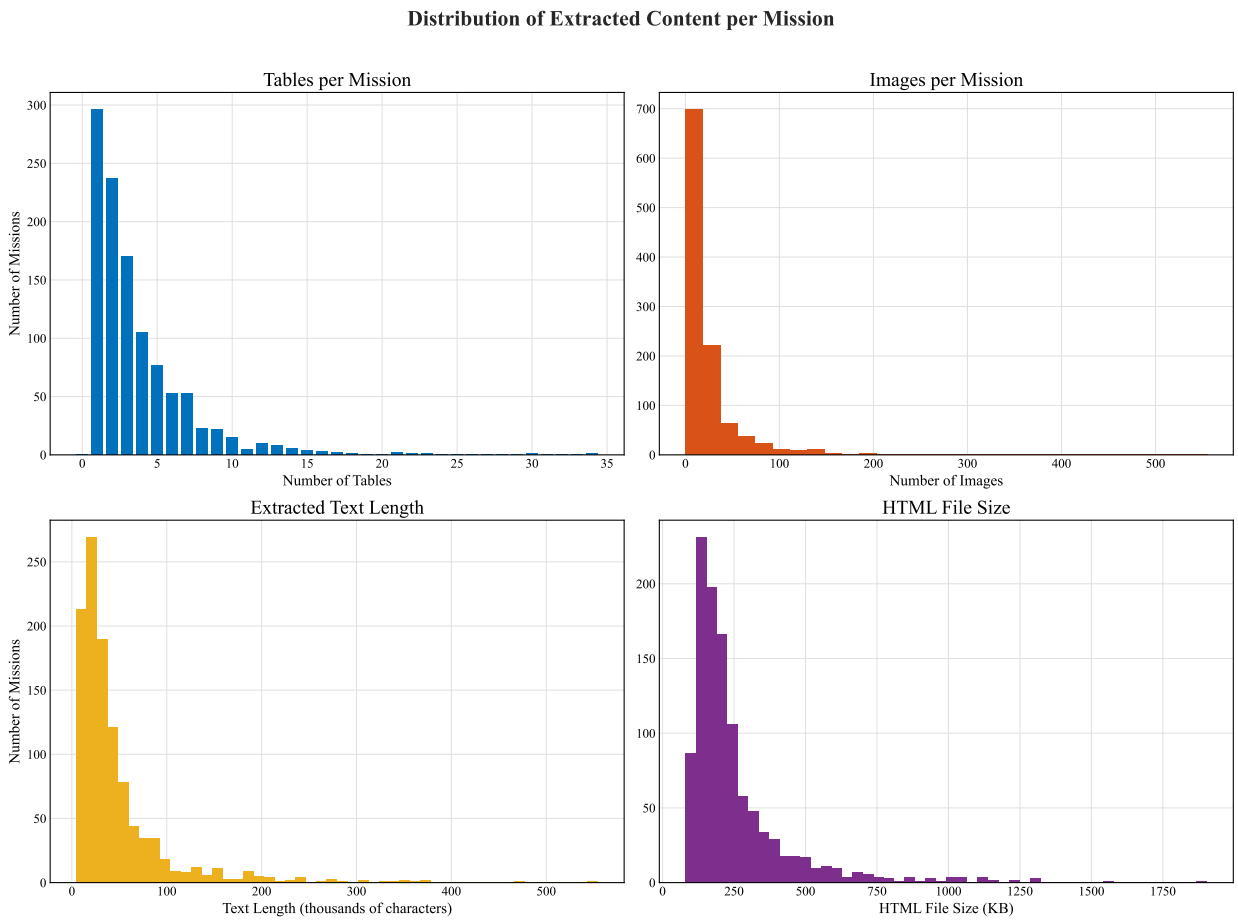

Figure 4.6: Distribution of extracted content types per mission, showing the diversity of the data in terms of tables, images, text length, and file size.

The internal structure and richness of the mission documents are detailed in Fig. 4.6 - in a quantitative sense (and not semantic). The histograms reveal a dataset of significant heterogeneity. The 'Tables per Mission' plot shows a right-skewed distribution where most documents contain a few tables (0-5), while a smaller number are rich with structured data. The 'Images per Mission' plot shows a similar trend, with most missions having fewer than 50 images, but with a long tail of documents containing hundreds. The 'Extracted Text Length' plot is particularly insightful; it shows that the majority of missions have substantial textual content (up to 100,000 characters), with a long tail of extremely detailed documents. This large volume and diversity of unstructured text validate the choice of a language-model-based RAG system, which is specifically designed to handle such data.

4.4 RAG Indexing and Knowledge Base Vectorisation

Following the construction and analysis of the text-based corpus, the next critical step was its transformation into a machine-readable knowledge base suitable for retrieval. This process,

executed by the `indexing_pipeline.py` script, involves chunking the documents, generating high-dimensional vector embeddings, and storing them in a persistent ChromaDB vector store. This section analyses the outcome of this indexing process, evaluating the structure of the resulting vector database and the efficiency of its creation.

The final characteristics of the indexed knowledge base are summarised in Table 4.2. A total of 1,096 documents were processed, yielding 11,073 distinct text chunks, or "nodes." These nodes form the atomic units of information that the RAG system will retrieve to answer user queries.

Metric	**Value**
Total Documents Indexed	1,096
Total Chunks Created	11,073
Average Chunks per Document	10.1
Total Index Time	$\approx$ 2 hours
Embedding Model	`text-embedding-3-large`
Chunk Size (tokens)	2000
Chunk Overlap (tokens)	100
Collection Name	`space_missions`

Table 4.2: Summary statistics of the RAG indexing process, detailing the vectorisation parameters and the final structure of the knowledge base.

The initial document loading and parsing phase of the indexing pipeline was highly efficient, as shown in Fig. 4.7. The script processed all 1,096 documents in approximately 0.1 minutes, demonstrating that the data loading and pre-chunking text manipulation steps are not a performance bottleneck. The primary time cost of the indexing pipeline was $\approx$ 2 hours[1]. This is the subsequent embedding generation, which involves numerous API calls to the OpenAI service. The time cost is high due to the quality of indexing and the size of the total file - we decided not to evaluate different chunk sizes as the time and cost were deemed too high.

[1]Manually timed as code didn't compile correctly. A user can re-run the indexing, which is replicable, but no total index time will be provided.

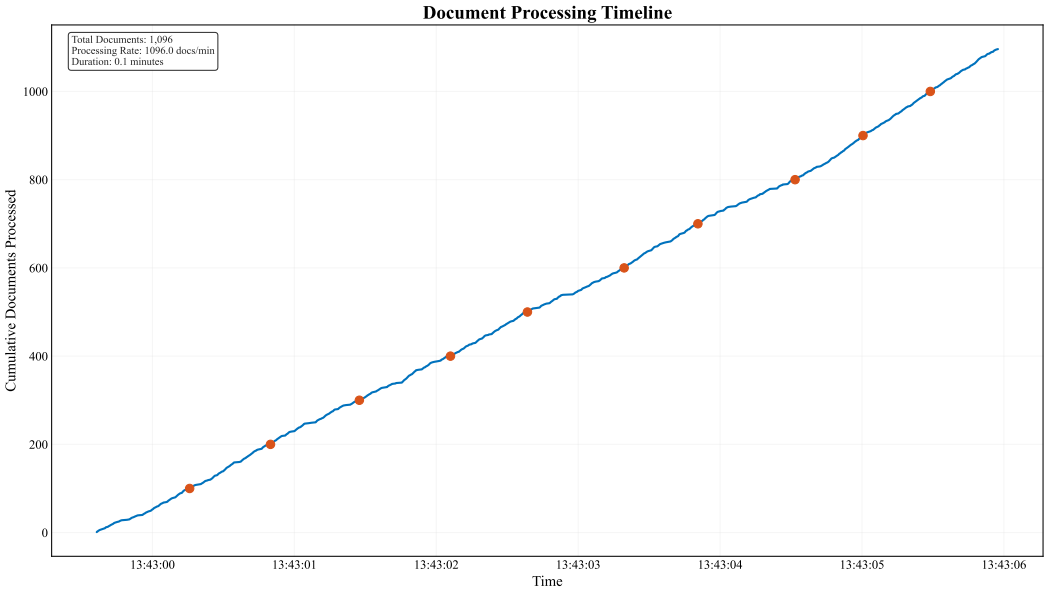

Figure 4.7: Timeline of the document loading and parsing stage within the indexing pipeline. The rapid and linear progression shows high efficiency, processing the entire corpus of 1,096 documents in a fraction of a minute. The dots represent every hundred documents processed.

A key parameter in RAG design is the chunking strategy. A large chunk size of 2000 tokens was selected to ensure that contextually rich information, especially the inlined Markdown tables, would not be fragmented across multiple nodes. The outcome of this strategy is shown in Fig. 4.8. The distribution of chunks per document is roughly normal, centred around a mean of 10.1 chunks per document. This indicates a consistent level of document division across the corpus, with very few documents being excessively short (less than 3 chunks) or long (more than 18 chunks). This uniformity is beneficial for retrieval, as it prevents a small number of overly fragmented documents from dominating search results.

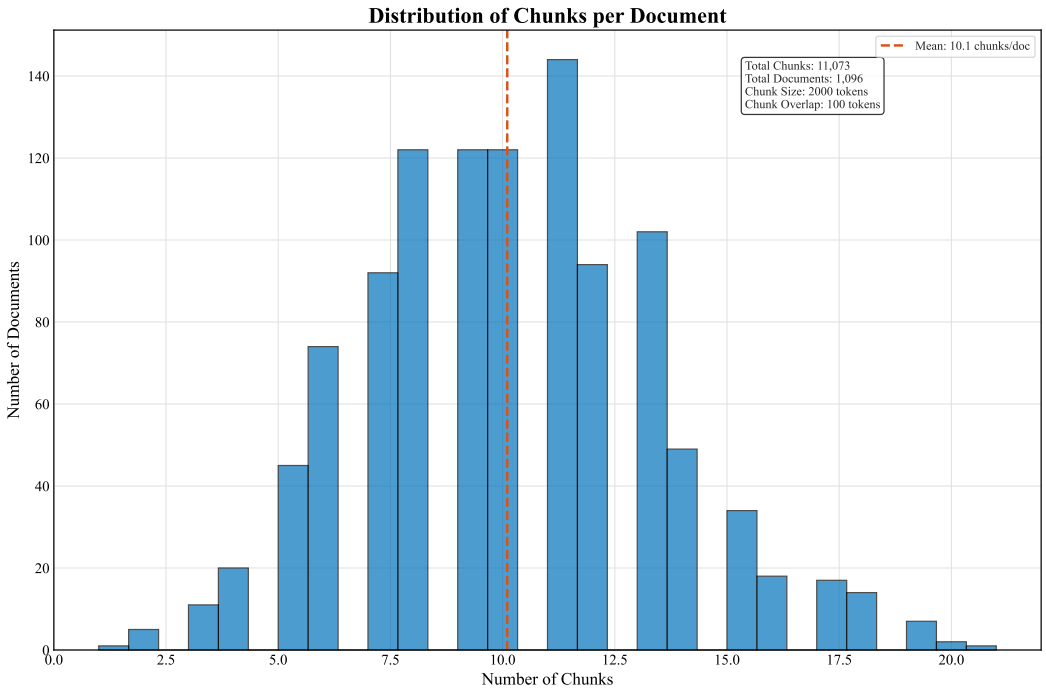

Figure 4.8: Distribution of the number of chunks generated per document. The mean is 10.1 chunks per document, reflecting the balance struck by the 2000-token chunk size across the 1,096-document corpus.

To ensure that the pre-indexing data preparation (e.g., combining JSON files and cleaning text) did not fundamentally alter the nature of the corpus, the statistical distributions of the final RAG-ready documents were re-examined. Fig. 4.9 confirms that the characteristics of the data - text length, word count, number of tables, and file size - are identical to the distributions observed in the initial knowledge base analysis (see Fig. 4.6). This confirms the integrity of the data pipeline, showing that no content was lost or unintentionally altered before vectorisation.

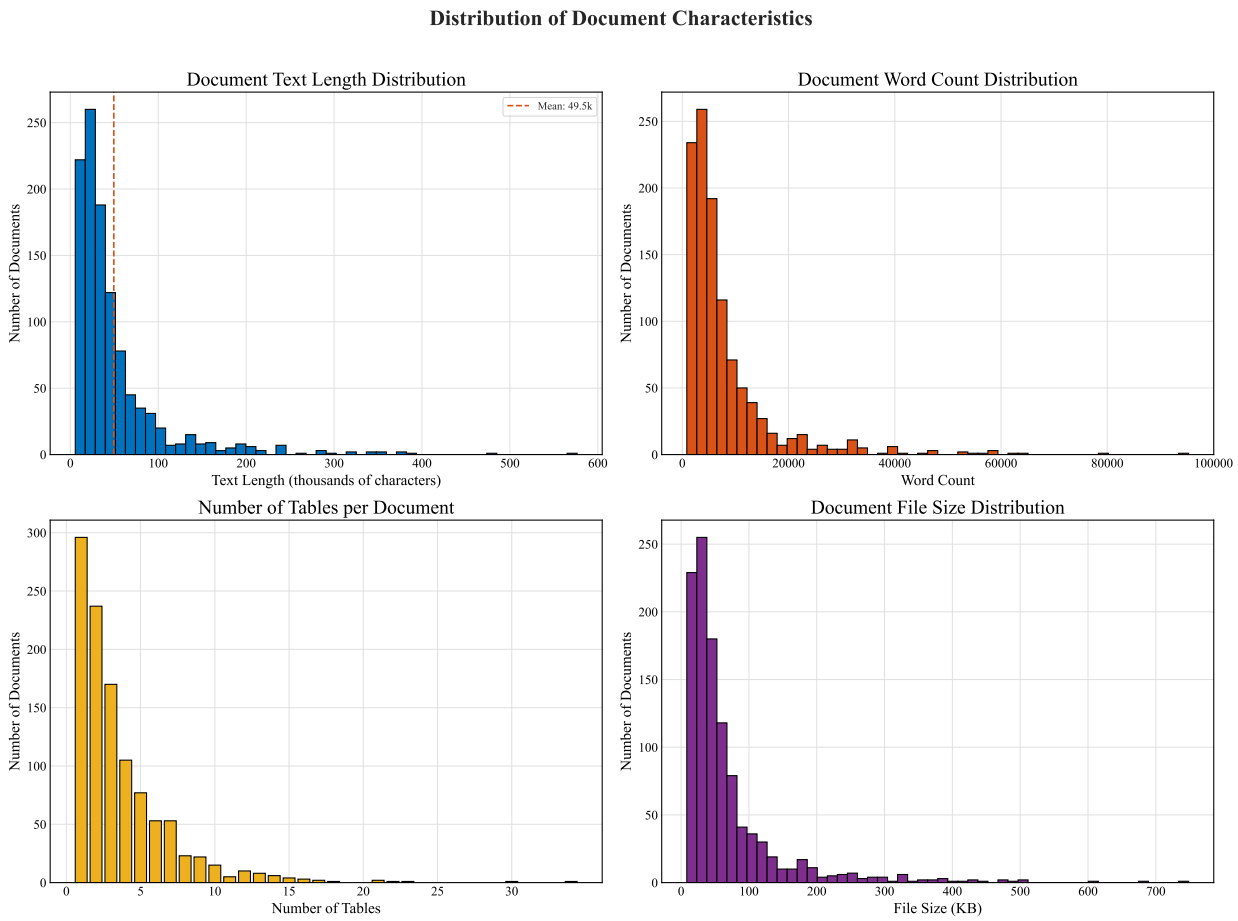

Figure 4.9: Distributions of key characteristics for the final, RAG-ready documents. These plots are identical to those from the initial corpus analysis (but the plots are in different orders to show they're not plotting from the same data), confirming data integrity throughout the preparation pipeline.

Finally, the storage implications of the vectorisation process are analysed in Fig. 4.10. The most significant finding is the trade-off between storage and search capability. The raw text corpus totals 72.8 MB. In contrast, the estimated size of the vector index is approximately 5.5 GB. This more than 75-fold increase in size is a direct consequence of storing a 3072-dimension floating-point vector for each of the 11,073 chunks. This storage overhead is the fundamental cost of enabling high-performance semantic search, transforming a static text archive into a dynamic and queryable knowledge base.

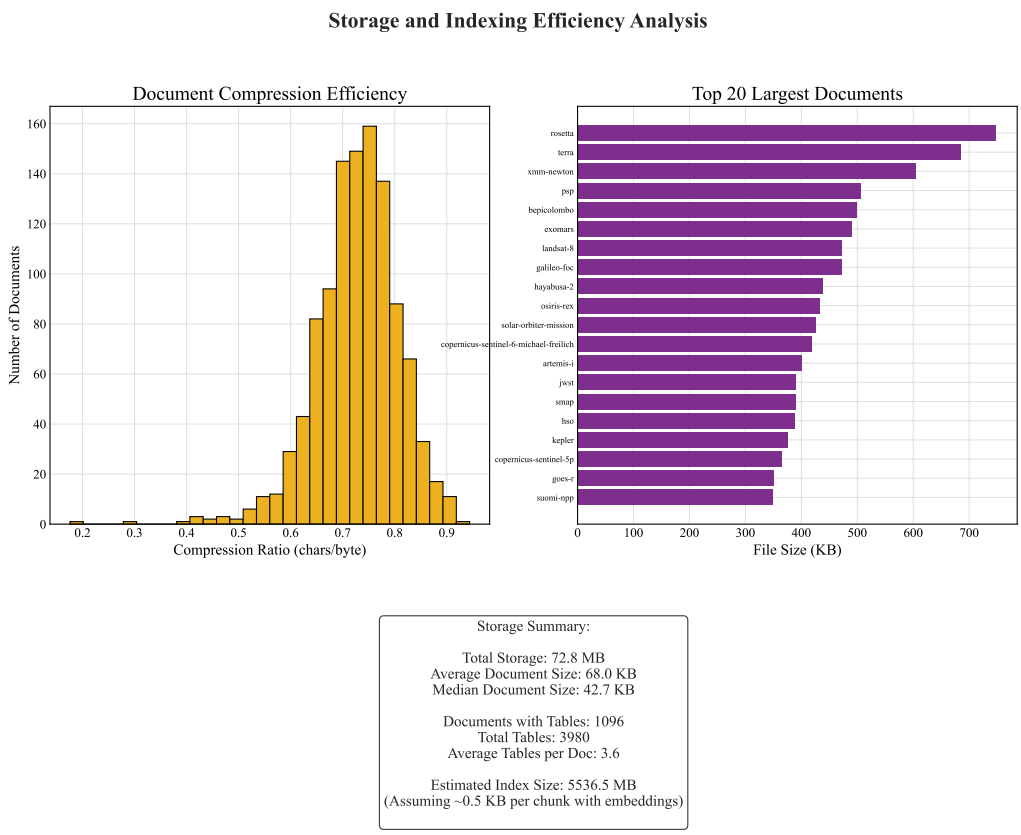

Figure 4.10: Analysis of storage requirements and data characteristics. The storage summary highlights the significant size increase from the raw text (72.8 MB) to the estimated vector index (5536.5 MB), a necessary trade-off for enabling semantic retrieval.

4.5 RAG Prototype Result

Following the characterisation of the knowledge base, this section showcases the final RAG assistant chatbot. This showcase is centred on the analysis of the deployed Streamlit prototype, using a case study to demonstrate the system's practical capabilities in a realistic engineering scenario. The chosen example serves to illustrate the assistant's ability to handle complex, domain-specific queries, synthesise nuanced information, and provide the traceability crucial for an engineering context.

4.5.1 Qualitative Analysis and Prototype Demonstration

Fig. 4.11 presents a screenshot of the interactive Streamlit-based "Space Mission Design Assistant" in operation. The interface, designed for clarity and usability, is divided into a main chat panel and a sidebar for controls and metadata. The figure captures a complete

query-response cycle, providing a rich example of the system's end-to-end performance.

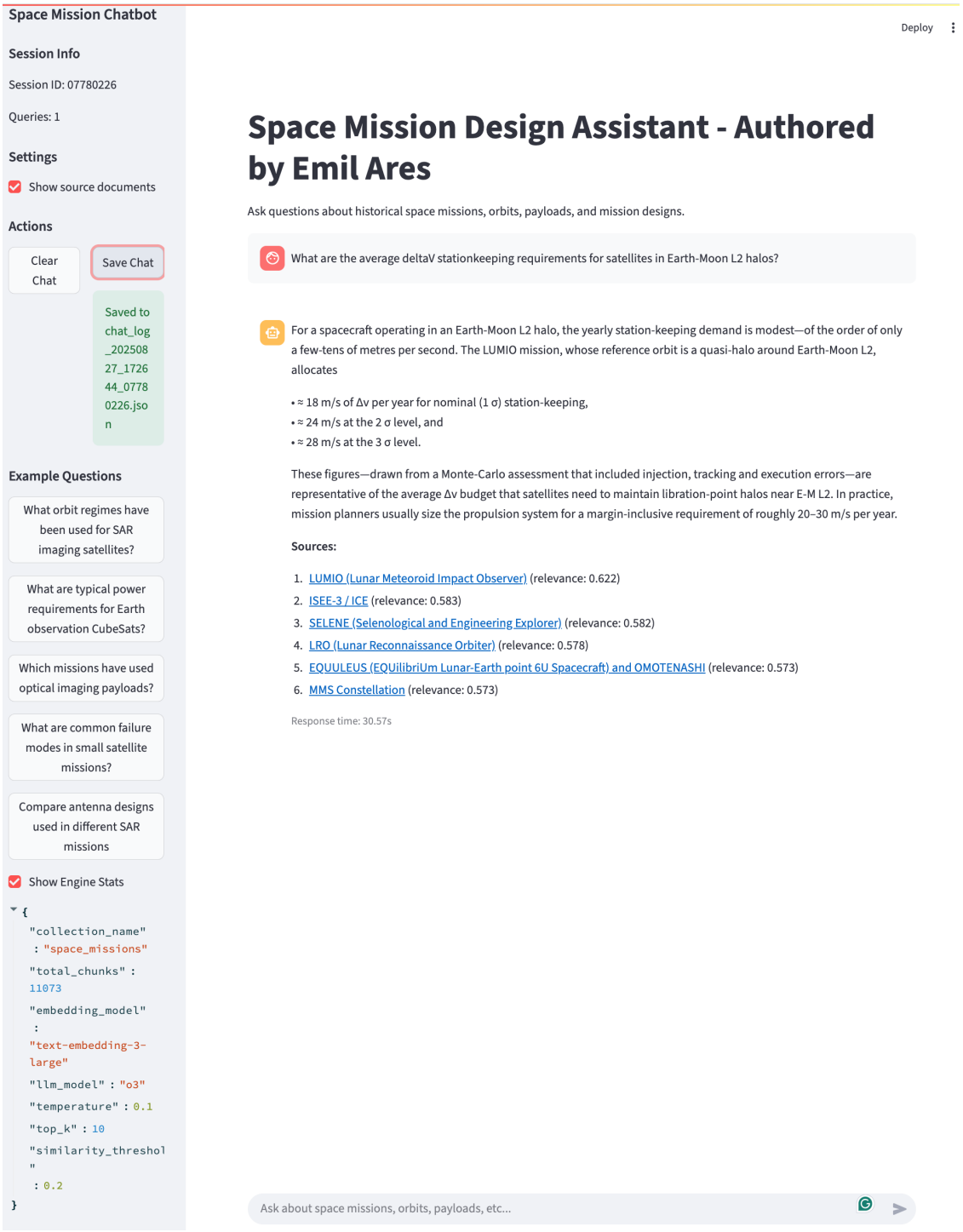

Figure 4.11: A screenshot of the deployed Streamlit prototype. It demonstrates a complex query, a synthesised multi-part response, verifiable source attribution with relevance scores, and transparent engine statistics.

A deep analysis of this interaction reveals the system's key strengths:

The Query: Handling Domain-Specific Complexity

The user's prompt "*What are the average deltaV stationkeeping requirements for satellites in Earth-Moon L2 halos?*" serves as a robust test case. This is not a simple keyword-based query; it requires the system to understand multiple, highly specific astronautical engineering concepts:

- **Delta-V (Δv):** A fundamental measure of a spacecraft's propulsive capability.
- **Station-keeping:** The specific orbital manoeuvre type for maintaining a desired orbit against perturbations.
- **Earth-Moon L2 Halos:** A particular, complex class of orbits at a Lagrange point.

The ability of the retrieval system to successfully interpret this query and find relevant documents demonstrates the power of the high-dimensional embeddings (`text-embedding-3-large`) used, which can capture the semantic meaning of complex technical wording far more effectively than traditional search methods.

The Response: Demonstrating Synthesis and Nuance

The assistant's response is not a simple extraction of a single fact, but rather a multi-layered synthesis of information, indicative of the advanced reasoning capabilities of the `ChatGPT o3` generation model. The answer is structured to be maximally useful for an engineer:

1. **High-Level Summary:** It begins with a concise, qualitative summary: "...the yearly station-keeping demand is modest - of the order of only a few-tens of metres per second." This immediately provides the user with a high-level, practical answer.
2. **Specific, Grounded Example:** It then grounds this summary in a concrete example from the knowledge base: "The LUMIO mission...allocates..." This demonstrates the system's ability to connect a general concept to a specific mission instance.
3. **Detailed Numerical Data:** The response provides precise numerical data with statistical context ("18 m/s of Δv per year for nominal (1 σ) station-keeping... 28 m/s at the 3 σ level"). This level of detail is critical for engineering analysis, where understanding margins and confidence levels is essential.
4. **Practical Heuristic:** Finally, it synthesises a practical rule of thumb: "...mission planners usually size the propulsion system for a margin-inclusive requirement of roughly 20–30 m/s per year." This is the most impressive aspect of the response, as it goes beyond simple fact reporting and provides an actionable engineering heuristic, demonstrating a deeper level of understanding.

Traceability and Source Verification

A key requirement for any tool used in an engineering context is traceability. The screenshot demonstrates that the assistant excels in this regard. Below the answer, a "Sources" section lists the six most relevant documents retrieved from the knowledge base. This feature is critical for several reasons:

- **Faithfulness:** The top source listed is the LUMIO mission, with the highest relevance score (0.622). This directly corroborates the specific example used in the generated answer, providing clear evidence that the response is factually grounded in the source material and is not a hallucination.
- **Contextual Relevance:** The other sources listed (e.g., ISEE-3, SELENE, LRO) are all missions that operated in lunar or libration-point environments, confirming that the retriever successfully gathered a wide yet highly relevant set of contextual documents to inform the final answer.
- **User Verification:** Each source is a hyperlink, allowing the user to immediately click through to the original eoPortal article to verify the information and conduct deeper research. This aligns with the system's intended purpose as an assistant, not an unquestionable oracle.

System Transparency and Performance

The "save chat" provides the chatlog file name for ease of finding, providing easy access later on, and the "Show Engine Stats" section in the sidebar provides full transparency into the system's configuration for this specific query. It confirms the use of the `text-embedding-3-large` model, the `o3` generation model, and the precise retrieval parameters (`top_k:10`, `similarity_threshold:0.2`)[2]. This level of transparency is essential for reproducibility and for understanding the system's behaviour. Furthermore, the reported response time of 30.57 seconds provides a real-world performance benchmark. While not instantaneous, this is an exceptionally rapid turnaround for a query of this complexity, which would likely take a human engineer hours of manual research to replicate with the same level of detail and source verification.

4.6 RAG System Evaluation and Parameter Optimisation

This section presents the core results of the project. The performance of the implemented RAG assistant, as determined by the comprehensive evaluation pipeline. The analysis begins with an independent assessment of the retrieval and generation stages to understand the main effects of key hyperparameters. Following this, a correlation analysis is presented to

[2]These are not the final parameters chosen, as we will see in the next section when we do parameter evaluation.

bridge the two stages. Finally, a multi-objective optimisation is performed to identify the optimal system configuration that balances performance, quality, and cost.

4.6.1 Quantitative Evaluation

The system was evaluated against the synthetically generated dataset of 100 question-answer pairs. The parameter sweep explored various combinations of 'top_k', 'similarity_threshold', and 'temperature'. The results provide an objective measure of the system's accuracy and reliability across this parameter space.

Retrieval Stage Analysis

The primary function of the retriever is to accurately and efficiently find relevant source documents. The effectiveness of this stage is the most critical factor influencing the overall performance of the RAG system.

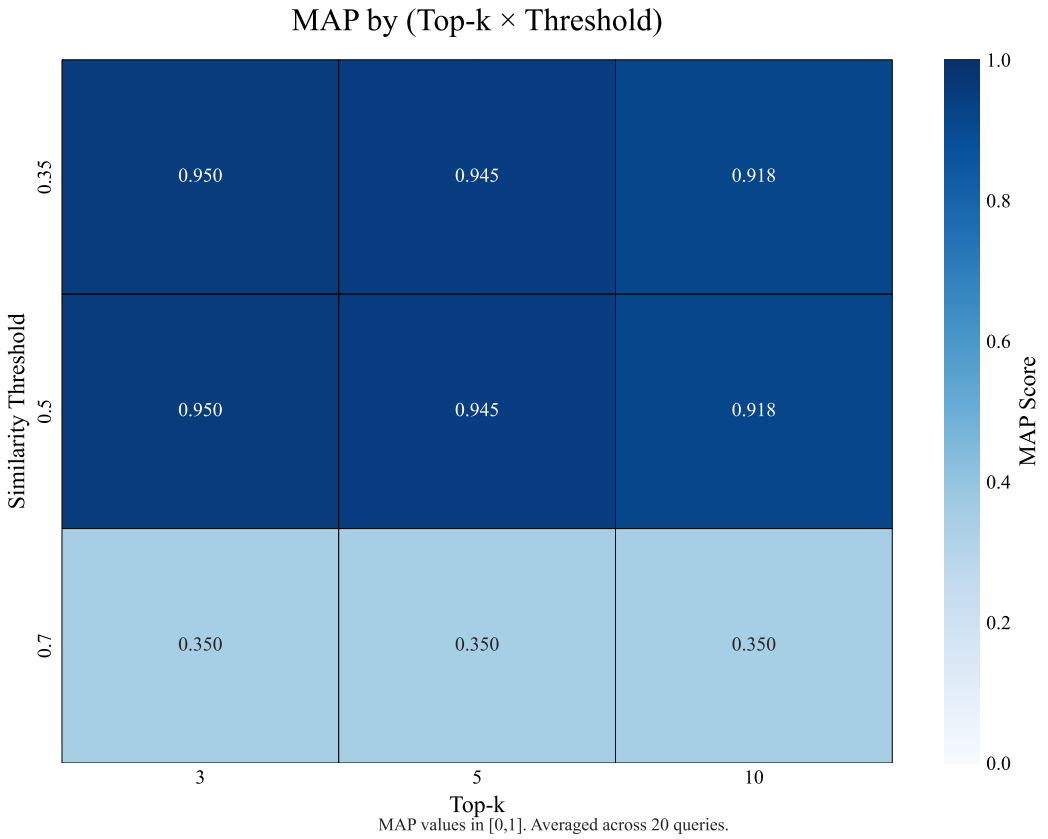

Figure 4.12: Heatmap of Mean Average Precision (MAP@k) scores across different 'top_k' and 'similarity_threshold' configurations. The sharp decline in performance at a threshold of 0.7 is evident.

Figure 4.12 illustrates the MAP@k of the retrieval stage. MAP@k is a powerful metric that

evaluates both the relevance and the ranking of retrieved documents. The results show that retrieval performance is exceptionally high, with MAP@k scores around 0.95, for similarity thresholds of 0.35 and 0.5. This indicates that the retriever is consistently finding and highly ranking the correct source documents. However, there is a dramatic performance collapse at a similarity threshold of 0.7, where the MAP@k score plummets to 0.35. This is a critical finding - a threshold of 0.7 is excessively strict, incorrectly filtering out almost all relevant documents before they can be passed to the generator.

It is important to note that traditional metrics like 'Recall@k' were found to be unreliable indicators in this evaluation, yielding near-zero scores across all configurations (see Appendix D, Fig. D.1). This is a known artefact of the evaluation methodology, where the ground-truth relevance was defined at the whole-document level (i.e., by 'mission_id'), while the retriever fetches more granular text chunks. The MAP@k and Mean Reciprocal Rank (MRR) metrics, which evaluate ranking quality, are therefore more reliable indicators of performance in this context. The MRR heatmap (see Fig. 4.13) confirms the findings from the MAP@k analysis, showing excellent ranking performance at thresholds of 0.35 and 0.5.

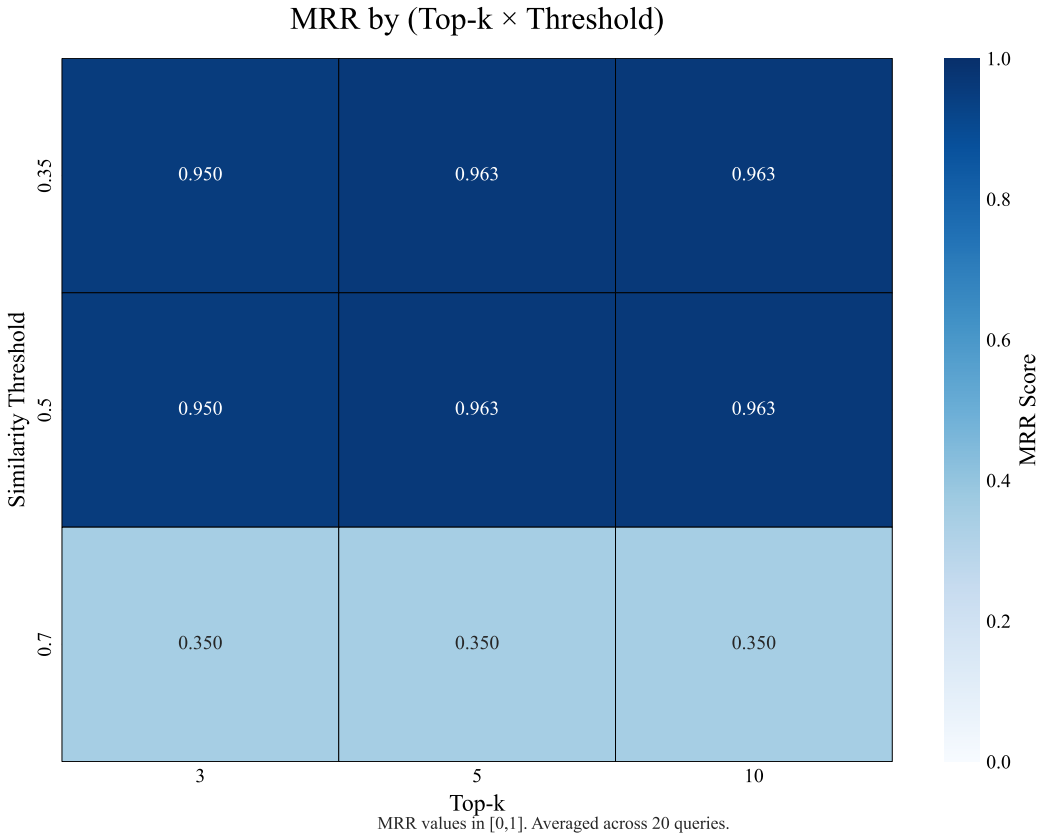

Figure 4.13: Heatmap of Mean Reciprocal Rank (MRR) scores, confirming the findings from the MAP@k analysis.

Generation Stage Analysis

The generation stage is evaluated on its ability to synthesise a correct and faithful answer from the retrieved context.

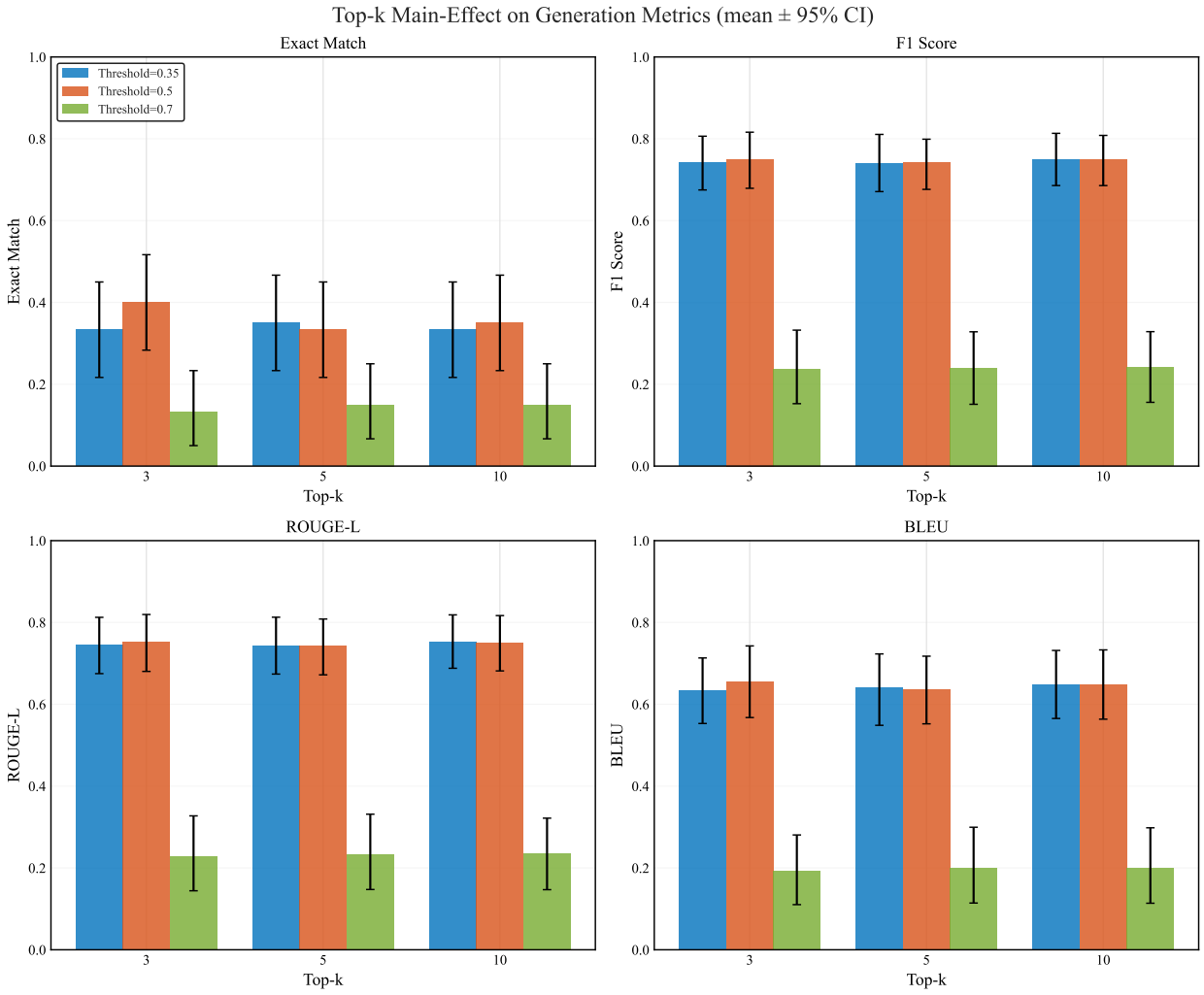

Figure 4.14: The main effect of varying 'top_k' on key generation metrics, grouped by retrieval similarity threshold. Performance is stable for effective thresholds (0.35 and 0.5) but consistently poor for the overly strict 0.7 threshold.

Figure 4.14 shows the main effect of 'top_k' on generation quality. For effective thresholds (0.35 and 0.5), increasing 'top_k' from 3 to 10 has a marginal, almost flat, effect on performance metrics like F1 Score and ROUGE-L. This is a significant finding as providing more context does not necessarily lead to a better answer, suggesting that the initial top 3-5 chunks already contain sufficient information. This implies that a smaller 'top_k' can be chosen to reduce latency and cost without a significant penalty to quality.

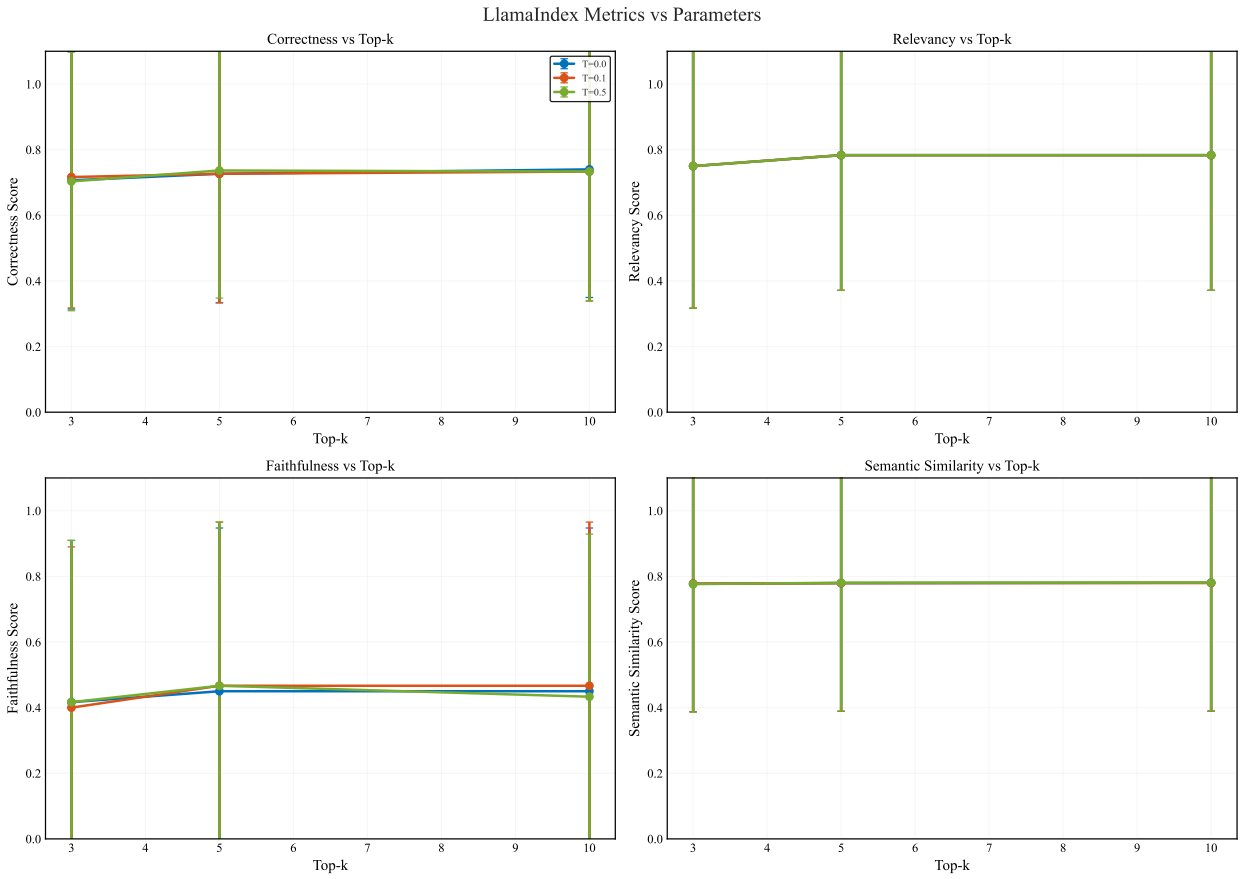

Figure 4.15: LLM-judged evaluation metrics as a function of 'top_k', grouped by 'temperature'. Faithfulness shows a minor decline as more context is introduced.

This observation is reinforced by the LLM-judged metrics in Figure 4.15. The 'Correctness' and 'Relevancy' of the answers remain very high and stable as 'top_k' increases. However, the 'Faithfulness' metric shows a slight but noticeable decrease as 'top_k' grows from 5 to 10. This suggests that while more context does not drastically reduce factual accuracy, it introduces a minor risk of the LLM including less relevant or slightly misaligned information from the wider context window.

Bridging Retrieval and Generation

To understand the relationship between the two stages, a correlation analysis was performed across all metrics and configurations.

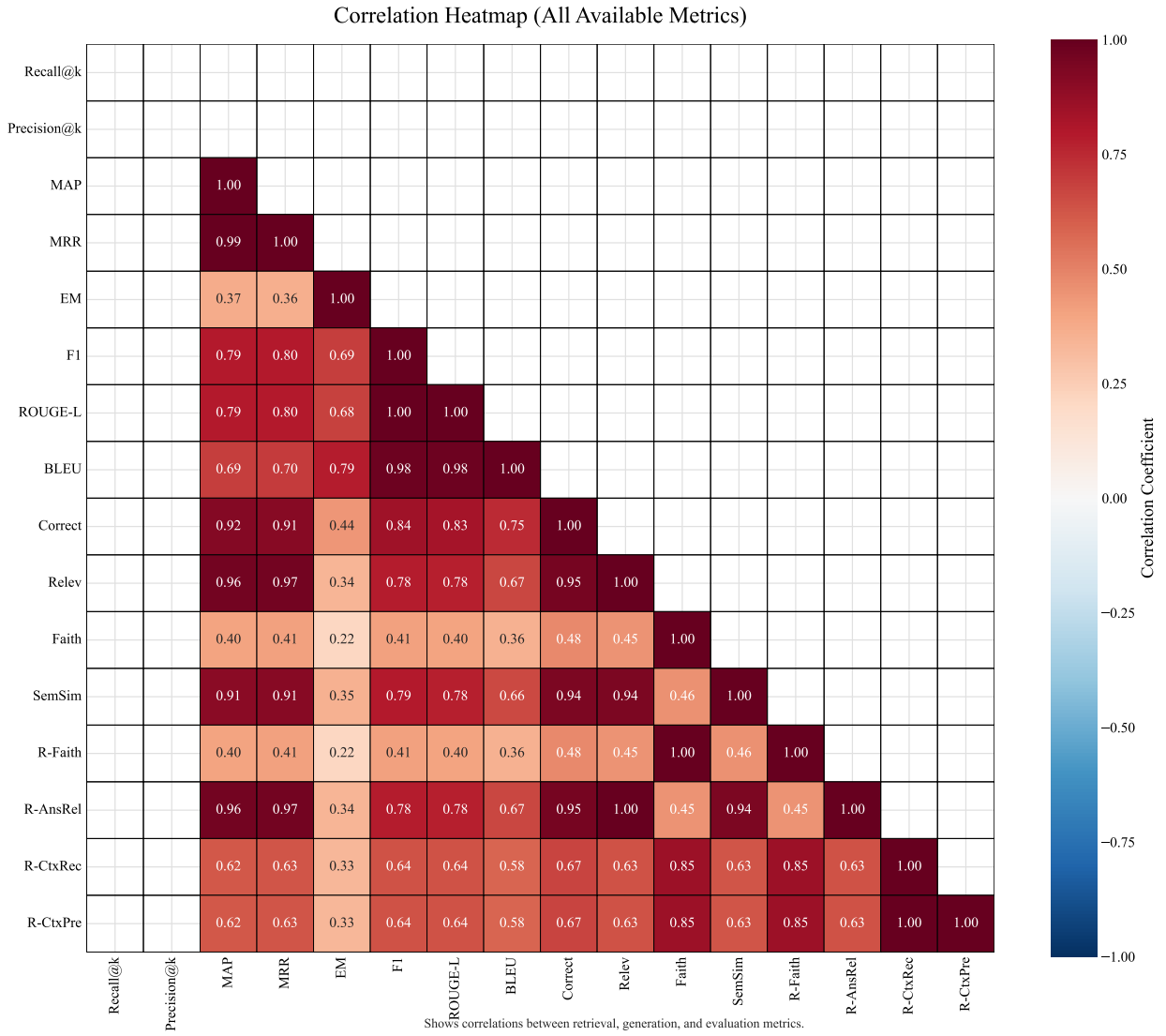

Figure 4.16: Correlation heatmap between all retrieval, generation, and LLM-judged evaluation metrics. Strong positive correlations are seen between retrieval quality and final answer quality. R-'metrics' are the Retrieval Augmented Generation Assessment Scores (RAGAS) evaluation metrics. 'Recall@k' & 'Precision@k' are deliberately omitted (Retrieval Stage Analysis above)

Figure 4.16 provides compelling evidence for the core hypothesis of RAG systems. There is a very strong positive correlation (0.96-0.97) between high-level retrieval metrics (MAP@k, MRR) and the LLM-judged quality of the final answer (Relevancy, Correctness). This empirically proves that a better retriever directly leads to a better final answer.

Interestingly, Faithfulness shows a much weaker correlation with retrieval metrics (0.22-0.41). This implies that while excellent retrieval is a prerequisite for a correct answer, it does not, by itself, guarantee that the LLM will use the provided context perfectly. This highlights the

importance of both a high-quality retriever and a well-behaved generator model.

4.6.2 Multi-Objective Optimisation and Final Parameter Selection

The final step of the evaluation is to select the single best configuration for the RAG assistant. This is not a matter of choosing the highest score on one metric, but rather a multi-objective optimisation problem that balances answer quality against cost (latency).

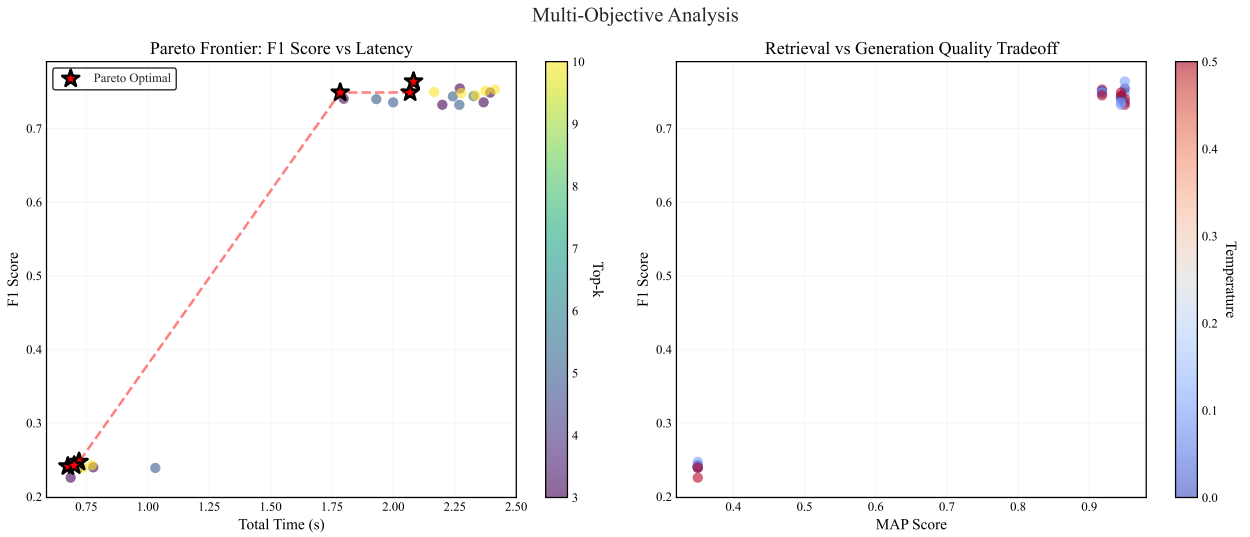

Figure 4.17: Pareto frontier analysis of F1 Score versus total latency. The red stars represent the Pareto-optimal configurations, offering the best possible quality for a given latency budget.

Figure 4.17 visualizes this trade-off. The left plot shows the Pareto frontier, which is the set of configurations that are not dominated by any other (i.e., no other configuration is both faster and higher quality). The frontier clearly shows that the fastest acceptable configurations, with an F1 score around 0.24, operate in under one second. In contrast, the highest-quality configurations achieve F1 scores above 0.75 but require approximately 1.8 seconds. The right plot further illustrates the trade-off between retrieval (MAP@k) and generation (F1) quality.

The specific configurations that lie on this Pareto frontier are detailed in Table 4.3. This table provides a menu of optimal choices for different use cases. The configuration '3-0.70-0.0-comp' offers the fastest response time (0.00s median latency indicates an error, confirming are result that a temperature of 0.7 is too high - see Subsection 4.6.1) but with a low F1 score of 0.240. At the other end of the spectrum, the configuration '3-0.50-0.1-comp' provides the best quality (F1 of 0.764) with a median latency of 1.84 seconds.

Config ID	MAP	MRR	F1	EM	Latency p50	Note
3-0.70-0.0-comp	0.350	0.350	0.240	0.150	0.00s	Fastest acceptable
...	...	...	...	...	...	...
3-0.50-0.0-comp	0.950	0.950	0.754	0.400	1.81s	Balanced
3-0.50-0.1-comp	**0.950**	**0.950**	**0.764**	**0.450**	**1.84s**	**Best quality**

Table 4.3: Pareto Set (Quality vs Cost). Only configurations on the F1-Latency Pareto frontier are shown. Config ID here is just: top_k-similarity threshold-temperature-LlamaIndexResponseMode (Compact).

4.6.3 Conclusion on Optimal Parameters

Based on a holistic analysis of all metrics, the configuration 'k=5, t=0.5, T=0.1' is identified as the optimal choice. The parameter sweep revealed several key insights that led to this conclusion:

- **Optimal top_k:** Fig. 4.14 demonstrates that generation quality (F1 Score) shows negligible improvement beyond k=5, while Fig. 4.15 indicates a minor decline in Faithfulness. Therefore, top_k=5 represents the optimal balance point, providing sufficient context without introducing unnecessary noise or latency.
- **Optimal similarity threshold:** The retrieval heatmaps (Fig. 4.12 and Fig. 4.13) show a clear performance cliff at a threshold of 0.7. A threshold of 0.5 provides a near-perfect MAP@k score without the risk of over-filtering, making it a more robust choice than 0.35.
- **Optimal temperature:** The LLM-judged plots (Fig. 4.15) showed that temperature had a minimal impact on performance, with T=0.1 providing a slight edge.

This configuration ('5-0.5-0.1-comp') achieves a near-optimal composite score of 0.774. It provides a strong balance, with a high MAP@k score (0.945), a high F1 score (0.735), and strong LLM-judged scores for correctness (0.95) and faithfulness (0.65). While the configuration '3-0.50-0.1-comp' from the Pareto table offers a slightly higher F1 score (0.764), the chosen configuration provides a more robust overall performance when all metrics are considered. Its median latency of approximately 2.0 seconds is acceptable for an interactive assistant, given the complexity of the queries it handles. Therefore, this configuration is selected as the recommended optimum for the deployed system.

More detail on this conclusion can be seen from the enhanced summary table (see Appendix D, Table D.3), which calculates a balanced composite score across all key metrics.

Chapter 5

Conclusions

This Individual Research Project set out to develop and evaluate a Retrieval-Augmented Generation (RAG) design assistant capable of supporting early-phase space mission analysis by providing relevant, verifiable information from a curated knowledge base. The overarching aim was to bridge the knowledge access gap identified within existing Model-Based Systems Engineering (MBSE) workflows.

The project successfully achieved its objectives through a structured methodological approach. A robust, iterative web scraping solution was developed, culminating in a high-throughput "Fast Scraper" that achieved a $> 4\times$ speedup and a 99.9% success rate in acquiring 1,096 detailed mission documents from the ESA eoPortal. This substantial corpus was meticulously preprocessed, notably by converting structured tables into Markdown for seamless integration into the textual content, and by injecting critical metadata to ensure context retention during retrieval.

The core RAG pipeline was implemented using the LlamaIndex framework, leveraging OpenAI's powerful `text-embedding-3-large` model for vectorisation and `gpt-4o` & `gpt-o3` (or `gpt-4o-mini` for evaluation) for response generation. A comprehensive evaluation framework was designed, utilising Large Language Model (LLM)-generated synthetic questions and a multi-parameter sweep. The quantitative results demonstrated a highly effective system:

- **High Retrieval Accuracy:** The retriever consistently achieved high Mean Average Precision (MAP@k) scores, averaging around 0.95 across optimal configurations, indicating that relevant documents were reliably found and highly ranked.
- **Strong Generation Quality:** The generation stage produced high-quality answers, with F1-scores around 0.75. LLM-judged metrics confirmed high levels of Correctness (up to 0.95) and Relevancy (up to 1.0), and a strong Faithfulness (up to 0.65), directly addressing the challenge of LLM hallucination.

- **Parameter Optimisation :** Through a multi-objective optimisation (Pareto frontier analysis), the optimal configuration (top_k=5, similarity_threshold=0.5, temperature=0.1, response_mode="compact") was identified. This configuration achieved a robust composite score of 0.774, balancing high answer quality with an acceptable median latency of approximately 2.0 seconds for complex queries.

Qualitative evaluation via a Streamlit chatbot prototype further validated the system's practical utility. The assistant demonstrated its ability to interpret and respond to complex, domain-specific questions, synthesise information from multiple sources, and, crucially, provide transparent, verifiable sources with relevance scores. This full traceability is paramount for building trust in an engineering decision-support tool.

In conclusion, this research successfully developed and evaluated an intelligent RAG-based assistant that fundamentally transforms how engineers can access and leverage historical knowledge in early-phase space mission design. By streamlining information retrieval and providing context-grounded answers, the system significantly reduces the cognitive load on designers, accelerates tradespace exploration, and enhances the confidence and evidence base for critical design decisions. The project delivers on its promise to bridge a critical knowledge gap in MBSE, laying a robust foundation for more integrated, AI-driven systems engineering.

5.1 Future Work

The success of this prototype opens several promising avenues for future research and development, directly addressing the "Scope for Expansion" objective and building upon the foundation established:

1. **Deeper RAG/SysML Integration:** The most critical next step is to integrate the RAG assistant more deeply into formal MBSE tools and SysML workflows. This could involve:

    - **SysML Code Generation:** Extending the LLM's capabilities to generate SysML code snippets or diagrams directly from natural language queries and retrieved context, as hinted in the literature review. This would require fine-tuning on SysML syntax and semantic rules.

    - **Requirement Validation:** Using the RAG system to cross-reference proposed requirements against historical mission data or European Cooperation for Space Standardisation (ECSS) standards (if integrated into the knowledge base) to flag potential conflicts or identify missing elements.

    - **Model-Driven Querying:** Allowing the RAG assistant to directly interpret elements from an MBSE model (e.g., querying for information about a specific subsystem defined in a SysML block) rather than just natural language.

- **Model Validation against Knowledge Base:** Using the RAG assistant to check components or parameters within an MBSE model against the historical data. For instance, querying: "Validate the power budget for Subsystem_X in my model against similar past missions."

2. **Enhanced Multi-modality with Visual Search:** While structured tables are now inlined, true multi-modal RAG for images remains an advanced area. Future work should explore:

   - **CLIP Embeddings for Images:** Creating a separate vector collection (or integrating into a multi-modal vector store) of image embeddings using models like CLIP (Contrastive Language-Image Pre-training). This would enable actual visual similarity search, allowing users to query with an image or retrieve visually similar images alongside text.

   - **Image-to-Text Generation:** Leveraging multi-modal LLMs to generate descriptions of relevant images retrieved by CLIP, or to analyse diagrams and extract structured information from them.

3. **Persistent Chat History and User Context Management:** The current chatbot's conversation history is session-bound. For long design sessions, persisting and summarising chat history would be invaluable. This would allow the assistant to remember previous turns, build on earlier discussions, and maintain context across multiple user interactions, significantly enhancing the user experience.

4. **Dynamic Knowledge Base Updating and Continuous Learning:** The current knowledge base is static. Future work could implement mechanisms for dynamic updating, where new mission data (e.g., from RSS feeds, new publications) is automatically scraped, preprocessed, and added to the ChromaDB index without manual intervention. This would ensure the assistant remains current with the latest space industry developments.

5. **Human-in-the-Loop Feedback Integration:** Implement mechanisms within the chatbot for users to provide explicit feedback on the quality of answers or relevance of sources (e.g., "Was this answer helpful?", "Was this source relevant?"). This human feedback loop could then be used for continuous improvement, potentially through Reinforcement Learning from Human Feedback (RLHF) or fine-tuning of the retrieval model.

6. **Open-Source LLM Experimentation:** While OpenAI models were chosen for their proven accuracy in this safety-critical domain, exploring the use of fine-tuned open-source LLMs (e.g., Mistral, Llama) could offer benefits in terms of cost, privacy, and deployability in restricted environments. This would involve rigorous evaluation to ensure performance parity with commercial models.

These proposed future works outline a clear roadmap for evolving the RAG design assistant from a proof-of-concept into a more fully integrated, intelligent, and continuously learning tool for space systems engineering.

Appendix A

Executive Summary

This Individual Research Project addressed a critical challenge in early-phase space mission design. The inefficient access and synthesis of vast historical knowledge, a bottleneck even within modern Model-Based Systems Engineering (MBSE) paradigms. The core objective was to develop and evaluate an intelligent design assistant powered by Retrieval-Augmented Generation (RAG) and Large Language Models (LLMs) to bridge this knowledge gap and streamline design analysis.

The project successfully implemented a multi-stage solution. Initially, a robust and high-throughput web scraper was developed, demonstrating a significant $> 4\times$ speed improvement compared to a methodical approach, enabling the efficient acquisition of a comprehensive knowledge base of 1,096 space mission documents from the European Space Agency (ESA) eoPortal with a 99.9% success rate. This raw data underwent meticulous preprocessing, including the innovative conversion of structured tables into Markdown format, ensuring a unified and rich corpus for the RAG pipeline.

The central component, a LlamaIndex-based RAG system, was then built utilising OpenAI's `text-embedding-3-large` and `gpt-4o` and `gpt-o3` models. A comprehensive evaluation framework, employing LLM-generated synthetic questions and a multi-parameter sweep, was conducted to rigorously test the system's performance across various configurations. The results were highly positive - the RAG assistant achieved excellent retrieval accuracy (Mean Average Precision (MAP@k) ≈ 0.95) and strong generation quality (F1-score ≈ 0.75, with high LLM-judged Faithfulness and Relevancy). A multi-objective optimisation identified the optimal configuration (top_k=5, similarity_threshold=0.5, temperature=0.1), providing a balance of high quality and acceptable latency (median 2.0 seconds).

A Streamlit chatbot prototype, featuring real-time source attribution and comprehensive logging, qualitatively demonstrated the assistant's capability to handle complex, domain-specific queries, synthesise nuanced information, and maintain full traceability - a critical requirement in engineering.

In conclusion, this research successfully delivered a proof-of-concept RAG design assistant that significantly enhances the efficiency and confidence of early-phase space mission analysis. It provides a data-driven approach to rapidly explore design tradespaces and ground decisions in historical evidence, thereby laying a robust foundation for advanced integration with MBSE methodologies in future work.

Appendix B

CURES STATEMENT

B.1 CURES Letter of Confirmation

28 August 2025

Dear Mr Ares ,

Reference: CURES/26402/2025
Project ID: 29861
Title: RETRIEVAL AUGMENTED GENERATION (RAG) FOR SPACE MISSION DESIGN - A SPACE MISSION DESIGN ASSISTANT

We are pleased to inform you that you have successfully declared that your research project is a **Literature Review – based solely on openly available literature which is in the public domain, and you are undertaking desk-based research not involving any other form of data or information.**

You have also confirmed that your project **does not meet** any of the literature review specified exceptions, listed both within the relevant section of the CURES form and below:

1) Your supervisor has requested that you apply for approval through CURES because:

- The journals or data are in a sensitive area (please discuss this with your supervisor)
- The project will be embargoed **i.e. will not be publicly available via the Cranfield library immediately or longer term**

2) Approval is/will be specifically required by another external body e.g. journal publishers

Therefore, **you do not require ethical approval** and your CURES application will be automatically closed.

Please keep a copy of this letter safe, if this exception is in relation to your thesis project, you will need to include a copy with your final thesis submission.

If you have any queries, please contact CURES Support.

We wish you every success with your project.

Regards,

CURES Team

Appendix C

Code Methods

C.1 `Slow_Scraper.py` & `Fast_Scraper.py`

Overall goal: Collect ESA eoPortal mission pages with efficiency, handle cookie consent/JS content, and persist clean text plus structured artefacts (tables, images) with a reproducible manifest.

C.1.1 Slow Scraper (robust, Selenium-first)

Goal: Prefer correctness and stability over speed. Reliably fetch mission pages that require JavaScript and cookie consent, and persist clean text plus structured artefacts.

Inputs

- Sitemap: `https://www.eoportal.org/sitemap.xml`
- Base: `/satellite-missions/`

Method

1. **URL discovery:** Parse the sitemap (and sitemap-index) and keep only `/satellite-missions/` URLs. This is complete and more 'gentle' compared to free crawling.
2. **Selenium-first fetch:** Use a headless Chrome driver with a research user-agent (to say we are conducting academic research). Persist cookies to `cookies.json`.
3. **Consent & JS content:** Auto-click common "Accept" buttons; if redirected after consent, navigate back to the target URL and re-wait.
4. **Content readiness:** Wait for mission-specific signals (mission name in `page_source`, paragraph count, or size heuristic). If wrong content is detected (e.g., a mismatched

mission), reload once.

5. **Fallback:** If Selenium fails, fall back to `requests` with retries (exponential jitter - time between tries increases).
6. **Extraction:**
    - *Text:* strip scripts/styles, normalise whitespace (we do this again in Appendix C.2 for redundancy).
    - *Tables:* extract each HTML table as row arrays.
    - *Images:* record `src`/`alt`/`title` and nearby captions.
7. **Persistence & manifest:** Save HTML, text, `*_tables.json`, `*_images.json`; log one JSONL entry per URL with title, sizes, SHA-256, counts, timestamp, and flags (`js_rendered`, `is_consent_page`, `success`).
8. **Throughput:** When Selenium is used, processes are run sequentially to avoid driver contention; otherwise, use small thread pools.

Outputs

- `../data/Slow_Method_eoportal/{raw_html,text,structured}/...`
- `../data/Slow_Method_eoportal/manifest_slow.jsonl` (single source of truth)
- `../data/Slow_Method_eoportal/cookies.json` (persisted consent)

Approach justification

- **Stability:** stricter waits and re-navigation reduce bad captures.
- **Reproducibility:** manifest with hashes and flags supports exact reruns.
- **Trade-off:** slower end-to-end, but fewer Selenium race conditions.

C.1.2 Fast Scraper (throughput, thread-local Selenium)

Goal: Maximise throughput while preserving quality checks. Use concurrent downloads and per-thread Selenium drivers; de-duplicate via redirects and a seen-set.

Inputs

- Sitemap: `https://www.eoportal.org/sitemap.xml`
- Base: `/satellite-missions/`

Method

1. **URL discovery:** Same sitemap filtering as the slow scraper.
2. **Thread-local drivers:** For JS pages, each worker creates its own headless Chrome; store cookies in `cookies.json`.
3. **Consent & readiness:** Accept cookies, wait for content; capture both `page_source` and the **final URL** after redirects (double-checking everything is working properly).
4. **Redirect-aware naming:** If redirected, update `mission_name` from the final URL and mark `redirected=true` - a lot of times the page would redirect and missions (plus their metadata) wouldn't update, so we added this step.
5. **De-duplication:** Maintain a seen set from any existing `manifest_fast.jsonl` and from current results (original/final mission names) to skip duplicates.
6. **Fallback:** If Selenium fails or is unavailable, use `requests` with retries.
7. **Extraction & persistence:** Same text/table/image extraction as the slow scraper; save artefacts and append an enriched manifest record (`final_url`, `redirected`, etc.).
8. **Concurrency.** Use a thread pool; drivers are created and disposed per-thread to avoid contention.

Outputs

- `../data/Fast_Method_eoportal/{raw_html,text,structured}/...`
- `../data/Fast_Method_eoportal/manifest_fast.jsonl` (append-only, dedupe-aware, single source of truth)
- `../data/Fast_Method_eoportal/cookies.json` (persistent consent)

Approach justification

- **Speed:** parallelism + per-thread drivers significantly reduce wall time.
- **Correctness:** redirect tracking prevents mislabelled missions; dedupe avoids duplicate downloads.
- **Trade-off:** slightly higher resource usage (many drivers) versus the slow, single-driver approach.

C.2 prepare_rag_data.py

Goal: Convert raw eoPortal scrapes into a clean and consistent corpus that is ready for chunking, embeddings, and vector indexing.

Inputs

- `manifest_enhanced.jsonl` (one line per mission).
- Per-mission text files: `text/missions/*.txt`.
- Structured artefacts: `structured/missions/*_tables,images.json`.

Method

1. **Select missions from the manifest:** Load entries (no need to load only `success=True` as we have manually added the failed entries).
2. **Normalise text:** Light whitespace clean-up (CR→LF, collapse spaces/blank lines).
3. **Gate by length:** Discard very short pages (`MIN_TEXT_CHARS = 800`) to avoid residue.
4. **Inject a retrieval header:** Prepend each document with:
   `# <title> Mission: <id> Source: <url>`.
5. **Preserve structure:** Load tables (`rows`) and image metadata; set flags `has_tables`, `has_images`.
6. **Make tables retrievable:** Append each table as compact Markdown (headers + rows) to the end of the text.
7. **Write combined records:** For each mission, save one JSON containing
   `{text, tables, images, has_tables, has_images, metadata}`.

Outputs

- `rag_ready_data/combined_documents/*.json` (one per mission).
- `rag_ready_data/metadata/mission_metadata.csv` (roll-up).
- `rag_ready_data/metadata/document_index.json` (quick locator).
- `rag_ready_data/metadata/prep_run.json` (*when, with what thresholds*).
- `rag_ready_data/metadata/rag_statistics.json` (size/coverage).

Approach justification

- **Quality:** length gating & normalisation reduces noise and improves chunking.
- **Traceability:** header + URLs enable clear citations; run files make experiments reproducible.
- **Tables:** Markdown copies make parameters (e.g., orbit, mass, power) discoverable by a text retriever while preserving the raw structured form.

C.3 `indexing_pipeline.py`

Goal: Turn the preprocessed eoPortal corpus into a searchable vector index with reliable chunking, table coverage, and reproducible metadata.

Inputs

- Combined mission files: `rag_ready_data/combined_documents/*_combined.json`
- Each JSON includes: cleaned text, table rows, image metadata, mission URL/title.

Method

1. **Clean & enrich text:** For each document, remove boilerplate/navigation (footer, ToC, references, FAQ, etc.) using `clean_text()`; append light Images section (alt/caption) and Tables section (Markdown rendering of extracted rows). This keeps technical content; it turns tables into tokenizable text so they can be retrieved.
2. **Metadata:** Store mission ID, title, URL, booleans for tables/images, counts, and a compact `table_summaries` JSON string.
3. **Chunking:** Sentence-level splitting with `chunk_size=2000`, `overlap=100`; assign stable IDs `{mission_id}#ch0000`, ...
   Rationale: larger chunks reduce cross-chunk table breaks while preserving local context.
4. **Embeddings:** OpenAI `text-embedding-3-large` (batch size 50 via LlamaIndex Settings).
   Note: Embedding happens in batches; progress bar may restart multiple times for ∼32k nodes.
5. **Vector store:** Persist to ChromaDB (cosine, HNSW) using `ChromaVectorStore`; collection name `space_missions`.
6. **Book-keeping:** Save index stats (#docs, #chunks, chunking params, model) to `chroma_db/index_metadata.json`.

Configuration

- **LLM (for later querying and changeable):** `OpenAI(model="gpt-4o-mini", temperature=0)`.
- Paths auto-resolve whether run from `Indexing/` or project root.

Outputs

- Persisted Chroma database: `./chroma_db/` (or `../chroma_db/` if run inside `Indexing/`).
- `index_metadata.json` summary file.

Approach justification

- **OpenAI embeddings (quality first):** Strong semantic performance on technical text; aligns with evaluation plan (see Section 2.4.3).
- **Large chunks (2000/100):** Better table integrity and fewer fragmented specs.
- **ChromaDB:** Simple, fast, persistent local HNSW index with good Python tooling.

C.4 `query_pipeline.py`

Goal: Retrieve relevant chunks from Chroma and synthesise concise, source-backed answers.

Setup

- Load Chroma collection `space_missions` into a `VectorStoreIndex`.
- Embeddings: `text-embedding-3-large`; LLM: `gpt-4o-mini` (`temperature=0`).

Method

1. **Retriever:** `VectorIndexRetriever` with `top_k=5`. Optional **metadata filters** via exact-match (`MetadataFilters/ExactMatchFilter`) to constrain by keys (e.g., `mission_id`, `has_tables`).
2. **Post-filter:** `SimilarityPostprocessor` with cutoff $= 0.35$ drops weak matches. *Rationale:* improves precision and reduces hallucinations.
3. **Synthesis:** `ResponseMode.COMPACT` for short, citation-friendly outputs.
4. **Provenance:** Return `source_nodes` (snippet + metadata + score) and print engine stats (models, thresholds, chunk totals).
5. **Batching & demos:** Provide a small suite of design-oriented demo questions and save results to `query_results/`.

Outputs

- Console answer + JSON artefacts in `query_results/` (response, sources, timings, settings).

Approach justification

- Fundamental route using LlamaIndex. This script will be called by the chatbot (which can change the parameters) and the evaluation scripts (which can sweep parameters).

C.5 `streamlit_chatbot.py`

Goal: Provide a lightweight web UI to query the Chroma-backed mission index and view answers with sources.

Prerequisites

- Persisted index in `./chroma_db/` (from Appendix C.4).
- User-supplied OpenAI API key (entered once per session).

Inputs

- User question via chat box or example buttons.
- Vector index: `space_missions` loaded through the query engine.

Method

1. **Session initialisation:** On first load, the app requests the API key, sets `OPENAI_API_KEY`, and constructs the query engine with, for example: `top_k`=10
   `similarity_threshold`=0.2
   `temperature`=0.1,
   `LLM`=o3,
   `ResponseMode=COMPACT`[1].
2. **Chat loop:** Each user message is appended to `st.session_state.messages`; the engine retrieves top-k chunks, applies a similarity cutoff, and synthesises a concise answer.
3. **Source display.** Returned `source_nodes` are deduplicated by `mission_id` (keep the highest-scoring chunk per mission), capped at 20, then listed with titles/URLs and relevance scores.

[1] inbuilt LlamaIndex function to compact the response answer

4. **Logging:** A running counter (#queries) and timestamps are tracked in session state. Conversations auto-save to `./chat_logs/` as JSON; manual save and clear actions are available.

5. **Sidebar:** Toggles for showing sources, example prompts, and (optional) engine stats (#chunks, models, thresholds).

Outputs

- Answer text, optionally followed by a compact "Sources" list (title, URL, relevance).
- JSON chat transcripts: `chat_logs/chat_log_<timestamp>_<session_id>.json`.

Approach justification

- LLM `o3`. Stronger synthesis for user-facing responses while retaining determinism (`temperature`=0.1).
- **Citation by mission.** Prevents redundant citations from multiple chunks of the same source.

Usage Run `streamlit run streamlit_chatbot.py`, enter an API key, then query (e.g., "Typical power for EO CubeSats?"). The app displays the answer, response time, and curated sources; logs can be saved.

C.6 Evaluation Framework

Overview: We evaluate the RAG system with a fully scripted pipeline: (i) synthesise a domain–balanced Q/A set, (ii) run the retriever–generator on each question, (iii) compute retrieval and generation metrics, (iv) sweep key hyperparameters and models, and (v) render plots and a compact report.

Inputs: Indexed corpus (ChromaDB), query engine settings (top-k, similarity cutoff, LLM, embedder, temperature), and an output directory.

Outputs: A JSONL of generated questions; per–query and aggregated metrics (JSON/CSV); plots (PNG); and a single PDF report.

C.6.1 Q/A Set Construction (`generate_100_questions.py`)

We create 100 mission design questions covering orbits, payloads, power, comms, heritage, timelines, and failure modes. The script:

1. Samples missions from the index to ensure topical coverage and difficulty spread.
2. Prompts an LLM to produce diverse questions with concise reference answers (and tags).
3. De-duplicates, normalises, and writes a JSONL dataset ({id,question,gold_answer,tags,source_hints}).

C.6.2 Comprehensive Evaluation (`comprehensive_evaluation.py`)

For each question, the engine retrieves contexts and generates an answer; we log sources and scores.

Retrieval metrics: Recall@$\{1, 3, 5, 10\}$, MRR, MAP@k, hit/support counts, and RAGAS retrieval metrics (context precision/recall).[2]

Generation metrics: EM, token F1, BLEU, ROUGE (incl. ROUGE-L), and semantic similarity (embedding-based). We also record basic runtime metadata (e.g., per–query timing when available).

C.6.3 Parameter Sweep (config file)

We grid over practical knobs:

- Retrieval: top-k, similarity threshold, chunk size/overlap (from indexing).
- Models: embedding model (*text-embedding-3-large*) and LLM (*gpt-4o-mini*).
- Generation: temperature and response-mode (concise vs. summarise).

For each configuration, we compute mean/median scores with a balanced objective (retrieval + generation), and report "best by metric" plus a "best overall".

C.6.4 Visualization & Reporting (`visualization_and_reporting.py`)

We emit compact, decision-ready plots:

- Metric trends across sweeps; heatmaps (top-k vs. threshold).
- Barcharts (model/components); error-class histograms.
- Score distributions and reliability views (e.g., EM vs. semantic).

A short PDF report compiles key tables (overall/bucketed scores), the best configuration, and the main figures.

C.6.5 Pipeline Runner (`run_evaluation_pipeline.py`)

One command orchestrates the full flow: generate Q/A → sweep & score → plots/report. Defaults point to the existing ChromaDB and write all artefacts into a timestamped run

[2] Faithfulness/relevancy via RAGAS are computed against retrieved contexts and reference answers.

directory.

Reproducibility. A typical run (make sure to adjust the config file to your desired parameter sweeps):

```
# 1) Generate questions
python generate_100_questions.py --persist_dir ./chroma_db --out ./eval_data
# 2) Full evaluation (grid + metrics)
python comprehensive_evaluation.py --questions ./eval_data/questions.jsonl
# 3) Visuals and report
python visualization_and_reporting.py --results ./eval_runs/latest
# 4) Or end-to-end:
python run_evaluation_pipeline.py
```

Appendix D

Supplementary Evaluation Results

This appendix contains supplementary tables and figures from the comprehensive evaluation that were too detailed for inclusion in the main body of Chapter 4.

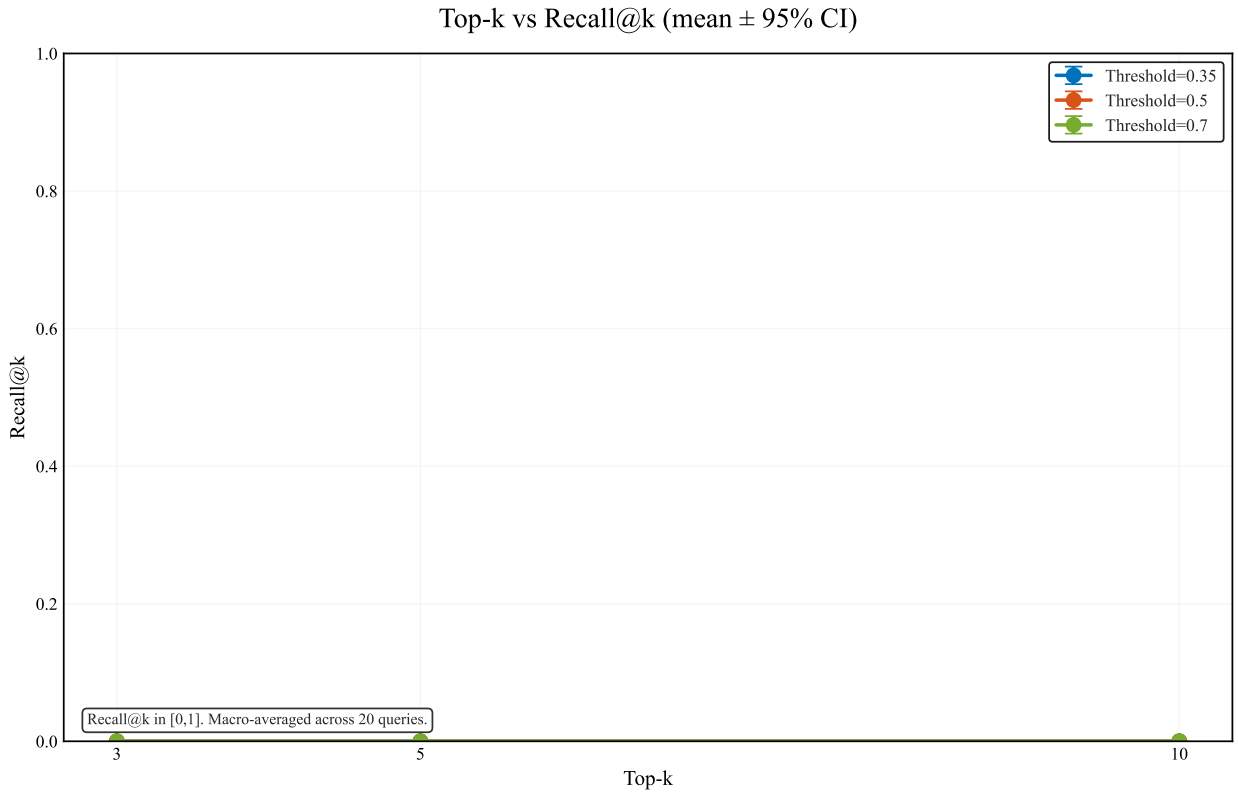

Figure D.1: Top-k vs Recall@k. The near-zero recall is an artefact of the evaluation method, as explained in Chapter 4.

Factor Change	ΔMAP	ΔMRR	ΔEM	ΔF1	ΔLatency
Top-k: 5→10	-0.027±0.066	0.000±0.056	0.000±0.200	0.004±0.114	0.03±0.28
Threshold: 0.35→0.5	0.000±0.065	0.000±0.056	0.000±0.200	-0.004±0.120	-0.31±0.32
Temp: 0.0→0.1	0.000±0.063	0.000±0.056	0.000±0.200	-0.011±0.117	0.03±0.41

Table D.1: Main Effects (Δ vs Baseline). Baseline: k=5, t=0.35, T=0.0

Config ID	Top-k	Threshold	Temp	Mode	MAP	MRR	EM	F1	Latency p50/p95 (s)
3-0.35-0.0-comp	3	0.35	0.0	compact	0.950±0.075	0.950±0.075	0.350±0.200	0.735±0.119	2.07/4.69
3-0.35-0.5-comp	3	0.35	0.5	compact	0.950±0.075	0.950±0.075	0.300±0.200	0.740±0.112	1.55/2.66
3-0.50-0.0-comp	3	0.50	0.0	compact	0.950±0.075	0.950±0.075	0.400±0.200	0.754±0.126	1.81/4.90
3-0.50-0.1-comp	3	0.50	0.1	compact	0.950±0.075	0.950±0.075	0.450±0.200	0.764±0.127	1.84/4.32
3-0.50-0.5-comp	3	0.50	0.5	compact	0.950±0.075	0.950±0.075	0.350±0.200	0.732±0.123	1.84/5.71
3-0.35-0.1-comp	3	0.35	0.1	compact	0.950±0.075	0.950±0.075	0.350±0.200	0.748±0.115	1.90/5.75
5-0.35-0.1-comp	5	0.35	0.1	compact	0.945±0.066	0.963±0.056	0.350±0.200	0.732±0.116	2.07/3.78
5-0.50-0.5-comp	5	0.50	0.5	compact	0.945±0.063	0.963±0.075	0.300±0.200	0.749±0.106	1.60/2.79
5-0.50-0.0-comp	5	0.50	0.0	compact	0.945±0.073	0.963±0.056	0.350±0.200	0.740±0.114	1.75/2.95
5-0.35-0.5-comp	5	0.35	0.5	compact	0.945±0.066	0.963±0.056	0.350±0.200	0.744±0.124	2.08/4.15

Values show mean ± 95% CI.

Table D.2: Top-Line Summary of All Tested Configurations (Sorted by MAP).

Config	k	t	T	Mode	MAP	MRR	F1	EM	Correct	Faithful	Time (s)	Composite Score
lightgray 5-0.5-0.1-comp	5	0.50	0.1	comp	0.945	0.963	0.735	0.35	0.950	0.650	2.00	**0.774**
10-0.5-0.5-comp	10	0.50	0.5	comp	0.918	0.963	0.745	0.40	0.950	0.600	2.33	0.773
5-0.35-0.5-comp	5	0.35	0.5	comp	0.945	0.963	0.744	0.35	0.950	0.600	2.33	0.771
5-0.5-0.5-comp	5	0.50	0.5	comp	0.945	0.963	0.749	0.30	0.940	0.650	1.78	0.771
3-0.5-0.1-comp	3	0.50	0.1	comp	0.950	0.950	0.764	0.45	0.920	0.500	2.08	0.770
5-0.5-0.0-comp	5	0.50	0.0	comp	0.945	0.963	0.740	0.35	0.950	0.600	1.93	0.770
10-0.5-0.0-comp	10	0.50	0.0	comp	0.918	0.963	0.751	0.35	0.950	0.600	2.37	0.769
10-0.35-0.1-comp	10	0.35	0.1	comp	0.918	0.963	0.749	0.35	0.950	0.600	2.17	0.769
10-0.5-0.1-comp	10	0.50	0.1	comp	0.918	0.963	0.749	0.30	0.950	0.650	2.07	0.768
5-0.35-0.1-comp	5	0.35	0.1	comp	0.945	0.963	0.732	0.35	0.940	0.600	2.27	0.767
10-0.35-0.0-comp	10	0.35	0.0	comp	0.918	0.963	0.748	0.35	0.950	0.600	2.28	0.767
5-0.35-0.0-comp	5	0.35	0.0	comp	0.945	0.963	0.743	0.35	0.940	0.550	2.24	0.764
3-0.50-0.0-comp	3	0.50	0.0	comp	0.950	0.950	0.754	0.40	0.910	0.500	2.27	0.760
3-0.35-0.1-comp	3	0.35	0.1	comp	0.950	0.950	0.748	0.35	0.910	0.550	2.39	0.758
10-0.35-0.5-comp	10	0.35	0.5	comp	0.918	0.963	0.753	0.30	0.950	0.550	2.41	0.758
3-0.35-0.0-comp	3	0.35	0.0	comp	0.950	0.950	0.735	0.35	0.900	0.600	2.37	0.758
3-0.5-0.5-comp	3	0.50	0.5	comp	0.950	0.950	0.732	0.35	0.900	0.550	2.20	0.751
3-0.35-0.5-comp	3	0.35	0.5	comp	0.950	0.950	0.740	0.30	0.900	0.550	1.80	0.748
3-0.7-0.1-comp	3	0.70	0.1	comp	0.350	0.350	0.247	0.15	0.320	0.150	0.72	0.263
5-0.7-0.0-comp	5	0.70	0.0	comp	0.350	0.350	0.239	0.15	0.290	0.200	0.72	0.263

k: top_k, t: similarity_threshold, T: temperature, EM: Exact Match, Correct: LlamaIndex Correctness, Faithful: LlamaIndex Faithfulness

Table D.3: Enhanced Summary Table of Top 20 Configurations, Sorted by Composite Score. The selected optimal configuration is highlighted.

Appendix E

GITHUB

https://github.com/JordanEmil/SpaceMissionDesignAssistant

E.1 README FILE - Guide to the codebase

Features

- **Interactive Chatbot Interface**: Streamlit-based web interface for easy interaction.
- **Comprehensive Knowledge Base**: Data from hundreds of space missions.
- **RAG Architecture**: Combines vector search with LLM generation for accurate, contextual responses.
- **Source Attribution**: Answers include references to source missions.
- **Optimised Performance**: Fine-tuned retrieval parameters for best results.

Quick Start

Prerequisites

- Python 3.8 or higher
- OpenAI API key

Installation

1. Clone the repository:

```
1 git clone <repository-url>
```

```
cd SpaceMissionDesignAssistant
```

2. Create a virtual environment (recommended):

```
python -m venv venv
# On macOS/Linux:
source venv/bin/activate
# On Windows (PowerShell):
venv\Scripts\Activate.ps1
```

3. Install dependencies:

```
pip install -r requirements.txt
```

Running the Streamlit Chatbot

1. Ensure the ChromaDB index is ready (the `chroma_db` directory exists).
2. Launch the Streamlit application:

```
streamlit run streamlit_chatbot.py
```

3. Open your browser and navigate to http://localhost:8501.
4. Enter your OpenAI API key when prompted.
5. Start asking questions about space missions.

Example Questions

- "What orbit regimes have been used for SAR imaging satellites?"
- "What are typical power requirements for Earth observation CubeSats?"
- "Which missions have used optical imaging payloads?"
- "What are common failure modes in small satellite missions?"
- "Compare antenna designs used in different SAR missions."

Project Structure

```
SpaceMissionDesignAssistant/
```

```
|-- streamlit_chatbot.py      # Main Streamlit application
|-- query_pipeline.py         # Core RAG query engine
|-- evaluation_framework.py   # Evaluation metrics and framework
|-- chroma_db/                # Vector database storage
|-- chat_logs/                # Conversation history logs
|-- data/                     # Raw mission data
|-- rag_ready_data/           # Processed documents for indexing
|-- Indexing/                 # Indexing scripts
|   |-- indexing_pipeline.py  # Main indexing script
|   `-- prepare_rag_data.py   # Data preparation utilities
|-- Scraping/                 # Web scraping utilities
`-- evaluation/               # Evaluation suite and results
```

Configuration

Optimised parameters determined through evaluation:

- **Top-K Retrieval**: 5 documents
- **Similarity Threshold**: 0.5
- **Temperature**: 0.1
- **LLM Model**: OpenAI GPT (configurable)

Building the Knowledge Base

If you need to rebuild the index:

1. Prepare the documents:

```
python Indexing/prepare_rag_data.py
```

2. Build the ChromaDB index:

```
python Indexing/indexing_pipeline.py
```

Advanced Usage

Evaluation Framework

```
cd evaluation
python run_evaluation_pipeline.py
```

Custom Queries (Programmatic)

```python
from query_pipeline import SpaceMissionQueryEngine

# Initialise engine
engine = SpaceMissionQueryEngine(
    chroma_persist_dir="./chroma_db",
    top_k=5,
    similarity_threshold=0.5,
    temperature=0.1
)

# Query
result = engine.query("What are the main components of a SAR satellite?")
print(result['response'])
```

Requirements

Main dependencies:

- `streamlit`
- `llama-index`
- `chromadb`
- `openai`
- `pandas`
- `numpy`

See `requirements.txt` for the full list.

Data Sources

The knowledge base is built from publicly available space mission data, including mission specifications, technical details, and historical information from various space agencies and organisations.

License

This project is provided as-is for educational and research purposes.

Author

Emil Ares

Troubleshooting

Common Issues

1. **"ChromaDB directory not found"**: Run the indexing pipeline first to create the database.
2. **"Invalid API key"**: Ensure your OpenAI API key is valid and has sufficient credits.
3. **"Module not found"**: Install all requirements using `pip install -r requirements.txt`.

Performance Tips

- The first query may take longer as the system loads the index.
- For faster responses, consider using a local LLM or adjusting the `top_k` parameter.
- Chat history is automatically saved in the `chat_logs` directory.

References

[1] "Satellite missions catalogue - eoportal." (), URL: https://directory.eoportal.org/satellite-missions?Mission+type=EO (Accessed: 30/4/2025) (cit. on pp. 1, 3, 29, 30).

[2] "Artificial intelligence tool to improve efficiency of mission design - earth online." (), URL: https://earth.esa.int/eogateway/success-story/artificial-intelligence-tool-to-improve-efficiency-of-mission-design (Accessed: 30/4/2025) (cit. on p. 1).

[3] "What is retrieval-augmented generation aka rag | nvidia blogs." (), URL: https://blogs.nvidia.com/blog/what-is-retrieval-augmented-generation/ (Accessed: 30/4/2025) (cit. on pp. 1, 2).

[4] P. Lewis, E. Perez, A. Piktus, *et al.*, "Retrieval-augmented generation for knowledge-intensive nlp tasks," *Advances in Neural Information Processing Systems*, vol. 2020-December, May 2020, ISSN: 10495258. URL: https://arxiv.org/pdf/2005.11401 (cit. on pp. 2, 9, 10, 16, 17, 23).

[5] E. Maleki, A. G. Fernandez, N. Fischer, Q. Wijnands, and N. Christofi, "Semantic-based systems engineering for digitalization of space mission design," *Frontiers in Industrial Engineering*, vol. 2, p. 1 426 074, Aug. 2024, ISSN: 2813-6047. DOI: 10.3389/FIENG.2024.1426074 (cit. on pp. 6, 7).

[6] "Esa agenda 2025," The European Space Agency, Tech. Rep., 2021 (cit. on p. 6).

[7] J. W. Evans, S. L. Cornford, D. Kotsifakis, N. /. Goddard, and M. S. Feather, *Enabling assurance in the mbse environment*, Jan. 2019 (cit. on pp. 6, 8).

[8] S. J. Hatakeyama, D. W. Seal, D. Farr, and S. C. Haase, "An alternate view of the systems engineering"v" in a model-based engineering environment," in *2018 AIAA SPACE and Astronautics Forum and Exposition*, American Institute of Aeronautics and Astronautics Inc, AIAA, 2018, ISBN: 9781624105753. DOI: 10.2514/6.2018-5326 (cit. on p. 7).

[9] "Orion sysml model, digital twin, and lessons learned for artemis i," *INCOSE International Symposium*, vol. 33, pp. 290–304, 1 Jul. 2023, ISSN: 2334-5837. DOI: 10.1002/IIS2.13022. URL: /doi/pdf/10.1002/iis2.13022%20https://onlinelibrary.wiley.com/doi/abs/10.1002/iis2.13022%20https://incose.onlinelibrary.wiley.com/doi/10.1002/iis2.13022 (cit. on p. 7).

[10] "Openmbee - open model based engineering environment." (), URL: https://www.openmbee.org/ (Accessed: 30/4/2025) (cit. on p. 7).

[11] L. Timperley, L. Berthoud, C. Snider, and T. Tryfonas, "Assessment of large language models for use in generative design of model based spacecraft system architectures," 2024. DOI: 10.2139/SSRN.4823264. URL: https://papers.ssrn.com/abstract=4823264 (cit. on p. 8).

[12] V. Rodriguez-Fernandez, A. Carrasco, J. Cheng, E. Scharf, P. M. Siew, and R. Linares, "Language models are spacecraft operators," Mar. 2024. URL: https://arxiv.org/pdf/2404.00413v1 (cit. on p. 8).

[13] "Nasa@sc24: Nasa-gpt: Searching the entire nasa technical reports server using ai." (), URL: https://www.nas.nasa.gov/SC24/research/project09.php (Accessed: 30/4/2025) (cit. on p. 8).

[14] A. T. Ray, "Standardization of engineering requirements using large language models a dissertation presented to the academic faculty," Tech. Rep., 2023 (cit. on p. 8).

[15] "Deploying a large language model in space." (), URL: https://www.boozallen.com/insights/ai-research/deploying-a-large-language-model-in-space.html (Accessed: 30/4/2025) (cit. on p. 9).

[16] Y. Gao, Y. Xiong, X. Gao, K. Jia, J. Pan, Y. Bi, Y. Dai, J. Sun, M. Wang, and H. Wang, *Retrieval-augmented generation for large language models: A survey*, 2024. URL: https://github.com/Tongji-KGLLM/ (cit. on pp. 9, 24, 27).

[17] "Rag vs. fine-tuning | ibm." (), URL: https://www.ibm.com/think/topics/rag-vs-fine-tuning (Accessed: 30/4/2025) (cit. on p. 9).

[18] "Seq2seq - wikipedia." (), URL: https://en.wikipedia.org/wiki/Seq2seq (Accessed: 30/4/2025) (cit. on p. 9).

[19] "Rag 101: Demystifying retrieval-augmented generation pipelines | nvidia technical blog." (), URL: https://developer.nvidia.com/blog/rag-101-demystifying-retrieval-augmented-generation-pipelines (Accessed: 30/4/2025) (cit. on p. 9).

[20] "Vector similarity explained | pinecone." (), URL: https://www.pinecone.io/learn/vector-similarity/ (Accessed: 30/4/2025) (cit. on p. 10).

[21] "Rag evaluation metrics: Assessing answer relevancy, faithfulness, contextual relevancy, and more - confident ai." (), URL: https://www.confident-ai.com/blog/rag-evaluation-metrics-answer-relevancy-faithfulness-and-more (Accessed: 30/4/2025) (cit. on p. 10).

[22] "Rag vs. fine-tuning | ibm." (), URL: https://www.ibm.com/think/topics/rag-vs-fine-tuning (Accessed: 30/4/2025) (cit. on pp. 10, 16, 26).

[23] "Retrieval augmented generation (rag) for llms | prompt engineering guide." (), URL: https://www.promptingguide.ai/research/rag (Accessed: 30/4/2025) (cit. on pp. 10, 20, 26, 27).

[24] "Evaluating rag part i: How to evaluate document retrieval | deepset blog." (), URL: https://www.deepset.ai/blog/rag-evaluation-retrieval (Accessed: 30/4/2025) (cit. on pp. 11, 14, 17).

[25] Z. Hasan. "Keyword search [mooc lecture]." (2025), URL: https://www.coursera.org/learn/retrieval-augmented-generation-rag/lecture/MYS0a/keyword-search-tf-idf (Accessed: 30/4/2025) (cit. on p. 11).

[26] "Understanding tf-idf (term frequency-inverse document frequency) - geeksforgeeks." (), URL: https://www.geeksforgeeks.org/machine-learning/understanding-tf-idf-term-frequency-inverse-document-frequency/ (Accessed: 1/5/2025) (cit. on p. 11).

[27] "Okapi bm25 - wikipedia." (), URL: https://en.wikipedia.org/wiki/Okapi_BM25 (Accessed: 30/4/2025) (cit. on p. 12).

[28] "What is bm25 (best matching 25) algorithm? - geeksforgeeks." (), URL: https://www.geeksforgeeks.org/nlp/what-is-bm25-best-matching-25-algorithm/ (Accessed: 1/5/2025) (cit. on p. 12).

[29] "Similarity metrics for vector search - zilliz blog." (), URL: https://zilliz.com/blog/similarity-metrics-for-vector-search (Accessed: 1/5/2025) (cit. on p. 13).

[30] "Vector similarity explained | pinecone." (), URL: https://www.pinecone.io/learn/vector-similarity/ (Accessed: 1/5/2025) (cit. on p. 13).

[31] "Rag retrieval performance enhancement practices: Detailed explanation of hybrid retrieval and self-query techniques - dev community." (), URL: https://dev.to/jamesli/rag-retrieval-performance-enhancement-practices-detailed-explanation-of-hybrid-retrieval-and-self-query-techniques-59ja (Accessed: 1/5/2025) (cit. on pp. 13, 14).

[32] "Evaluating retrieval | coursera." (), URL: https://www.coursera.org/learn/retrieval-augmented-generation-rag/lecture/RKASW/evaluating-retrieval (Accessed: 1/5/2025) (cit. on pp. 14, 15).

[33] "Module 3 introduction | coursera." (), URL: https://www.coursera.org/learn/retrieval-augmented-generation-rag/lecture/r24ke/module-3-introduction (Accessed: 1/5/2025) (cit. on p. 15).

[34] "Chunking | coursera." (), URL: https://www.coursera.org/learn/retrieval-augmented-generation-rag/lecture/AvkDR/chunking (Accessed: 1/5/2025) (cit. on p. 16).

[35] "Query parsing | coursera." (), URL: https://www.coursera.org/learn/retrieval-augmented-generation-rag/lecture/JsPOs/query-parsing (Accessed: 2/5/2025) (cit. on pp. 16, 17).

[36] "Cross-encoders and colbert | coursera." (), URL: https://www.coursera.org/learn/retrieval-augmented-generation-rag/lecture/8BPMB/cross-encoders-and-colbert (Accessed: 2/5/2025) (cit. on p. 17).

[37] "Mastering rag: How to select a reranking model." (), URL: https://galileo.ai/blog/mastering-rag-how-to-select-a-reranking-model (Accessed: 2/5/2025) (cit. on p. 17).

[38] "Colbert and beyond: Advancing retrieval techniques | by marc puig | medium." (), URL: https://medium.com/@mpuig/colbert-and-beyond-advancing-retrieval-techniques-81df1b2324d6 (Accessed: 2/5/2025) (cit. on p. 17).

[39] A. Vaswani, G. Brain, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, Ł. Kaiser, and I. Polosukhin, "Attention is all you need," (cit. on p. 18).

[40] "Foundation models, transformers, bert and gpt | niklas heidloff." (), URL: https://heidloff.net/article/foundation-models-transformers-bert-and-gpt/ (Accessed: 2/5/2025) (cit. on p. 18).

[41] A. Holtzman, J. Buys, L. Du, M. Forbes, Y. Choi, and P. G. Allen, *The curious case of neural text degeneration*. URL: https://github.com/ari-holtzman/degen (cit. on pp. 19, 20).

[42] "Top-p sampling - wikipedia." (), URL: https://en.wikipedia.org/wiki/Top-p_sampling (Accessed: 2/5/2025) (cit. on p. 19).

[43] T. Vergho, J.-F. Godbout, R. Rabbany, and K. Pelrine, "Comparing gpt-4 and open-source language models in misinformation mitigation," Jan. 2024. URL: https://arxiv.org/pdf/2401.06920v1 (cit. on p. 20).

[44] "Why blindly applying prompt engineering techniques to rag systems as a user is a bad idea | by aaron tay | medium." (), URL: https://aarontay.medium.com/why-blindly-applying-prompt-engineering-techniques-to-rag-systems-a-bad-idea-9ca7f893f6f3 (Accessed: 2/5/2025) (cit. on p. 21).

[45] "Prompt engineering: Building your augmented prompt | coursera." (), URL: https://www.coursera.org/learn/retrieval-augmented-generation-rag/lecture/L5CxJ/prompt-engineering-building-your-augmented-prompt (Accessed: 2/5/2025) (cit. on p. 21).

[46] "What is in-context learning, and how does it work: The beginner's guide | lakera – protecting ai teams that disrupt the world." (), URL: https://www.lakera.ai/blog/what-is-in-context-learning (Accessed: 2/5/2025) (cit. on pp. 22, 23).

[47] J. Wei, X. Wang, D. Schuurmans, M. Bosma, B. Ichter, F. Xia, E. H. Chi, Q. V. Le, and D. Zhou, "Chain-of-thought prompting elicits reasoning in large language models," *Advances in Neural Information Processing Systems*, vol. 35, Jan. 2022, ISSN: 10495258. URL: https://arxiv.org/pdf/2201.11903 (cit. on p. 22).

[48] "Prompt engineering: Advanced techniques | coursera." (), URL: https://www.coursera.org/learn/retrieval-augmented-generation-rag/lecture/Unbks/prompt-engineering-advanced-techniques (Accessed: 3/6/2025) (cit. on p. 23).

[49] Z. Ji, N. Lee, R. Frieske, *et al.*, "Survey of hallucination in natural language generation," *ACM Computing Surveys*, vol. 55, 12 Jul. 2024. DOI: 10.1145/3571730. URL: http://arxiv.org/abs/2202.03629%20http://dx.doi.org/10.1145/3571730 (cit. on p. 23).

[50] "Handling hallucinations | coursera." (), URL: https://www.coursera.org/learn/retrieval-augmented-generation-rag/lecture/NZlU7/handling-hallucinations (Accessed: 3/6/2025) (cit. on p. 23).

[51] "An overview on rag evaluation | weaviate." (), URL: https://weaviate.io/blog/rag-evaluation (Accessed: 3/6/2025) (cit. on p. 24).

[52] "Evaluating - llamaindex." (), URL: https://docs.llamaindex.ai/en/latest/module_guides/evaluating/ (Accessed: 3/6/2025) (cit. on p. 24).

[53] X. Xu, H. Weytjens, D. Zhang, Q. Lu, I. Weber, and L. Zhu, "Ragops: Operating and managing retrieval-augmented generation pipelines," *Arxiv*, Jun. 2025. URL: https://arxiv.org/pdf/2506.03401 (cit. on pp. 24–26).

[54] L. Ammann, S. Ott, C. R. Landolt, and M. P. Lehmann, "Securing rag: A risk assessment and mitigation framework," May 2025. URL: https://arxiv.org/pdf/2505.08728v1 (cit. on p. 25).

[55] "Which is better, retrieval augmentation (rag) or fine-tuning? both. | snorkel ai." (), URL: https://snorkel.ai/blog/which-is-better-retrieval-augmentation-rag-or-fine-tuning-both/ (Accessed: 30/4/2025) (cit. on p. 26).

[56] "Llamaindex - build knowledge assistants over your enterprise data." (), URL: https://www.llamaindex.ai/ (Accessed: 20/7/2025) (cit. on pp. 29, 33).

[57] "Vectorstoreindex vs chroma integration for llamaindex's vector embeddings - bitpeak." (), URL: https://bitpeak.com/vectorstoreindex-vs-chroma-integration-for-llamaindexs-vector-embeddings/ (Accessed: 20/7/2025) (cit. on p. 29).

[58] "Openai." (), URL: https://openai.com/ (Accessed: 20/7/2025) (cit. on p. 29).

[59] "Chroma." (), URL: https://www.trychroma.com/ (Accessed: 20/7/2025) (cit. on pp. 29, 33).

[60] "Streamlit • a faster way to build and share data apps." (), URL: https://streamlit.io/ (Accessed: 11/8/2025) (cit. on p. 35).

[61] "Ragas/docs/howtos/integrations/llamaindex.ipynb at main · explodinggradients/ragas." (), URL: https://github.com/explodinggradients/ragas/blob/main/docs/howtos/integrations/llamaindex.ipynb (Accessed: 11/8/2025) (cit. on p. 40).