FACULDADE DE ENGENHARIA DA UNIVERSIDADE DO PORTO

Retrieval-Augmented Generation for Industrial Applications: Design and Evaluation of an AI Assistant

Francisco Pires da Silva Galhardo

Mestrado em Engenharia Mecânica

Supervisor: Armando Jorge Miranda de Sousa, Professor Associado, FEUP

Co-supervisor: Carlos Daniel Santos Souto, Investigador, FEUP

July 30, 2026

Retrieval-Augmented Generation for Industrial Applications: Design and Evaluation of an AI Assistant

Francisco Pires da Silva Galhardo

Mestrado em Engenharia Mecânica

July 30, 2026

Abstract

Industrial machinery manufacturers distribute technical knowledge through dense manuals that operators and engineers must navigate manually to answer operational questions. This process is time-consuming, prone to error, and inaccessible to users without prior familiarity with the documentation. Large language models offer a promising alternative, but their reliance on static parametric knowledge makes them unsuitable for domain-specific technical queries where safety, accuracy and grounding in authoritative sources are critical.

This dissertation presents AMOB-RAG, a modular Retrieval-Augmented Generation system designed to answer natural language questions in Portuguese about technical documentation for rotary draw bending machines. The system is structured around two pipelines: an offline ingestion pipeline that preprocesses, chunks, and indexes markdown documentation into a persistent vector store, and an online query pipeline that retrieves, reranks, and assembles context before generating grounded answers. Key design decisions include semantic chunking with configurable breakpoint detection, CrossEncoder reranking with a table penalty heuristic, domain-specific query expansion from a structured axis synonym file, and a similarity threshold gate that prevents low-confidence retrievals from reaching the generation stage.

The system is evaluated against a human-verified golden dataset of 20 question-answer pairs in Portuguese, covering five question categories. Evaluation combines automated metrics with an LLM-as-a-judge framework that scores each answer across three qualitative dimensions: faithfulness, answer relevance, and context relevance. A controlled experiment strategy isolates the contribution of each pipeline component. A baseline experiment demonstrates that a standalone large language model fails to answer domain-specific questions reliably, establishing the necessity of the RAG architecture. Subsequent controlled experiments validate the individual contributions of each ingestion and query pipeline component.

The results, obtained from this initial case study, confirm that the modular architecture achieves reliable retrieval and grounded answer generation over specialized industrial documentation, with the evaluation framework providing sufficient diagnostic resolution to identify failure modes in specific pipeline components. As the system has not yet been deployed within AMOB's production environment, these findings establish a validated technical foundation ahead of a future field deployment and evaluation with AMOB's operators.

Keywords: Retrieval-Augmented Generation, Large Language Models, CrossEncoder reranking, semantic chunking, Industry 5.0, Digitalization.

Resumo

Os fabricantes de maquinaria industrial distribuem conhecimento técnico através de manuais extensos que operadores e engenheiros consultam manualmente para responder a questões operacionais. Este processo é moroso, sujeito a erros e inacessível a utilizadores sem familiaridade prévia com a documentação. Os modelos de linguagem de grande escala constituem uma alternativa promissora, mas a sua dependência de conhecimento paramétrico estático torna-os inadequados para consultas técnicas específicas de domínio, onde a segurança, precisão e o ancoramento em fontes autorizadas são críticos.

Esta dissertação apresenta o AMOB-RAG, um sistema modular de Geração Aumentada por Recuperação concebido para responder a perguntas em linguagem natural, em português, sobre documentação técnica de máquinas de dobragem por encurvamento rotativo. O sistema é estruturado em dois pipelines: um pipeline de ingestão offline que pré-processa, divide e indexa documentação markdown numa base vetorial persistente, e um pipeline de consulta online que recupera, reordena e constrói contexto antes de gerar respostas fundamentadas. As principais decisões de desenho incluem divisão semântica com deteção de pontos de quebra configurável, reordenação com CrossEncoder e penalização de tabelas, expansão de consultas a partir de um ficheiro de sinónimos de eixos, e um limiar de similaridade que impede recuperações de baixa confiança de chegarem à fase de geração.

O sistema é avaliado com base num conjunto de dados dourado verificado por humanos, composto por 20 pares pergunta-resposta em português, abrangendo cinco categorias de questões. A avaliação combina métricas automáticas com um framework de avaliação por modelo de linguagem que pontua cada resposta em três dimensões: fidelidade, relevância da resposta e relevância do contexto. Uma estratégia de experimentação controlada isola a contribuição de cada componente do pipeline. Uma experiência de linha de base demonstra que um modelo de linguagem autónomo falha em responder de forma fiável a questões específicas do domínio, estabelecendo a necessidade da arquitetura RAG. Experiências controladas subsequentes validam as contribuições individuais de cada componente de ingestão e de consulta.

Os resultados, obtidos a partir deste estudo de caso inicial, confirmam que a arquitetura modular alcança recuperação fiável e geração de respostas fundamentadas sobre documentação industrial especializada, com o framework de avaliação a fornecer resolução diagnóstica suficiente para identificar modos de falha em componentes específicos do pipeline. Uma vez que o sistema ainda não foi implementado no ambiente de produção da AMOB, estes resultados estabelecem uma base técnica validada, previamente a uma futura implementação e avaliação em contexto real com os operadores da AMOB.

Palavras-chave: Geração Aumentada por Recuperação, Modelos de Linguagem de Grande Escala, reranking com CrossEncoder, chunking semântico, Indústria 5.0, Digitalização.

UN Sustainable Development Goals

The United Nations Sustainable Development Goals (SDGs) provide a global framework to achieve a better and more sustainable future for all. It includes 17 goals to address the world's most pressing challenges, including poverty, inequality, climate change, environmental degradation, peace, and justice.

This dissertation presents AMOB-RAG, a Retrieval-Augmented Generation system that enables operators and engineers to query technical documentation for industrial bending machines using natural language. By reducing the time and expertise required to access specialized knowledge, the system contributes to more productive, inclusive, and innovative industrial environments. The contribution of this work can be aligned with the following Sustainable Development Goals:

SDG 8 — Decent Work and Economic Growth — Target 8.2 Promote sustained, inclusive and sustainable economic growth, full and productive employment and decent work for all. Target 8.2: Achieve higher levels of economic productivity through diversification, technological upgrading and innovation, including through a focus on high-value added and labour-intensive sectors.

SDG 9 — Industry, Innovation and Infrastructure — Target 9.5 Build resilient infrastructure, promote inclusive and sustainable industrialization and foster innovation. Target 9.5: Enhance scientific research, upgrade the technological capabilities of industrial sectors in all countries, including by 2030 encouraging innovation and substantially increasing the number of research and development workers per 1 million people and public and private research and development spending.

SDG 12 — Responsible Consumption and Production — Target 12.6 Ensure sustainable consumption and production patterns. Target 12.6: Encourage companies, especially large and transnational companies, to adopt sustainable practices and to integrate sustainability information into their reporting cycle.

SDG 17 — Partnerships for the Goals — Target 17.6 Strengthen the means of implementation and revitalize the global partnership for sustainable development. Target 17.6: Enhance North-South, South-South and triangular regional and international cooperation on and access to science, technology and innovation and enhance knowledge sharing on mutually agreed terms, including through improved coordination among existing mechanisms, in particular at the United Nations level, and through a global technology facilitation mechanism.

SDG	Target	Contribution	Performance Indicators and Metrics
8	8.2	By providing fast, accurate access to technical documentation at the point of need, the system reduces downtime and supports higher productivity in industrial machine operation and maintenance workflows.	Reduction in machine downtime attributed to documentation lookup; increase in operator task completion rate.
9	9.5	The development of a modular, configurable RAG architecture with a formal evaluation framework contributes to the scientific research base for AI-assisted industrial systems and demonstrates a replicable methodology for deploying language models in specialized technical domains.	Number of controlled experiments conducted; reusability of the modular architecture across document sets; reproducibility of the evaluation pipeline.
12	12.6	Improved access to machine documentation supports correct operation and maintenance practices, reducing errors that lead to material waste, premature component replacement, and inefficient use of resources during industrial manufacturing processes.	Reduction in operator errors attributable to documentation misuse; rate of correct procedure adherence as measured by system query success.
17	17.6	The development of AMOB-RAG as a collaboration between FEUP and AMOB exemplifies a university-industry partnership that transfers AI research into an industrial setting, demonstrating a replicable model for knowledge sharing between academic and private sector organizations.	Reusability of the system architecture across industrial partners; number of documented knowledge-transfer outcomes between FEUP and AMOB.

Acknowledgements

I would like to thank my supervisor, Professor Armando Jorge Miranda de Sousa, and my co-supervisor, Carlos Souto, for their guidance and feedback throughout this dissertation, every week. I would also like to thank Ana Luísa for her help in guiding me through the whole thesis writing process.

I would like to thank Carlos Agra, AMOB's technician responsible for the project's feedback, and João Sousa, who gave this project a good head start by laying the groundwork that made the development of the RAG system possible.

A special thanks to my university friends, for showing me that university could actually be fun, and for giving me something I can cherish for the rest of my life.

An even more special thanks to my mom, my dad, my brother, and my grandparents. They made me believe in myself, always giving me the strength I needed through these years. They taught me the great value of hard work, determination, and self-belief to achieve my goals.

This work is a result of the project "Smartubending - Smart Tube Bending Machines", with no. 18440 and operation code at the Funds Platform COMPETE2030-FEDER-01480000, co-financed by COMPETE 2030, by Portugal 2030, and by the European Union.

Francisco Galhardo

Contents

List of Figures

List of Tables

List of Listings

List of Acronyms

AI	Artificial Intelligence
API	Application Programming Interface
BM25	Best Match 25
CLI	Command-Line Interface
CNC	Computer Numerical Control
CPU	Central Processing Unit
GPT	Generative Pre-trained Transformer
GPU	Graphics Processing Unit
HTML	HyperText Markup Language
IoT	Internet of Things
JSON	JavaScript Object Notation
LLM	Large Language Model
NLP	Natural Language Processing
PDF	Portable Document Format
Q&A	Question and Answer
RAG	Retrieval Augmented Generation
REGEX	Regular Expression
RSLP	Removedor de Sufixos da Língua Portuguesa
TF-IDF	Term Frequency-Inverse Document Frequency
UI	User Interface

Chapter 1

Introduction

The present chapter provides a comprehensive introduction to the context, motivation, and objectives of this dissertation. It starts by outlining the current challenges in industrial knowledge management, with a particular focus on the limitations of manual documentation lookup in technical manufacturing environments and the growing potential of large language models for domain-specific question answering. As part of addressing these challenges, this work focuses on two core technological foundations: Retrieval-Augmented Generation (RAG), which grounds language model responses in authoritative source documents at query time, and a modular service-layer architecture, which enables systematic experimentation and evaluation of each pipeline component in isolation. The motivation for this work is then presented, highlighting the gap between the capabilities of general-purpose language models and the precision required for industrial technical queries. The chapter concludes with a presentation of the dissertation's primary objectives and an overview of its structure.

1.1 Context

Industrial environments are known for their complexity, automation in all sectors and necessity of correlation between all of them within the company and at factory level. At this level, a large number of machines coexist, all of them with documentation for the correspondent guidance, trouble shooting and to serve as general instructions and specifications manual for the operator.

AMOB is a world-renowned company known for developing, designing, commercializing and serving as a technical assistant for a large variety of products, all in the field of equipment and tools for tube and sheet metal forming, from simple to complex CNC machines, totally electric [1]. For this dissertation the rotary draw bending machines are the main focus. The machine itself is a device used to bend tubes with high precision by rotating them around a fixed radius die, to shape them into a perfectly angled form. Its numerous technical documents are the source of data used throughout the development of this project.

This dissertation was developed within the Smartubending research project, funded by Portugal 2030, Compete 2030, and the European Union, in which FEUP/DEMec and AMOB collaborated directly. The partnership positions this work at the intersection of academic research and industrial application: the system developed is not a proof-of-concept built on synthetic data, but a functional prototype grounded in real machine documentation from a company whose products are deployed in manufacturing facilities worldwide. For a national company of AMOB's scale, the ability to make dense technical knowledge instantly accessible to operators and engineers through natural language interaction represents a direct operational gain, reducing the time and expertise required to navigate documentation that currently demands manual search through hundreds of pages. The research contribution of this dissertation is therefore both scientific, in its design and evaluation of a modular RAG architecture for industrial documentation, and practical, in its potential to be adopted and extended within AMOB's product and support ecosystem.

Usually, these technical documents are very dense, complex and can also be very difficult to interpret. They contain large volumes of structured and domain-specific information. A common issue at factory level is the time lost searching for information between the hundreds or thousands of pages in the documentation. This challenge leads to lost productivity, augmented costs for the company and increases the likelihood of operational errors. Beyond the factory floor, the burden also extends to the AMOB technical support team, who frequently field operator queries that could be resolved by consulting the documentation, diverting engineering time away from higher-value tasks. A system capable of answering such queries autonomously would therefore reduce the support workload at the manufacturer level as well.

The growth of Artificial Intelligence (AI) in recent years has accelerated across many domains. Among the most promising developments within this field is RAG, a technique that enables Large Language Models (LLMs) to retrieve relevant information from a source document, allowing them to answer questions about that source with greater accuracy. The recent emergence of the RAG approach brings diverse capabilities and a range of functionalities that can be applied in industrial environments, such as assisting human operators in retrieving information from technical manuals.

Retrieving relevant content in this context requires not only identifying pertinent sections, but also ensuring that the retrieved information is complete and contextually coherent. Incomplete or fragmented retrieval can lead to incorrect or misleading answers, particularly in procedural tasks. Looking for solutions to this problem is a growing topic of interest, with attention shifting towards the retrieval capabilities of the model. The usual challenges can be associated with the quality of the data or the quality of the pipeline the system follows. The key is to find a pipeline that is both effective and efficient for the associated data.

Furthermore, one of the challenges in this type of architecture are the LLMs themselves. Hallucination is a common issue, especially for technical documents, given the premise that the generated answers need to be factually accurate. The validation of the LLMs still needs to be done manually, to validate the information acquired.

Despite improvements of RAG systems, ensuring accurate and reliable responses without hallucinations remains a challenge, given the accuracy needed for technical documents. Incomplete

context, fragmented information and bad structuring are some of the common barriers to achieve efficiency in the pipelines. These challenges justify the need for a very robust and efficient approach for building of the pipeline.

1.2 Motivation

Automation and digitalization plays a fundamental role in modern industrial environments by improving profitability, reducing time lost and overall improvement in company productivity. All these improvements work towards a cost-efficient future for the industrial companies, underscoring the importance of developing, studying and advancing such technologies.

The European Union has been a strong driver of this transition, actively promoting Industry 5.0 as the next step in industrial evolution. Unlike its predecessor, Industry 5.0 shifts the focus toward collaboration between humans and intelligent systems, rather than pursuing full automation as an end goal. This framework explicitly encourages the integration of AI-powered tools that support and augment workers in their daily tasks, making the development of AI assistants for industrial environments directly aligned with this broader policy direction.

Although there has been significant progress in AI capabilities, manual search through hundreds of pages of technical documentation remains the prevailing practice. Recent advances in RAG systems enable new approaches to information retrieval tasks that traditionally rely on manual search and expert navigation of technical documentation.

While conceptually straightforward, RAG presents common issues, depending on the quality of the data and on the quality of the pipeline. The chunking of the data into meaningful units is key for a consistent RAG system, as poor segmentation decisions can lead to loss of structure, small chunks that lack context and introduce noise, or chunks that are too large causing irrelevant retrieval.

Moreover, industrial technical documents tend to have numerous pieces of information about various topics in a very dense arrangement, which compounds the challenges of preprocessing the data. The data is typically hierarchical, procedural and filled with domain-specific terminology, all of which represent considerable challenges to be addressed.

Another critical aspect lies in the rigor needed when searching through technical documents. Any inaccuracy in the generated answer can cause equipment damage, safety incidents, information wrongly given to clients or costly operational errors. This increases the need for RAG applications to be far more robust, in this context. In this context, hallucinations go far beyond a minor inconvenience, constituting a serious risk to industrial safety and operational protocols. It is important to conduct thorough research in this matter, before any real-world implementation.

Despite growing interest in RAG applications, there is still a significant research gap between a robust, reliable RAG pipeline tailored for industrial data and the current state of the art. The goal of this dissertation is to contribute toward establishing a strong theoretical and practical foundation for building effective RAG pipelines applicable to the challenges outlined.

1.3 Research Objectives

Given the context and motivation behind this dissertation, it is important to define concrete objectives to have a structured and well-defined workflow. The main goal is to develop a functional system, capable of answering questions related to the eMOB machine technical documents with accuracy.

However, to achieve the main objective it is important to also define specific objectives. With this, the following objectives were defined:

- Collection and preprocessing of AMOB's eMOB technical documentation, ensuring the data is structured and suitable for ingestion into a RAG pipeline.
- Design and implementation of a RAG pipeline tailored to industrial technical documents, minimizing context loss and hallucination risks.
- Development of an interactive application exposing the RAG pipeline to end users, enabling natural language querying of the machine documentation.
- Validation of the complete system with real-world use cases and AMOB's professional workers, assessing response accuracy, relevance, and practical utility against manual documentation search.

1.4 Dissertation Structure

In addition to this introduction, this dissertation contains five more chapters.

Chapter 2 surveys the state of the art in large language models, retrieval-augmented generation, and evaluation frameworks for natural language generation systems, situating the present work within the existing literature and identifying the gaps it addresses.

Chapter 3 presents the system architecture and design of AMOB-RAG at a conceptual level, describing the two core pipelines, the ingestion pipeline that preprocesses and indexes documentation into a persistent vector store, and the query pipeline that retrieves, reranks, and assembles context before generating grounded answers, together with the evaluation architecture.

Chapter 4 covers the implementation of the system in detail, documenting the configuration mechanism, the ingestion and query pipeline stages in code, the evaluation framework, and the frontend interface.

Chapter 5 presents the evaluation and results, including a baseline experiment that establishes the necessity of retrieval, an end-to-end evaluation against a human-verified golden dataset, an LLM-as-a-judge assessment, a response analysis with concrete examples, a multi-model comparison, and an observability and trace inspection study.

Chapter 6 concludes the dissertation with a summary of contributions and directions for future work.

Chapter 2

State of the Art

2.1 Background

2.1.1 Industrial transition

The history of industrial production has been marked by successive waves of transformation, each redefining how goods are made and how work is organized. The emergence of digital technologies in recent decades accelerated this process, introducing cyber-physical systems, the Internet of Things (IoT), cloud computing, and artificial intelligence into manufacturing environments [2]. This convergence, broadly known as Industry 4.0, promises to realize lean and responsive production through the use of smart machines and data-driven decision making [2]. At the same time, operating these increasingly complex production lines has become more knowledge-intensive, putting growing strain on a factory's capacity to train and support operators [3]. However, excessive reliance on technological solutions and the underwhelming outcomes of some Industry 4.0 initiatives have led to a reassessment of the role of the human element in production, giving rise to the Industry 5.0 paradigm [4]. This shift, which also reflects the transition from Operator 4.0 to Operator 5.0, emphasizes research and innovation oriented toward a sustainable, human-centric, and resilient industry [4].

Despite the digitization push, the knowledge that operators need daily remains largely inaccessible in practice. Manufacturing environments accumulate vast amounts of technical information such as machine manuals, standard operating procedures, safety protocols, and issue reports, yet this knowledge is often stored across disconnected systems and formats, making it difficult to process and interpret when needed [3]. The situation is compounded by the unstructured nature of much of this content, particularly issue reports generated on the shop floor, which frequently suffer from inconsistent terminology, poor grammar, and incomplete descriptions [3]. This systemic breakdown is best illustrated by the fact that operators have been observed resorting to personal smartphones to find answers rather than consulting official documentation, a behavior that signals not individual failure but a fundamental inadequacy in how industrial knowledge is made available [3].

The human cost of this knowledge gap is measurable: operators working with complex tasks under inadequate documentation support experience significantly higher cognitive load, leading to process errors and reduced output quality [4]. Despite the broader Industry 4.0 push toward intelligent tooling, existing documentation systems in manufacturing remain largely keyword-based and dependent on expert navigation, leaving operators without a reliable way to query the knowledge contained within their own technical documentation [2]. This challenge is particularly pronounced in highly specialized manufacturing domains, where technical documentation is dense, process-specific, and requires expert knowledge to interpret correctly.

2.1.2 Rotary Draw Bending

The present dissertation is situated within one such domain, addressing the documentation of rotary draw bending machines developed by AMOB, a company specializing in equipment for tube and sheet metal forming [1]. Rotary draw bending is a precision metal forming technique widely adopted in aerospace, automotive, and general manufacturing for shaping metallic tubes with high dimensional accuracy [5, 6]. The process works by clamping the tube against a rotating bend die and drawing it around a fixed radius, while a set of tooling components, including a pressure die, wiper die, and mandrel, control the material flow and prevent defects from forming [7, 8]. Figure 2.1 illustrates the typical tooling configuration of a rotary draw bending machine, identifying each of the key components involved in the process.

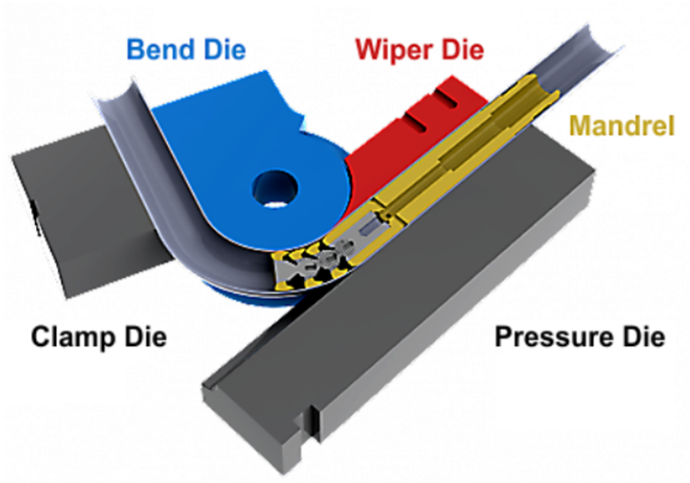

Figure 2.1: Setup representation of the rotary draw bending manufacturing process [9].

The technical complexity of the process arises from the interaction between several process parameters, including the centerline radius, tube diameter, wall thickness, and bending angle, all of which must be precisely calibrated to achieve the intended final geometry [5]. Figure 2.2 illustrates the key geometric parameters used to characterize the tube and the bend throughout the process.

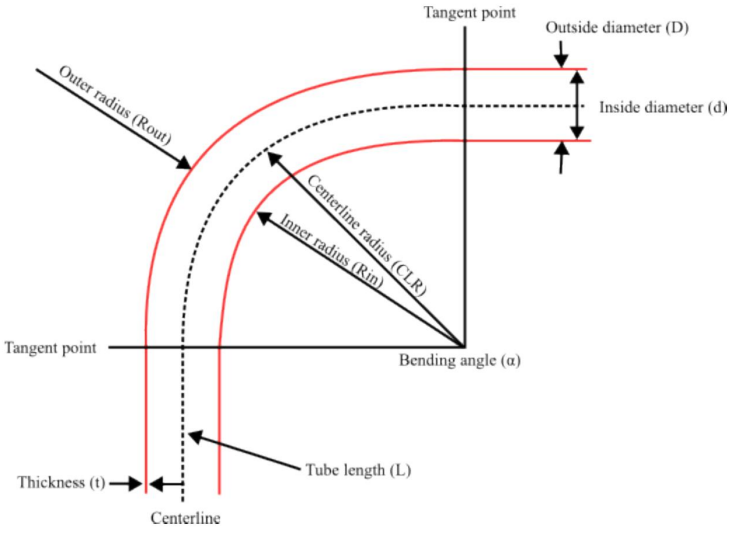

Figure 2.2: Schematic illustration of a bent tube with the applied nomenclature [10].

During bending, the material undergoes elastic-plastic deformation, and upon unloading, the elastic component is partially recovered, a phenomenon known as springback, which introduces geometric deviations that must be anticipated and compensated in the tooling design [5, 6]. Additional defects such as wall thinning, wrinkling, and cross-section distortion can arise from improper tool selection, inadequate setup, or incorrect process parameters, requiring operators to navigate dense, technically demanding documentation to diagnose and resolve these issues in practice [7, 8]. Figure 2.3 illustrates the four main defects that can arise during the rotary draw bending process.

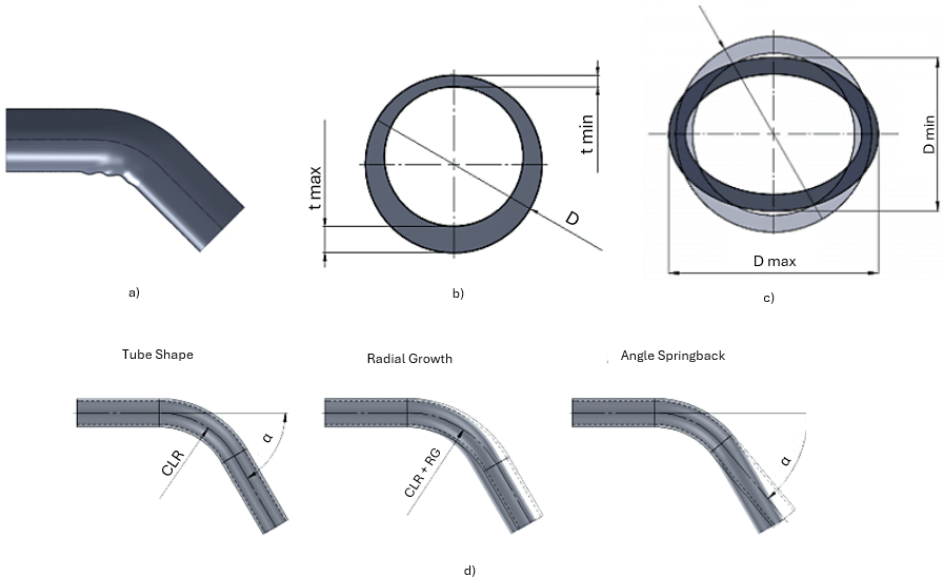

Figure 2.3: Main defects found in the rotary draw bending process: a) wrinkling; b) wall thinning and thickening; c) cross-section distortion; d) springback [7].

With advancements in digital manufacturing technologies, computer numerical control (CNC) has been integrated into rotary draw bending systems to improve accuracy, repeatability, and automation [5, 6]. These developments promoted the transition from traditional hydraulic actuators to faster and more precise electric motors, increasing competitiveness in terms of productivity and efficiency [5, 6]. This evolution has made it possible to achieve high-volume production without compromising quality standards, while also increasing the technical complexity of the machines and the documentation required to operate them [5].

To fully understand why defects arise, it is necessary to consider the mechanical behavior of the tube during bending. As illustrated in Figure 2.4, the tube cross-section contains a region surrounding the neutral axis where induced stresses remain below the material's yield strength, resulting in purely elastic deformation, while regions further from the neutral axis are subjected to stresses that exceed the yield limit, leading to plastic deformation [6, 7]. Tensile stresses develop on the extrados due to stretching, while compressive stresses occur on the intrados due to material compression, and it is this opposing stress distribution that drives the formation of the defects described above [6, 7].

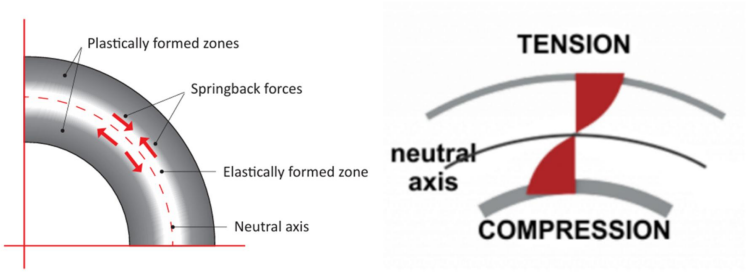

Figure 2.4: Visual representation of: a) zones of plasticity and elasticity with springback forces; b) distribution of stress along the diameter of the tube [7].

A direct consequence of this stress distribution is the existence of a narrow process window within which acceptable bends can be produced. Figure 2.5 illustrates how this window is bounded above by cracking on the outer surface and below by wrinkling on the inner surface, with the acceptable operating range narrowing considerably for the most demanding bending operations, characterized by small centerline radii and thin wall thicknesses [8]. Operating within this window requires precise calibration of process parameters and a thorough understanding of the machine's documentation, reinforcing the need for reliable and accurate information access in industrial environments where these machines are deployed.

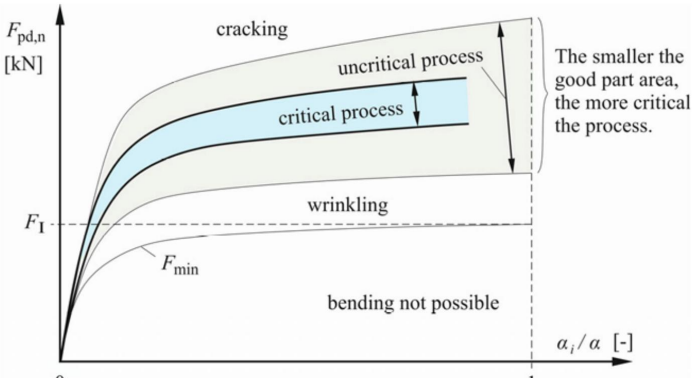

Figure 2.5: Process window of the pressure die's normal force over the normalised bending angle, adapted from [8].

2.2 Large Language Models (LLMs)

Large Language Models (LLMs) represent a fundamental shift in how machines process and generate natural language. Built on neural architectures capable of learning from vast amounts of text, these models have demonstrated remarkable ability to understand context, follow instructions, and produce coherent responses across a wide range of tasks. Their capacity to interpret and retrieve information from domain-specific documents makes them a particularly compelling tool for addressing the knowledge access challenges described in industrial environments.

Prior to the Transformer, sequence modeling tasks in natural language processing (NLP) relied predominantly on recurrent neural networks, which processed input sequentially and struggled to capture long-range dependencies efficiently. A new architecture built entirely on attention mechanisms was proposed, dispensing with recurrence and convolutions entirely, which allowed the model to process all positions in a sequence simultaneously rather than one at a time [11]. This parallelization not only drastically reduced training time but also enabled scaling to model sizes and dataset volumes that were previously impractical. At the core of this architecture is the self-attention mechanism, in which every position in a sequence can directly attend to every other through scaled dot-product attention, capturing global dependencies in a constant number of sequential operations [11]. These computations are organized into multiple parallel heads, known as multi-head attention, enabling the model to jointly attend to information from different representation subspaces simultaneously, laying the architectural foundation upon which modern large language models are built. Figure 2.6 illustrates the overall structure of the Transformer model, highlighting the encoder-decoder organization and the multi-head attention mechanism at its core [11].

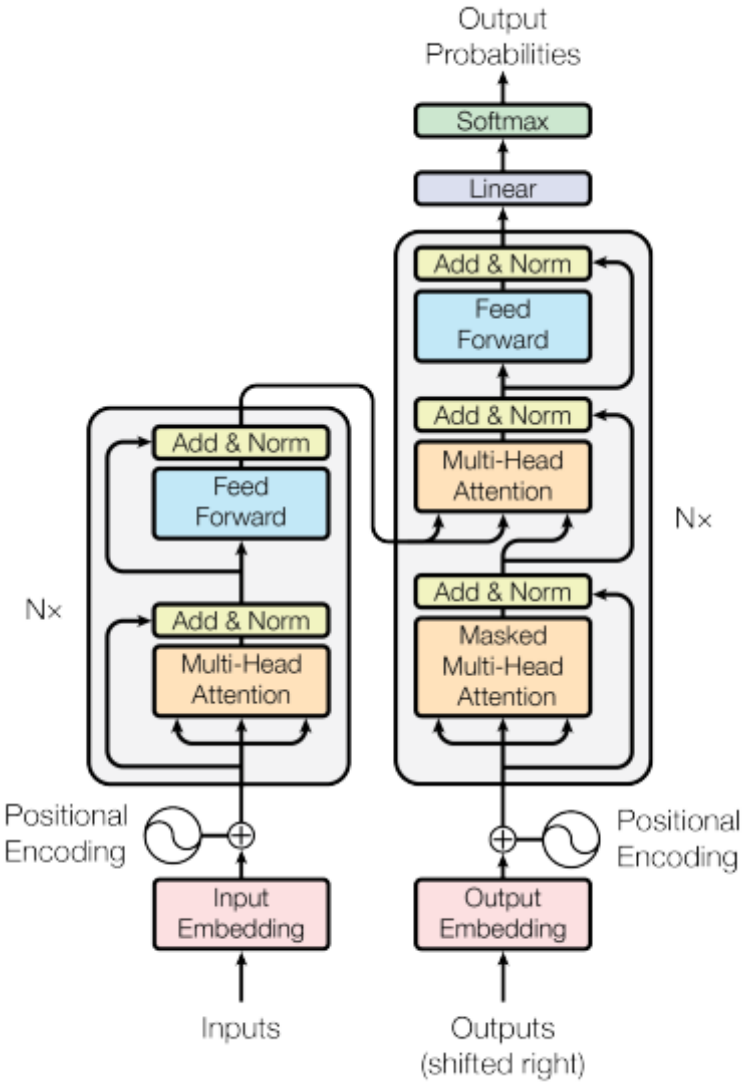

Figure 2.6: The Transformer model architecture [11].

The Transformer architecture enabled a new training paradigm in which models are first pre-trained on large-scale texts, acquiring broad linguistic and factual knowledge before being applied to any specific task. Brown et al. demonstrated that scaling this approach to 175 billion parameters produced a model, GPT-3, capable of performing a wide range of tasks from translation to question answering without any task-specific fine-tuning, relying solely on a few examples provided in the input prompt [12]. This work established that sufficiently large pre-trained models exhibit emergent capabilities, abilities that were never explicitly trained for but arise naturally from exposure to diverse text at scale [12]. These qualities, particularly the ability to interpret, summarize, and retrieve information from large text-based datasets, make LLMs a compelling tool for industrial documentation environments, where operators need to query complex, heterogeneous technical knowledge in natural language [3, 12].

While pre-training on large text corpora equips a model with broad linguistic and factual

knowledge, it does not on its own produce a model that reliably follows user instructions or generates helpful, well-structured responses. A further training stage known as instruction tuning addresses this gap by fine-tuning pre-trained models on datasets of human-written instructions paired with desired responses, teaching the model to align its outputs with human intent rather than simply continuing text in a statistically likely way [13]. This approach is often combined with reinforcement learning from human feedback, in which human evaluators rank model outputs and a reward model is trained to capture those preferences, guiding subsequent fine-tuning toward responses that humans judge to be more useful, honest, and safe [13]. Modern instruction-tuned models inherit the broad capabilities of pre-training while becoming substantially more reliable as conversational and task-oriented assistants, a property that is essential when deploying them in operational contexts such as industrial documentation querying.

The original demonstrations of large language model capabilities were carried out using proprietary models accessible only through commercial APIs, which limited research reproducibility and made deployment in environments with strict data confidentiality requirements impractical. Beyond access restrictions, running these models in-house requires substantial computational infrastructure, including high-end GPUs or NPUs, which places self-hosted deployment out of reach for many research groups and industrial organizations operating under tight hardware budgets. The release of open-weight foundation models marked a significant shift in this landscape, providing the research and industrial communities with models whose weights can be inspected, fine-tuned, and run locally without external dependencies [14]. The Llama family of models, in particular, demonstrated that competitive performance could be achieved at parameter counts substantially smaller than the largest proprietary models, while being trained exclusively on publicly available data [14]. This shift has direct implications for industrial deployments, where data privacy concerns, network reliability constraints, and the need for predictable inference costs often make local or self-hosted deployment preferable to reliance on third-party APIs.

Beyond architecture and training, the number of parameters a model contains is one of the primary determinants of its capability, with larger models generally exhibiting stronger reasoning and broader knowledge as a direct consequence of scale [12]. This relationship is not strictly linear, and the Llama family demonstrated that careful training can allow smaller models to rival much larger ones, making parameter count a tradeoff against inference cost, memory footprint, and latency rather than a simple proxy for quality [14]. A model with fewer parameters is faster and cheaper to run but tends to struggle with complex multi-step tasks, while a larger model offers stronger performance at the expense of higher computational demand, a tension that becomes central when selecting a model for deployment in a resource-constrained or latency-sensitive setting. A further consideration is the distinction between dense and Mixture-of-Experts (MoE) architectures, as dense models activate all of their parameters for every token, whereas MoE models route each token through only a small subset of specialized expert subnetworks, so that the total parameter count, which reflects the model's overall capacity, can be far larger than the active parameter count, which reflects the computation actually performed per token. This separation allows MoE models to offer the knowledge capacity of a large model while keeping inference closer to the cost

of a much smaller one, a design that is increasingly common among recent open-weight models.

Table 2.1 summarizes the architectural characteristics of a representative set of recent open-weight models available on the Groq inference platform, illustrating both the parameter-count spectrum and the contrast between dense and Mixture-of-Experts designs [14, 15].

Table 2.1: Architectural comparison of representative open-weight language models.

Model	**Architecture**	**Total Parameters**	**Active Parameters**	**Context Window**
Llama 3.1 8B	Dense	8B	8B	128K
Llama 3.3 70B	Dense	70B	70B	128K
GPT-OSS 20B	MoE	21B	3.6B	128K
GPT-OSS 120B	MoE	117B	5.1B	128K

A complementary thread of research has shown that the way a prompt is constructed can substantially influence the quality of an LLM's response, even without any change to the underlying model. Chain-of-thought prompting, in which the model is encouraged to produce intermediate reasoning steps before arriving at a final answer, has been shown to significantly improve performance on tasks requiring multi-step reasoning, particularly when the model is sufficiently large [16]. This and related prompting strategies have established prompt engineering as a first-class technique in the deployment of LLM-based systems, allowing system designers to shape model behaviour through carefully constructed system prompts and input templates rather than through additional training. In retrieval-augmented systems, this principle extends to how retrieved context is formatted and integrated into the prompt, with the structure of the prompt itself directly affecting the faithfulness and relevance of the generated response.

Despite their remarkable potential, large language models present two fundamental limitations that constrain their reliable deployment in knowledge-intensive applications. The first is hallucination, the tendency of LLMs to generate plausible but factually incorrect content, a phenomenon that raises significant concerns over the reliability of model outputs in real-world settings where accuracy is critical [17]. The second is the static knowledge cutoff, the fact that a model's knowledge is frozen at the time of training and cannot reflect updates, new procedures, or domain-specific information added after that point [3]. Together, these limitations mean that a standalone LLM, however capable, is not sufficient to reliably answer queries grounded in a specific, evolving body of technical documentation without additional mechanisms to anchor its responses to verified sources.

Classical information retrieval systems, such as those based on Term Frequency-Inverse Document Frequency (TF-IDF) and Best Match 25 (BM25), match documents to queries through lexical overlap, identifying relevant content by counting shared keywords rather than understanding meaning. This approach fails when a query and a relevant document use different vocabulary to express the same concept, a limitation that becomes particularly problematic in technical domains where terminology is specialized and variable. Dense semantic embeddings offer a step forward by

representing both queries and documents as vectors in a continuous space, where proximity reflects semantic similarity rather than surface-level word overlap. Reimers and Gurevych demonstrated that pre-trained language models could be adapted through a dual-encoder architecture to produce sentence-level embeddings that can be compared efficiently using cosine similarity, reducing retrieval from computationally prohibitive cross-encoding to a fast nearest-neighbor search [18]. This shift from sparse lexical matching to dense semantic retrieval is what makes it possible to retrieve contextually relevant document passages in response to natural language queries. While bi-encoders are well suited to first-stage retrieval over large collections, cross-encoders process query-document pairs jointly and produce more accurate relevance scores, making them better suited to reranking a smaller set of already retrieved candidates, at the cost of higher computational overhead [18].

2.3 Data Treatment

Before any retrieval can occur, raw documents must be transformed into a form that a system can meaningfully index and search. This preparation phase is not a secondary concern but a foundational one: the quality of what is retrieved depends directly on how well the source material has been parsed, cleaned, and segmented. A RAG system is only as good as the chunks it operates on, and decisions made during ingestion propagate through every subsequent stage of the pipeline. Large language models process text as natural language, and Markdown has emerged as a particularly effective intermediate representation for feeding structured documents into these systems, as it preserves meaningful layout signals such as headers, lists, and tables in a lightweight plain-text format that LLMs can interpret without ambiguity, while spending fewer tokens. For this reason, converting source documents from their original formats, such as PDF, Word, or HyperText Markup Language (HTML), into Markdown before ingestion is an important preprocessing step in RAG pipelines. This parsing step is particularly challenging for technical documentation, where complex page layouts, multi-column tables, and embedded figures must be accurately recognized and mapped into a format that downstream chunking strategies can interpret, as structural information lost at this stage cannot be recovered later.

The simplest approach to document segmentation is fixed-size chunking, which divides text into segments of a predetermined character or token length regardless of the underlying content structure. While straightforward to implement, this strategy frequently cuts across sentence and paragraph boundaries, producing incoherent fragments that degrade both embedding quality and retrieval precision [19]. Recursive splitting strategies attempt to address this by respecting a hierarchy of separators such as paragraphs, sentences, and words, but still rely on size thresholds that impose arbitrary boundaries on semantically coherent content [20]. These limitations motivated the development of semantically aware segmentation methods that determine chunk boundaries based on meaning rather than size [19].

Semantic chunking addresses this by using dense sentence embeddings to detect shifts in

meaning within a document, inserting boundaries at points where the cosine similarity between adjacent sentences drops below a configurable threshold [19]. This produces variable-length chunks that are more semantically coherent than fixed-size alternatives, as each segment tends to contain a single topical unit rather than an arbitrary slice of text [19, 20]. However, the quality of the resulting segments is sensitive to the choice of threshold and embedding model, and performance can vary significantly across domains with different linguistic characteristics [19]. Figure 2.7 provides a visual comparison of the main chunking strategies discussed, illustrating how each approach determines segment boundaries differently [20].

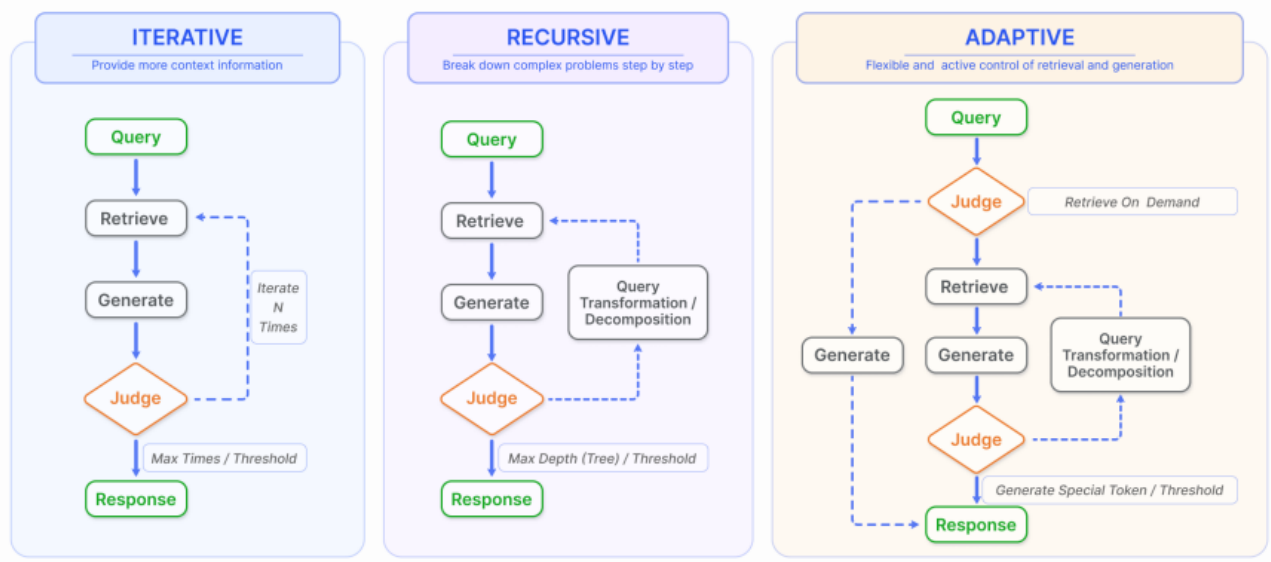

Figure 2.7: Comparison of chunking strategies for document segmentation in RAG pipelines [20].

While semantic chunking improves boundary detection, it treats documents as flat sequences of text and remains blind to the explicit structural signals that technical documents provide through headers, lists, and tables. Structure-aware approaches address this by using the document's own organization as a segmentation guide, splitting content along section boundaries defined by Markdown headers and preserving the metadata of each section within the resulting chunks [21]. Lists and enumerated procedures present a separate challenge, as splitting them mid-sequence destroys the logical dependency between steps, making atomic preservation of list blocks a necessary constraint in technically dense documents [20]. Tables require dedicated handling as well, since their information is encoded in relational row-column structures that lose meaning when treated as plain text and are better extracted as independent retrieval units with their surrounding context preserved [21]. Figure 2.8 illustrates how a structure-aware approach uses the document's own organizational hierarchy to guide segmentation, preserving the integrity of sections, lists, and tables as distinct retrieval units [21].

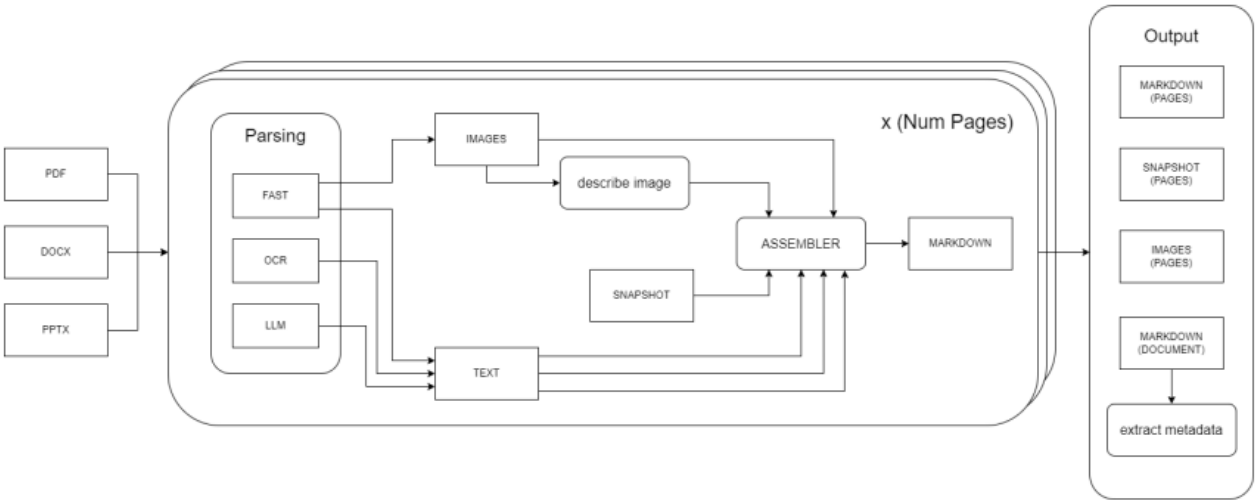

Figure 2.8: Structure-aware document chunking using Markdown hierarchy as a segmentation guide [21].

Underlying all chunking strategies is a fundamental tension between chunk size and context completeness. Smaller chunks improve retrieval precision by producing more focused embeddings that match specific queries more accurately, but risk losing the surrounding context that gives individual sentences their meaning. Larger chunks preserve narrative and procedural coherence but dilute the embedding signal, reducing the likelihood that a chunk will surface in response to a narrow query. There is no universally optimal chunk size, and the right balance depends on the nature of the documents, the query distribution, and the embedding model in use, making the ingestion pipeline a design space that must be carefully calibrated for each specific domain and use case.

2.4 Retrieval-Augmented Generation (RAG)

Retrieval-Augmented Generation (RAG) is a paradigm that addresses the core limitations of standalone LLMs by coupling them with an external retrieval mechanism at inference time. Rather than relying solely on knowledge encoded in model parameters during training, a RAG system dynamically retrieves relevant passages from a document and incorporates them as context before generating a response. This grounds the model's output in domain-specific information, making it possible to keep the system's knowledge current without retraining the model from scratch. The approach is particularly well suited to knowledge-intensive applications where the information required to answer a query that is either too specific, too recent, or too domain-dependent to have been reliably captured during training. Unlike fine-tuning, which permanently alters model weights and requires significant computational resources, RAG separates the knowledge store from the generative model, allowing each to be maintained and improved independently. The overall quality of a RAG system, however, is not determined by any single component in isolation.

Instead, it emerges from the interplay between the retrieval of relevant documents, the selection and ranking of retrieved content, and the synthesis of the final response from the provided context.

In practice, a RAG system operates through two distinct phases. The first is the ingestion pipeline, an offline process responsible for parsing, cleaning, segmenting, and embedding the source documents into a searchable vector index, as discussed in the preceding section. The second is the query pipeline, an online process that, for each incoming question, encodes the query, retrieves relevant passages from the index, ranks and filters the candidates, and passes the selected context to the generative model to produce a final response. Figure 2.9 illustrates this three-stage structure, showing how the offline indexing pipeline and the online query pipeline together form a complete RAG system [20].

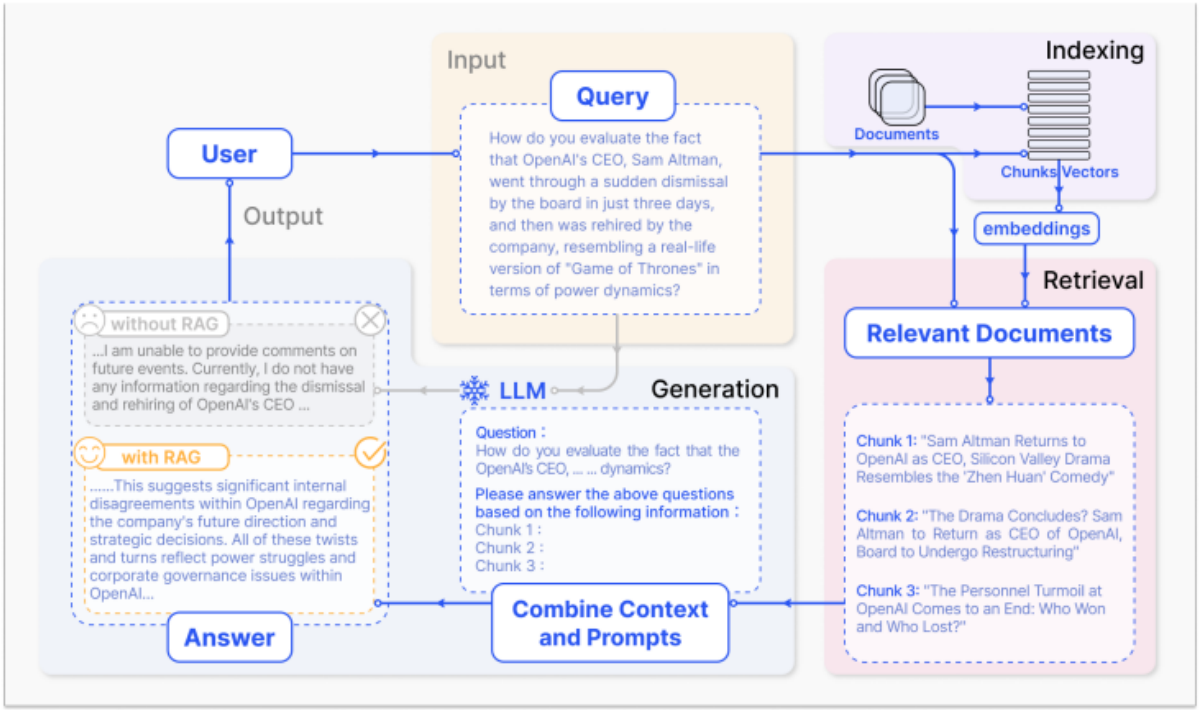

Figure 2.9: Overview of the RAG process, comprising three stages: indexing, retrieval, and generation [20].

2.4.1 RAG pipeline

The original RAG formulation, introduced by Lewis et al., proposed combining a pre-trained retriever with a pre-trained sequence-to-sequence generator, creating a system that could draw on both the parametric knowledge stored in model weights and the non-parametric knowledge contained in an external document index [22]. At inference time, a query is encoded into a dense vector representation and used to retrieve the top-K most similar document passages from the index, which are then combined with the original query and passed to the generator as context for producing the final response [22]. The quality of this process depends critically on the embedding

model that bridges the document store and the retrieval mechanism, as it is responsible for projecting both queries and documents into a shared vector space where semantic similarity can be measured [18]. To prevent the system from generating responses when no relevant content exists in the knowledge base, a minimum similarity threshold can be applied as a guard condition, rejecting queries whose best retrieved match falls below a configurable score and returning a predefined fallback response instead [20]. Unlike fine-tuning, which permanently modifies model weights and requires significant computational resources to incorporate new knowledge, RAG separates the knowledge store from the generative model entirely, allowing the document index to be updated, extended, or replaced without any retraining [22].

Despite its significance as a foundational framework, the original RAG formulation has several limitations that constrain its reliability in practice, collectively referred to as naive RAG [20]. Single-pass retrieval, where the system queries the index once with the original user question and accepts whatever is returned, offers no mechanism to refine or verify the relevance of retrieved content before it reaches the generator [20]. Without a re-ranking step, the top-K retrieved passages are passed directly to the generator regardless of their actual relevance to the query, meaning that irrelevant or partially relevant chunks occupy valuable context space alongside genuinely useful content [20]. This context stuffing problem is compounded when the retrieval window is large, as the generator must process and reconcile a growing volume of potentially contradictory or off-topic information, which can degrade the coherence and factual accuracy of the final response [20]. Ultimately, naive RAG systems are only as good as their retrieval step, and when retrieval fails, generation fails with it, and there is no recovery mechanism built into the pipeline to detect or compensate for poor retrieval quality [20]. Figure 2.10 contrasts these three levels of pipeline sophistication, from the single-pass structure of naive RAG to the modular compositions enabled by more advanced designs [20].

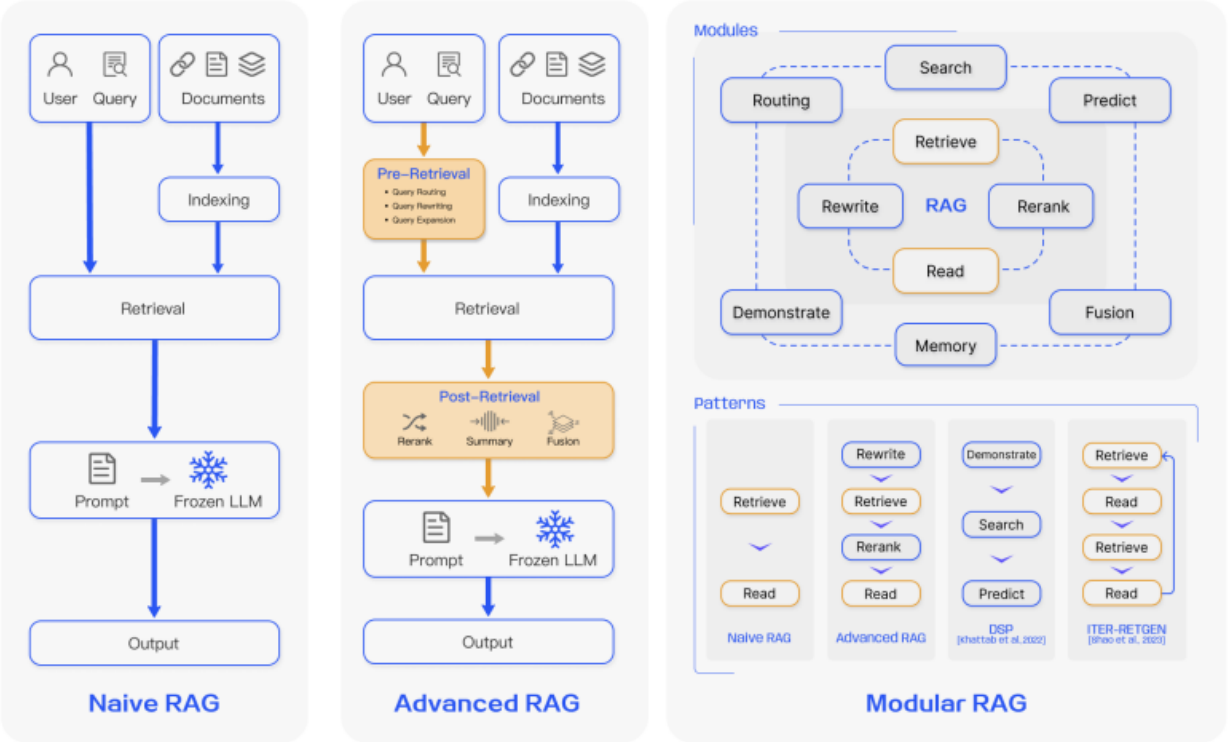

Figure 2.10: Comparison of Naive RAG, Advanced RAG, and Modular RAG paradigms, illustrating the progressive introduction of pre-retrieval, post-retrieval, and orchestration components [20].

2.4.2 Advanced RAG techniques

The limitations of naive RAG have driven the development of a range of techniques aimed at improving retrieval precision, context quality, and overall pipeline reliability. Rather than treating retrieval as a single undifferentiated step, advanced approaches introduce structure at multiple stages of the pipeline, from how documents are segmented and typed before indexing, to how candidates are scored and filtered before reaching the generator. The combined effect of these techniques is a system that is more aware of what it retrieves, more selective about what it passes forward, and more robust in the face of domain-specific challenges such as technical terminology, procedural content, and multilingual queries.

A core limitation of dense retrieval is that the same bi-encoder architecture that enables efficient similarity search across large document collections inherently trades accuracy for speed, encoding queries and documents independently without directly comparing them against each other [18]. To address this, a two-stage retrieve-then-rerank architecture has emerged as a standard pattern in advanced retrieval pipelines, where a fast bi-encoder first retrieves a broad set of candidates and a slower but more accurate cross-encoder subsequently rescores each candidate by jointly encoding the query and the document together [23]. This joint encoding allows the cross-encoder to capture fine-grained interactions between query and document tokens that the bi-encoder cannot, producing a more precise relevance score at the cost of higher computational overhead, which is

acceptable given that it operates on a small pre-filtered candidate set rather than the full document collection [23].

In practice, this two-stage approach can be structured as a sequence of discrete operations applied to each incoming query [20, 23]:

1. **Candidate retrieval** — a bi-encoder performs similarity search over the full document index, returning a broad set of top-K candidate chunks ranked by cosine similarity.
2. **CrossEncoder rescoring** — each candidate is jointly encoded with the query by a cross-encoder model, producing a refined relevance score that captures deeper query-document interactions.
3. **Score adjustment** — domain-specific penalties or boosts are applied to the rescored candidates based on query intent, such as suppressing chunk types that are unlikely to be relevant for a given query category.
4. **Top-N selection** — the re-ranked candidates are cut to a smaller top-N set, keeping the context window focused and preventing irrelevant content from reaching the generator.

Beyond reranking, retrieval quality is also shaped by how documents are segmented before indexing, and a flat uniform chunking strategy treats all content as equivalent when in practice technical documents contain fundamentally different types of information that benefit from different handling [20]. A structure-aware approach recognises these differences by classifying content into distinct chunk types during ingestion, each designed to preserve the semantic and structural integrity of the information it contains [21].

Three primary chunk types can be identified in a technically oriented RAG pipeline [20, 21]:

- **Semantic chunks** — paragraph blocks segmented using embedding-based boundary detection, grouping sentences by semantic similarity to produce coherent variable-length units.
- **List and procedural chunks** — sequential content preserved as indivisible units, ensuring that the logical dependency between steps is never broken by an arbitrary chunk boundary.
- **Table chunks** — tabular content extracted as dedicated retrieval units, each carrying the parent section context and structured as header-value pairs to preserve the relational meaning of the original table.

Advanced RAG pipelines can incorporate retrieval-time score adjustments that modulate the relevance of different chunk types based on signals extracted from the query, a pattern identified as part of the broader move towards context-aware post-retrieval filtering [20]. This allows the pipeline to dynamically favour the chunk type most appropriate for each query category, improving both retrieval precision and the coherence of the context passed to the generator.

A less commonly addressed but practically significant challenge in multilingual RAG systems is the mismatch between the surface form of a query and the forms present in the indexed

documents, particularly in morphologically rich languages where the same word can appear in many inflected forms [24]. Query normalization techniques, including lower casing, punctuation stripping, and lemmatization, address this by reducing query terms to their base forms before embedding, improving the likelihood that semantically equivalent queries and document passages are mapped to nearby positions in the vector space [24]. For morphologically rich languages, lemmatization in particular has been shown to yield substantial improvements in retrieval recall compared to simpler stemming approaches, as it produces linguistically accurate base forms rather than heuristically truncated word stems [24].

2.4.3 Evaluation and observability

Evaluating a RAG system requires moving beyond anecdotal testing and establishing structured mechanisms capable of measuring retrieval quality, generation faithfulness, and end-to-end answer relevance in a repeatable and comparable way. A framework specifically designed for this purpose is RAGAS, which proposes a suite of reference-free metrics capable of evaluating RAG pipelines across three core dimensions: faithfulness, answer relevance, and context relevance, without requiring manually annotated ground truth for every query [25]. Faithfulness measures whether the generated answer is grounded in the retrieved context, answer relevance assesses whether the answer addresses the question that was asked, and context relevance evaluates whether the retrieved passages contain the information necessary to produce a correct response [25]. Faithfulness is computed by decomposing the generated answer into individual statements and verifying each against the retrieved context, such that a response drawing on information not present in the retrieved passages will receive a lower score regardless of its factual accuracy in isolation [25]. Answer relevance is measured by prompting a language model to generate multiple questions from the produced answer and computing the mean similarity between those synthetic questions and the original query, penalizing responses that are technically correct but address a different question than the one posed [25]. Context relevance is assessed by extracting the sentences from the retrieved passages that are necessary to answer the query and computing the proportion of the total retrieved content that those sentences represent, such that a retrieval step returning large volumes of irrelevant content will be penalised even when the correct information is present [25].

A complementary evaluation technique is LLM-as-a-judge, in which a separate language model is prompted to score the outputs of the system under evaluation, offering a scalable alternative to human annotation that has been shown to achieve high agreement with human judgements on open-ended tasks [26]. In the context of RAG evaluation, this technique can be applied to score generated answers along the same evaluation dimensions defined by the RAGAS framework, using a judge model operating at zero temperature to ensure scoring consistency across repeated evaluations [25]. Temperature is a parameter that controls the randomness of a language model's output, with higher values producing more varied responses and lower values making the model more deterministic, so setting it to zero ensures that the judge produces the same score for the same input every time, which is a necessary condition for reliable automated evaluation.

Beyond evaluation, deploying a RAG system in a production environment introduces a distinct set of operational challenges, as the multi-step nature of the pipeline, spanning retrieval, reranking, and generation, makes it difficult to attribute response quality issues to specific components without structured instrumentation [27]. The AgentOps paradigm addresses this by proposing systematic observability for AI agents, enabling the collection of structured traces that record the inputs, outputs, and latency of each step in the pipeline [27]. This makes it possible to correlate end-user feedback with specific pipeline executions and identify failure points without relying on manual inspection [27]. LangFuse is an open-source observability platform that implements this paradigm for LLM applications [27]. It provides hierarchical trace structures that capture each stage of a RAG query as a named span with associated metadata, allowing quality signals from explicit user feedback to be linked directly to the pipeline runs that produced them [27].

2.5 Summary

This chapter has surveyed the theoretical foundations and state of the art underlying the design of a RAG-based assistant for industrial technical documentation, structured around three interconnected areas: the industrial and manufacturing context that motivates the work, the language model capabilities that enable it, and the retrieval and generation pipeline that implements it.

The industrial context established that the transition towards Industry 4.0 and 5.0 has created a growing demand for intelligent systems capable of bridging the knowledge gap between complex technical documentation and the operators who depend on it, with rotary draw bending machinery serving as the concrete domain of application, a process characterized by precise mechanical constraints, rich procedural documentation, and a specialized technical vocabulary that places particular demands on any retrieval system.

LLMs, grounded in the transformer architecture and scaled through instruction tuning and reinforcement learning from human feedback, have demonstrated remarkable capacity for language understanding and generation, yet their reliance on static parametric knowledge introduces fundamental limitations in the form of hallucination and knowledge cutoff, limitations that make them unsuitable as standalone solutions for domain-specific technical assistance.

The data treatment literature established that the quality of a RAG system is critically dependent on how source documents are segmented before indexing, with structure-aware chunking strategies that respect the semantic and structural boundaries of technical content consistently outperforming uniform fixed-size approaches, particularly for documents containing procedural sequences and tabular data.

Retrieval-augmented generation addresses the limitations of standalone language models by grounding generation in an external, updatable knowledge base, and the evolution from naive single-pass retrieval to advanced pipelines incorporating cross-encoder reranking, structure-aware ingestion, query normalization, and post-retrieval filtering represents a progressive effort to close the gap between retrieval theory and the precision demands of real-world technical applications.

Evaluation and observability frameworks, including RAGAS metrics and LLM-as-a-judge scoring, provide the structured measurement mechanisms necessary to move beyond anecdotal testing and quantify pipeline performance across retrieval quality, generation faithfulness, and answer relevance, while observability platforms such as LangFuse enable the instrumentation required to trace failures to specific pipeline components in a production environment.

Despite the maturity of the individual techniques surveyed, no single established pipeline configuration has been shown to consistently achieve both retrieval precision and generation faithfulness across domain-specific industrial documentation, and this work positions itself within that gap, aiming to design, implement, and evaluate a RAG pipeline that is as effective and efficient as possible for the technical documentation of AMOB's rotary draw bending machinery.

Chapter 3

System architecture and design

This chapter presents the architecture and design of the retrieval-augmented generation system developed to support natural language querying of technical documentation for rotary draw bending machines. Section 3.1 introduces the overall system structure, formalizing it as two distinct pipelines, and describes the layered architecture that organizes its components. Section 3.2 describes the ingestion pipeline, detailing how raw Markdown documentation is parsed, segmented, and indexed into the vector store. Section 3.3 covers the query pipeline, from query preprocessing through context assembly to response generation. Section 3.4 presents the evaluation architecture, a first-class component through which the system measures the quality of its own responses. Finally, Section 3.5 presents the frontend interface and its design rationale.

3.1 System overview

The system is a modular retrieval-augmented generation application designed to provide operators of rotary draw bending machines with accurate, context-grounded answers to natural language queries in Portuguese, drawn directly from the technical documentation of the equipment in use. Rather than relying on a standalone large language model with static parametric knowledge, the system grounds its responses in the most relevant passages retrieved from an indexed corpus of Markdown documentation, or explicitly declines to answer when no sufficiently relevant content is found. This retrieval grounding mitigates the limitations of hallucination and knowledge staleness identified in Section 2.2, anchoring responses to verifiable source material and allowing the underlying documentation to be updated without retraining the model. The following sections describe the architecture of the system, the design decisions that shaped it, and the rationale behind each of its core components.

At the highest level, the system is organized into two distinct pipelines that are decoupled, operating at different times and in distinct contexts. The ingestion pipeline runs offline, transforming the raw Markdown documentation into a searchable knowledge base: source documents are parsed, cleaned, segmented into coherent units, and converted into dense vector representations that are stored in a persistent vector database. This process is executed once for a given set

of documents and does not need to be repeated unless the underlying documentation changes or the chunking configuration is modified. The query pipeline, by contrast, runs online in response to each user question: it retrieves the most relevant segments from the previously indexed knowledge base, refines that selection, and uses it to generate a grounded natural language answer. Separating these two concerns allows each pipeline to be developed and configured in isolation, since the quality of the indexed knowledge base produced by the ingestion pipeline determines the upper bound on what the query pipeline can retrieve and, ultimately, on the quality of the responses delivered to the user.

Internally, the system follows a layered architecture that separates the user-facing interface, the stable boundary through which it is accessed, the logic that orchestrates the query pipeline, and the individual components that perform the underlying work. At the top, a web interface handles all interaction with the user, capturing questions and presenting the generated answers along with any supporting figures or tables. Beneath it, a public application programming interface (API) acts as a stable boundary, exposing a small set of high-level operations such as querying the system, loading documents, and resetting the knowledge base, while hiding the complexity of how each operation is carried out. Both pipelines are reached through this boundary, but they are coordinated differently: querying is routed through a dedicated orchestration layer that sequences the steps of the query pipeline, whereas document loading is coordinated within the service responsible for the vector store, which drives the ingestion process directly. The remaining work is delegated to a set of services that form the lowest layer, each encapsulating a single responsibility, such as managing the vector database, producing embeddings, reranking candidates, or communicating with the language model. This decomposition into isolated services keeps each concern self-contained and independently replaceable, so that many components can be substituted or reconfigured with little impact on the rest of the system, although some changes propagate further, as replacing the embedding model requires the vector store to be rebuilt. This same modularity is what later supports the systematic enabling and disabling of individual components to assess their contribution. Figure 3.1 presents the overall structure of the system and the relationship between the two pipelines.

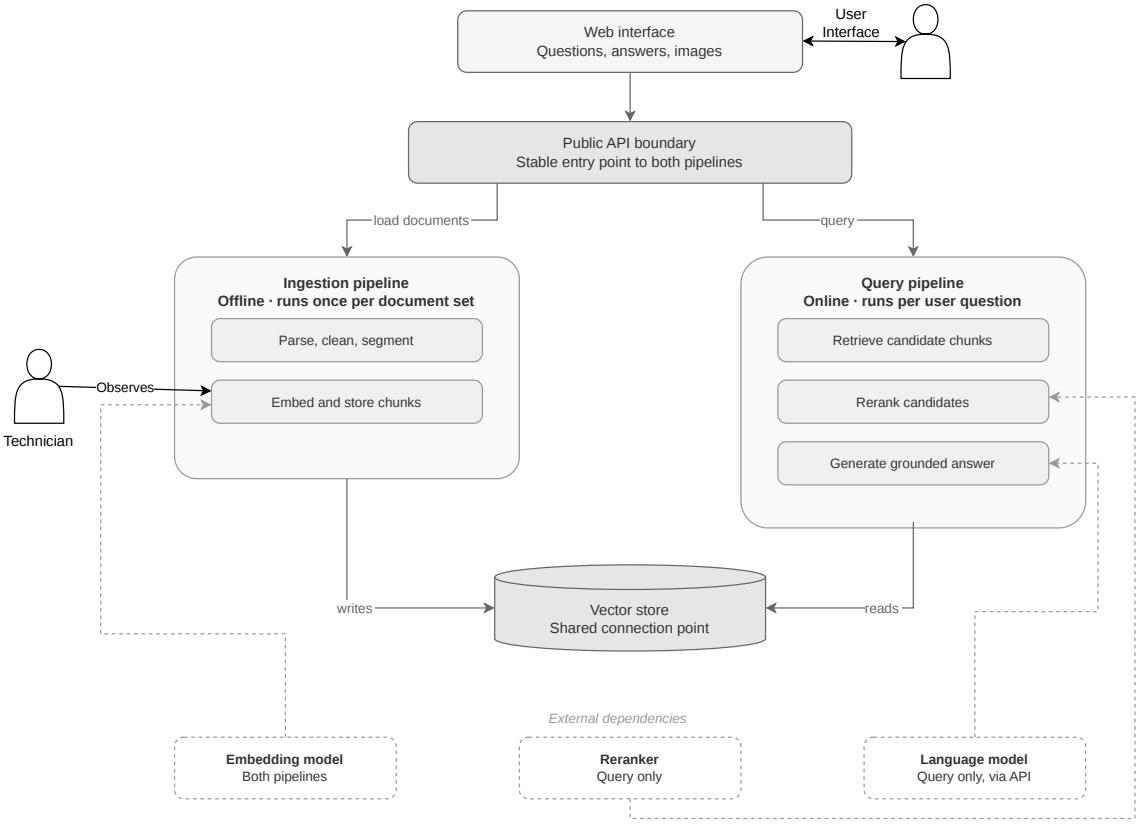

Figure 3.1: Global architecture of the system, showing the two decoupled pipelines and their shared connection through the vector store.

3.2 Ingestion pipeline

The ingestion pipeline is responsible for transforming the human-readable technical documentation into a structured, searchable knowledge base that the query pipeline can later draw upon. Because every answer the system produces is grounded in the segments retrieved from this knowledge base, the decisions made during ingestion set an upper bound on the quality of everything that follows, which makes this stage foundational rather than preparatory. The pipeline takes documentation in Markdown format, a lightweight markup language that uses plain-text symbols to represent formatting elements such as headers, lists, and tables, as its input, and passes it through a sequence of stages that progressively refine the raw text into coherent, self-contained units suitable for retrieval. Figure 3.2 illustrates the complete ingestion pipeline, from the Markdown source documents through type-specific segmentation to the enriched chunks stored in the vector store. These stages begin by cleaning the source text and removing structural noise, then segment the content along its natural boundaries while giving special treatment to elements such as tables and lists that require structure-aware treatment to remain semantically useful. The resulting units are finally converted into dense vector representations and stored in the vector database, where they become available to the query pipeline. The following subsections describe each stage of this process.

3.2.1 Document preprocessing

The source documentation is provided as PDF files, a format designed for visual presentation rather than machine interpretation. Before ingestion can begin, each document is therefore converted into Markdown using Docling [28], an open-source toolkit purpose-built to turn complex documents into representations suited to large language model pipelines. The choice is motivated by the nature of the two formats: extracting text directly from PDF tends to introduce structural noise such as repeated headers and footers, page numbers, and layout artifacts that inflate the volume of text without adding meaning and that degrade the quality of later retrieval. Markdown, by contrast, encodes the semantic structure of a document, its headings, lists, and tables, as explicit textual markers, exposing the organization of the content in a form that the subsequent stages can interpret directly and with far less noise. Following conversion, the resulting Markdown files were manually reviewed and curated to correct residual artifacts introduced by the automated conversion process, ensuring that the structural integrity of the source material was preserved before ingestion began. The conversion is not flawless, however, and may leave residual artifacts that require cleanup, which is precisely what the first preprocessing stage addresses.

The first stage prepares the raw Markdown for segmentation by removing content that would otherwise pollute retrieval. Technical manuals typically open with a table of contents, a navigational aid consisting largely of section titles paired with page numbers. If left in place, this material would be indexed as though it were substantive content, producing chunks that match many queries by keyword while carrying no real information. It is therefore removed before any further processing. The same stage also removes the structural noise left by the conversion itself, such as residual image placeholders, stray figure labels, and superfluous blank lines, leaving a cleaner text on which the subsequent stages can operate reliably.

Once cleaned, the document is divided into sections according to its own heading structure, with each second-level heading marking the boundary of a new section. This produces an initial set of section-level units that inherit their heading as metadata, preserving the position of each piece of content within the overall structure of the manual.

3.2.2 Table extraction strategy

Tables pose a particular challenge for segmentation because their meaning is encoded in the relationships between rows and columns rather than in a linear sequence of sentences. Treating a table as ordinary running text would flatten this structure, scattering values away from the labels that give them meaning and producing fragments from which the original information cannot be reconstructed. To avoid this, tables are detected and extracted before the surrounding text is segmented, and each is handled as an independent unit rather than being left to the general-purpose splitting that follows. Extracting them first also prevents the table markup from interfering with the boundaries chosen for the remaining prose.

Not all tables share the same structure, so a single extraction rule would serve some well and others poorly. Three strategies are therefore applied according to the structure of each table:

- **Property-value tables**, which pair a description or a label, such as a component name, with a value, such as weight or measurement, are merged into a single chunk and reformatted as natural-language sentences, so that each value remains attached to the label that describes it.
- **Merged-cell tables**, whose cells span several rows, are consolidated so that the spanning relationship is preserved rather than broken apart.
- **Generic tables**, which follow a regular row-by-row structure, are split so that each row becomes a self-contained unit.

Regardless of the strategy applied, tables that carry no identifying number in the source document are assigned a synthetic identifier, ensuring that every extracted table can still be referenced unambiguously by later stages of the system, including the resolution of any associated images.

3.2.3 Structural block splitting

With tables removed, the text that remains within each section still mixes two kinds of content that call for different treatment: ordinary prose, which can be divided freely at sentence or paragraph boundaries, and lists, whose items form an ordered or interdependent sequence. Before any further segmentation, the remaining content is therefore separated into structural blocks, distinguishing prose paragraphs from numbered or bulleted lists so that each can be handled appropriately.

Lists are treated as atomic units and are never split internally. A procedure expressed as a numbered sequence of steps loses its meaning if divided partway through, since each step depends on the ones before it, and a retrieved fragment containing only some of the steps could mislead an operator rather than guide them safely. Keeping each list intact ensures that a procedure is always retrieved in full. A list on its own, however, can be difficult to interpret without the sentence that introduces it, which often states the purpose or the conditions of the procedure. To preserve this context, the introductory sentence that immediately precedes a list is prepended to the list block, so that the retrieved unit carries both the instruction to act and the steps required to carry it out.

3.2.4 Semantic chunking

At this point in the pipeline, tables have been extracted as independent units and lists have been preserved as atomic blocks, leaving only the prose paragraphs still in need of further segmentation. These paragraphs vary considerably in length and may cover several distinct topics within a single section, so dividing them into smaller, more focused units is necessary to improve the precision of later retrieval. The default strategy employs semantic chunking, the embedding-based approach introduced in Section 2.3, which uses the same embedding model that will later be used for retrieval to measure the semantic similarity between consecutive sentences. Each sentence is encoded as a dense vector, and the cosine similarity between adjacent sentence vectors is computed across the entire paragraph. Cosine similarity measures the cosine of the angle between two

vectors: when two sentences are topically related their vectors point in similar directions, the angle between them is small, and the cosine approaches 1. When the topic changes the vectors diverge, the angle widens, and the cosine drops toward 0. The chunker monitors all such drops within a document and inserts a boundary at points where the drop exceeds the 90th percentile of all observed drops, meaning only the sharpest transitions in the embedding space trigger a new chunk. This produces variable-length chunks that tend to correspond to coherent topical units rather than arbitrary slices of text, since the boundaries are driven by the content itself rather than by a fixed character or token count.

3.2.5 Post-chunking merging and header prepending

After all segmentation strategies have been applied, the resulting chunks, prose segments from semantic chunking, atomic list blocks, and extracted table units, are brought together and sorted according to their original position in the source document. At this stage, some chunks are too small to function as meaningful retrieval units on their own, typically orphaned figure captions or short transitional sentences left behind after the surrounding text was absorbed by other blocks. Rather than indexing these fragments independently, where they would contribute noise without carrying enough context to produce useful answers, they are merged into the nearest adjacent chunk, consolidating the final set into units that each carry sufficient substance to be worth retrieving.

The final enrichment step prepends the section header to the content of every chunk before its embedding is computed. Without this addition, a chunk drawn from deep within a section would be embedded based solely on its own text, with no trace of the broader topic it belongs to. By including the section header in the content that the embedding model sees, the resulting vector representation captures not only what the chunk says but also where it sits within the structure of the document. This synthetic context enrichment improves retrieval precision, because a query about a particular topic is more likely to surface chunks that were written under the relevant section heading. Once enriched, the chunks are converted into dense vector representations using the embedding model and stored in ChromaDB [29], an open-source vector database that persists collections of embeddings to disk and supports efficient similarity search through approximate nearest-neighbour indexing, allowing the query pipeline to retrieve the most relevant chunks at inference time without reloading or recomputing any embeddings. This completes the ingestion pipeline and makes the knowledge base available to the query pipeline.

Taken together, these stages transform a raw Markdown document into a set of enriched, self-contained chunks ready for retrieval, applying distinct treatment to tables, lists, and prose so that each retains the particular form of meaning it carries. Figure 3.2 summarizes the complete pipeline, showing how the content is separated by type once the document has been divided into sections, and how the resulting chunks are subsequently brought back together, enriched, and stored.

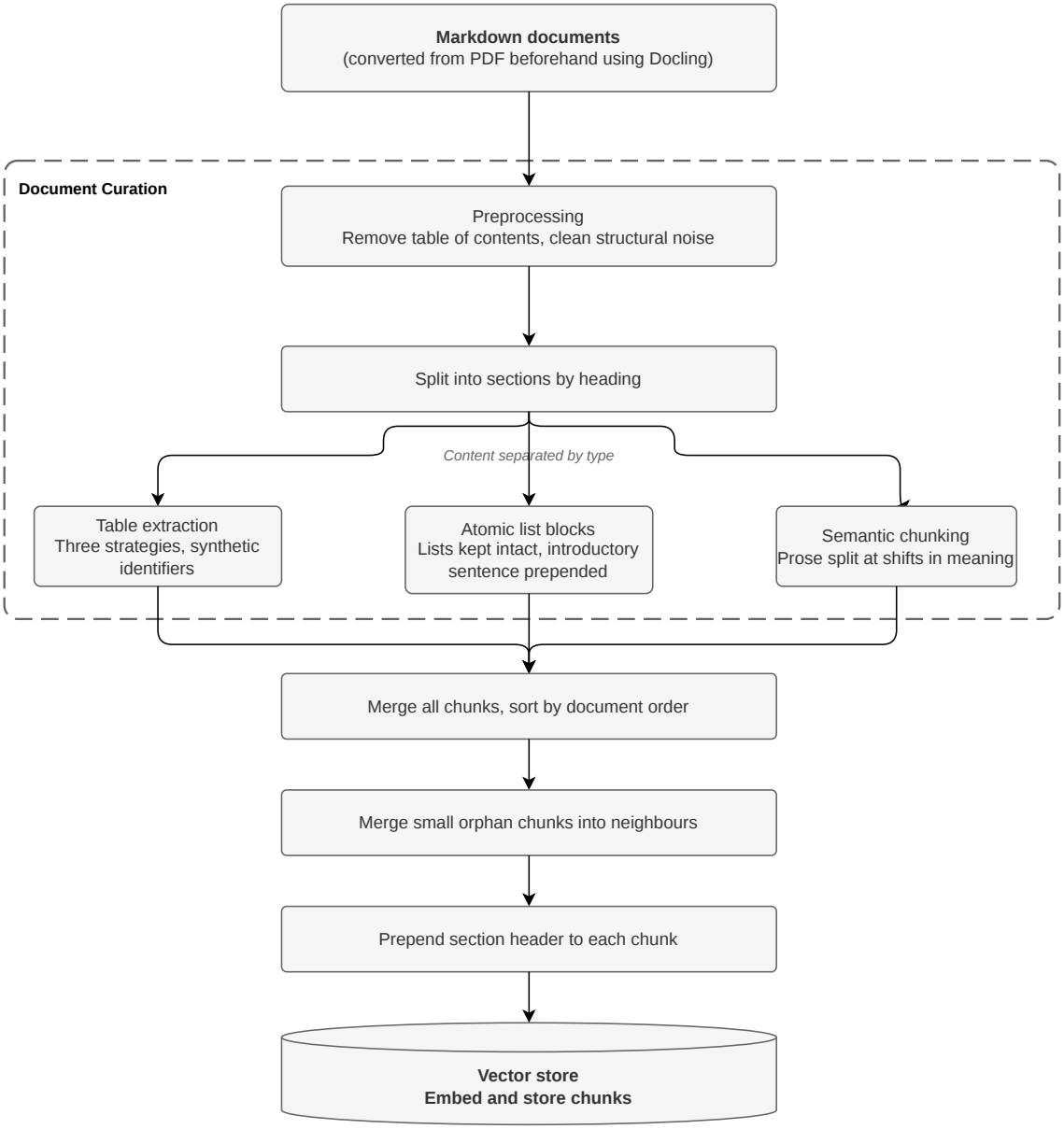

Figure 3.2: The ingestion pipeline, from the Markdown source documents through type-specific segmentation to the enriched chunks stored in the vector store.

3.3 Query pipeline

The query pipeline executes in response to each incoming query, transforming a natural language question from the user into a grounded answer drawn from the indexed knowledge base produced during ingestion. Where the ingestion pipeline is concerned with preparing and structuring information, the query pipeline is concerned with locating the small portion of that information relevant to a specific question and presenting it to the language model in a form that supports an accurate response. Figure 3.3 illustrates the complete query pipeline, from the user's question through two-stage retrieval and context assembly to the grounded answer with resolved images and tables. The

pipeline begins by preparing the incoming question, then retrieves a broad set of candidate chunks from the vector store and progressively narrows them to the few most relevant, which are assembled into a context and passed to the language model for answer generation. A central design principle is the use of two-stage retrieval, in which a fast similarity search first gathers a broad pool of candidates and a more precise but computationally heavier reranking step then reorders them, balancing efficiency against accuracy in a way that neither stage could achieve alone. The following subsections describe each stage of this process.

3.3.1 Query preprocessing

Before the query reaches the vector store, it undergoes a normalization step that reduces surface-level linguistic variation which would otherwise weaken retrieval. Most large language models and embedding models are trained predominantly on English corpora, which means their representations of morphologically complex languages such as Portuguese are inherently less robust. Portuguese is a morphologically rich language in which the same verb can appear in dozens of conjugated forms, and technical documentation tends to use the infinitive while operators phrase their questions using conjugated forms. If the query is passed to the embedding model without normalization, these inflectional differences can reduce the cosine similarity between a question and the chunk that answers it, causing relevant content to rank lower than it should. The system addresses this through a targeted normalization step: the query is lowercased, accented characters are normalized to their unaccented equivalents (for example, *ç* becomes *c*, *ã* becomes *a*, and *é* becomes *e*), and a fixed lookup table maps a curated set of domain-relevant conjugated forms back to their infinitives (for example, *dobrar* is mapped back from its conjugated forms such as *dobra* or *dobrando*), so that the embedded query aligns more closely with the vocabulary of the indexed documentation.

A second preprocessing step expands the query with domain-specific synonyms. In rotary draw bending machines, the same physical component or process is often referred to by more than one name depending on the context or the convention of a particular manual. An operator asking about a component by one name would fail to retrieve documentation that describes the same component under a different name, even though the underlying content is exactly what they need. For example, the lateral support tool is referred to in the documentation as both *eixo da paralela* and *eixo do encosto*, and the bending die itself is known interchangeably as *meia cana* and *meia lua*. To prevent retrieval failures caused by this terminological variation, the system maintains a mapping of axis-name and component synonyms and appends the known alternatives to the query before retrieval, ensuring that a match can occur regardless of which name the operator or the documentation happens to use.

3.3.2 Two-stage retrieval rationale

Retrieval is carried out in two stages, the first of which is a dense similarity search over the entire knowledge base. As described in Section 2.2 [18], the embedding model encodes the query

and each stored chunk independently into vectors whose proximity reflects semantic similarity. This independence between query and document encoding is what makes the approach efficient, because the two operations happen at different points in time: during ingestion, every chunk is encoded once and its vector is stored in the database; at query time, only the incoming question needs to be encoded, after which the database performs a fast nearest-neighbor search against all stored vectors to retrieve the fifty most similar chunks. Because the computational cost of encoding the corpus is paid upfront during ingestion, the query-time search reduces to a single encoding operation followed by a vector lookup, which is what allows it to scan the full collection in a fraction of a second. Its weakness, however, is precision: because the query and each chunk are encoded separately, the model never examines them together, and subtle distinctions that determine whether a chunk truly answers the question might be lost.

The second stage addresses this weakness by reordering the candidate pool with a more discriminating but computationally heavier model. Unlike the bi-encoder used in the first stage, this model cannot rely on pre-computed vectors: it must process the query and each candidate chunk together as a single input, running a full forward pass for every pair. This makes it far more accurate, since it can attend to the specific interactions between the words of the query and the words of the chunk, but it also means the computational cost scales linearly with the number of candidates, making it impractical to apply to the entire collection. The two-stage design resolves this tension by using the fast vector search to produce a broad pool of candidates, of which the fifteen most promising are passed to the cross-encoder for precise reordering. This funnel arrangement combines the efficiency of dense retrieval with the accuracy of joint query-document scoring, achieving a quality of ranking that neither stage could provide on its own at a cost that remains acceptable for interactive use. The mechanics of this second stage are described in the following subsection.

3.3.3 Similarity threshold gating

A dense similarity search will always return a ranked set of candidate chunks, even when the question falls entirely outside the scope of the documentation, because the search simply returns the nearest neighbors regardless of how distant they are. Passing these irrelevant chunks to the language model would invite exactly the kind of hallucinated response the system is designed to avoid, since the model would attempt to construct an answer from context that does not actually address the question. To guard against this, a similarity gate is applied immediately after the first retrieval stage. The similarity score of the best-matching chunk is compared against a minimum threshold, and if even the closest match falls below it, the system concludes that no sufficiently relevant content exists in the knowledge base. In that case it does not proceed to generation, returning instead a predefined fallback response that informs the user the question cannot be answered from the available documentation.

This gate is a deliberate reliability safeguard rather than a limitation. In an industrial setting, an answer fabricated from unrelated material is more dangerous than an honest acknowledgment that the information is not available, since an operator may act on it. By declining to answer when the

evidence is insufficient, the system favours a transparent refusal over a confident but ungrounded response, consistent with the grounding principle established at the outset of this chapter.

3.3.4 CrossEncoder reranking and table penalty heuristic

The second stage of retrieval reranks the candidate pool using a cross-encoder, a model that takes a query and a candidate chunk together as a single input and produces a relevance score for the pair. This differs fundamentally from the dense retrieval of the first stage, where the query and each chunk are encoded separately and compared only through the distance between their vectors. By processing the two together, the cross-encoder can attend to the specific ways in which the words of the query relate to the words of the chunk, capturing subtle relevance signals that independent encoding cannot represent. This makes it considerably more accurate at judging whether a chunk genuinely answers a question, which is why it is applied as a precise second pass over the smaller candidate set produced by the first stage. The chunks are reordered according to their cross-encoder scores, and only the five highest-ranked are retained to form the context from which the language model generates its response.

A domain-specific adjustment is applied during this reranking to counter a recurring retrieval problem. Because table chunks are dense with keywords such as component names and numerical values, they tend to score highly even for questions that are not seeking tabular information, displacing the prose passages that would better answer such questions. To address this, the system applies a penalty to the score of table chunks whenever the query does not appear to be about tabular content, lowering their rank so that prose is favoured when prose is what the question calls for. Whether a query is seeking tabular content is determined by the presence of a small set of indicative terms associated with tables and the kinds of information they typically hold, such as maintenance schedules and periodic intervals. This detection is a binary decision: if any indicative term is found in the query, the query is classified as table-seeking and no penalty is applied to any table chunk; if no term is found, the penalty is applied uniformly to every table chunk in the candidate pool, regardless of how relevant any individual table chunk might actually be. This uniform application is a deliberate simplification that avoids the complexity of a per-chunk confidence assessment, though it also means the system cannot distinguish between table chunks that happen to be useful for a non-table query and those that are genuinely irrelevant. The direction of this adjustment is fixed by design, while its strength and the set of indicative terms are tunable parameters.

3.3.5 Context assembly and multi-turn chat history

Once the five highest-ranked chunks have been selected, they are assembled into a single context block that will accompany the user's question when it is passed to the language model. Each chunk is tagged with the name of the source document it was drawn from, so that the language model can identify where each piece of information originates and, when appropriate, include that attribution in its response. This source tagging also provides the user with a way to trace any claim

in the answer back to a specific document, reinforcing the grounding principle that distinguishes this system from a standalone language model. The chunks are presented in the order determined by the reranker, preserving the relevance ranking established during the previous stage.

The system supports multi-turn conversational interaction rather than treating each question in isolation. When a user asks a follow-up question, the full history of previous exchanges in the current session is included alongside the retrieved context, allowing the language model to interpret the new question in light of what has already been discussed. This means the model can resolve references to earlier answers, refine a previous response when asked, or build on information that was already retrieved in a prior turn. The current design passes the complete session history without truncation or turn limits, relying on the capacity of the model's context window as the only constraint. This is a prototype simplification that prioritises conversational coherence, and would need to be revisited as session lengths grow, since the combined input, comprising history, retrieved context, and current question, is bounded by the language model's context window.

3.3.6 LLM generation, model selection and image extraction

With the context assembled and the chat history included, the system sends the complete prompt to a large language model for answer generation. The model is accessed through a cloud inference API, which allows the system to use large-scale models without requiring local hardware capable of running them. The generation temperature is set to zero, producing deterministic outputs that do not vary between runs for the same input. This choice is motivated by the need for reproducibility, both during development, where consistent outputs simplify debugging, and during evaluation, where non-deterministic responses would introduce uncontrolled variation into the scoring of the system's answers.

The system defines two distinct model roles. The generation model, used to produce answers for the end user, defaults to the largest of the models used in this system, selected to maximise answer quality. A separate, lightweight model is reserved exclusively for the evaluation framework, where it acts as the judge that scores the system's answers, as described in Section 3.4. This separation ensures that the model being evaluated and the model doing the evaluation are never the same, avoiding the circularity that would arise if a model were asked to judge its own outputs. The generation model can be overridden through a configuration parameter, which provides the operational flexibility needed to run manual comparison experiments with alternative models.

After the answer has been generated, a post-processing step extracts any references to figures or tables that the language model has included in its response. These references are matched against the identifiers of the images associated with the documentation, and the corresponding image files are resolved so that they can be displayed alongside the textual answer in the user interface (UI). If the generated answer does not mention any figures or tables explicitly, a fallback mechanism scans the top retrieved chunks for references, ensuring that relevant visual material can still be surfaced even when the model does not cite it directly. This image extraction step bridges

the gap between the text-based retrieval pipeline and the visual content embedded in the source documentation.

Taken together, these stages transform a natural language question into a grounded, source-attributed answer by progressively narrowing a broad set of candidates to the few most relevant and presenting them to the language model alongside the conversational context. Figure 3.3 summarizes the complete query pipeline, from the initial preprocessing of the question through retrieval, reranking, and generation to the final extraction of any referenced figures or tables.

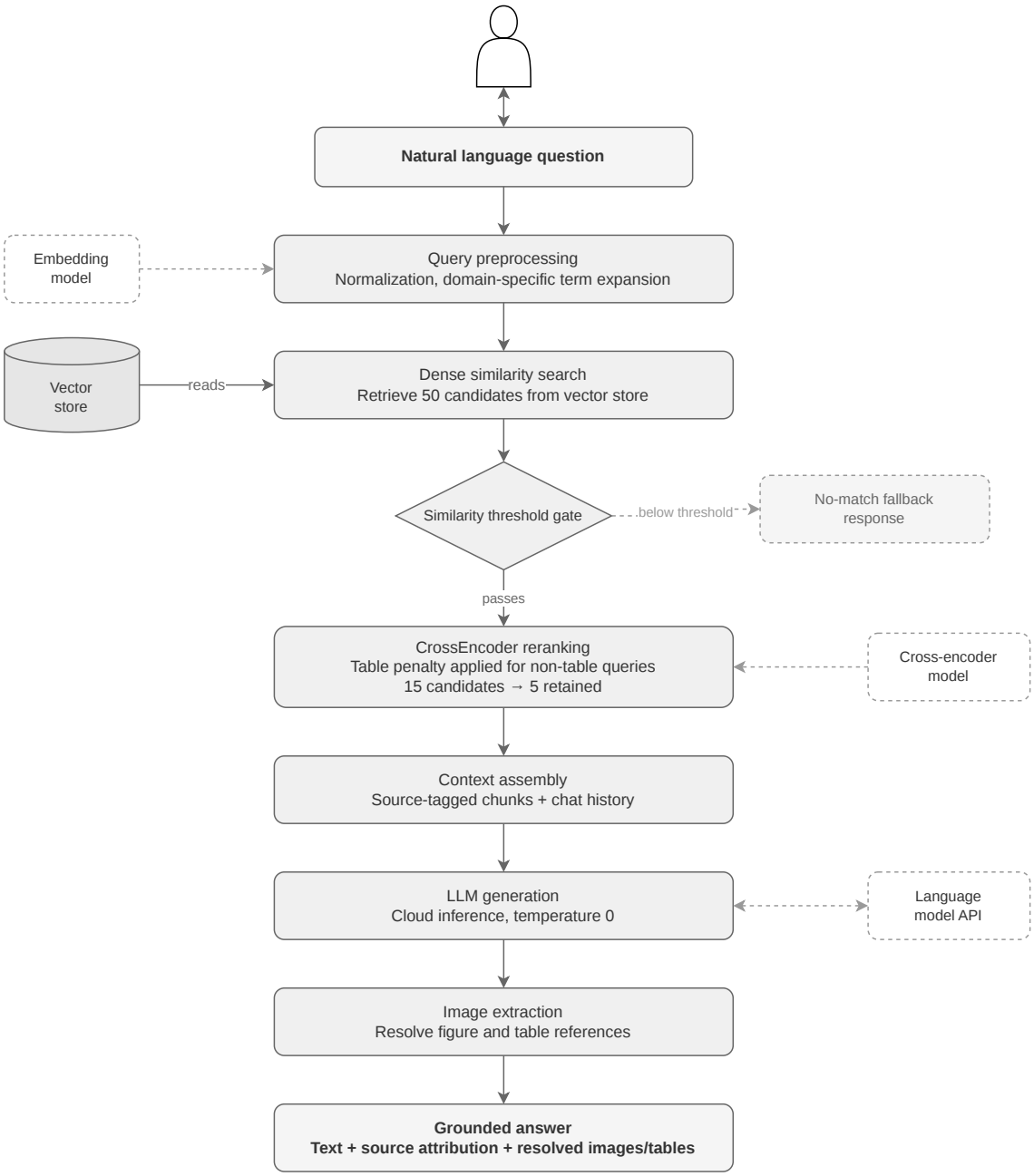

Figure 3.3: The query pipeline, from the user's question through two-stage retrieval and context assembly to the grounded answer with resolved images and tables.

3.4 Evaluation architecture

The evaluation machinery is not an external process applied to the system after the fact, but a first-class component built into the architecture itself. At its foundation is a golden dataset: a fixed set of twenty Portuguese question-answer pairs, each grounded in the actual technical documentation the system is designed to query. The system evaluates itself by running the full pipeline against every question in this dataset, producing a retrieved context and a generated answer for each, and then applying two scoring layers to those runs: an automated metrics framework that quantifies retrieval and answer quality through deterministic comparisons against the reference answers, combining embedding-based semantic similarity with exact keyword overlap and retrieval hit rate measures, and a judge model, separate from the generation model, that scores each response on faithfulness, relevance, and context quality, providing an assessment that approximates human judgment without requiring a human in the loop for every question. Outside the golden dataset loop, two further mechanisms operate on live queries as the system is used. An observability layer traces the internal behaviour of each query through the pipeline, recording spans and intermediate results so that failures can be diagnosed rather than merely detected. A user feedback mechanism captures human ratings in addition to the automated and judge-based scores, providing a lightweight qualitative signal that complements the formal evaluation. The following subsections describe the golden dataset and each of these mechanisms in turn.

3.4.1 Golden dataset

The golden dataset is a curated benchmark of twenty question-answer pairs, written in Portuguese and grounded in the technical documentation of the eMOB-52 rotary draw bending machine. Each entry consists of a question that an operator might realistically ask, a reference answer that has been manually verified against the source material, and the identity of the source documents and section from which the answer is drawn. A question may reference more than one source document, reflecting the fact that some answers require information drawn from multiple parts of the documentation. This structure allows the evaluation framework to assess not only whether the system's generated answer matches the expected content, but also whether the retrieval stage located the correct sources, providing separate visibility into retrieval quality and generation quality. When the evaluation is executed against this dataset, retrieval is restricted to the eMOB-52 manual, ensuring that retrieved chunks are drawn exclusively from the document the questions and reference answers were constructed against.

The twenty questions are distributed across six categories: procedural, technical specifications, maintenance, troubleshooting, safety, and operation. The distribution is not uniform, with procedural questions making up the largest share of the dataset, reflecting their prevalence in everyday operator interactions with the machine. The remaining categories are represented with fewer entries each, but their inclusion ensures that the benchmark exercises the breadth of the documentation rather than concentrating exclusively on one type of query, so that weaknesses in specific areas can be identified rather than masked by strong performance in the most common

category. Each pair has been manually verified to confirm that the reference answer is correct, complete, and traceable to the cited sources, which is what gives the benchmark its authority as a ground truth against which the system's outputs are measured.

3.4.2 Automated metrics framework

The automated metrics framework provides a set of deterministic, reproducible scores that can be computed without any language model involvement, making them fast to run and stable across evaluations. They are designed to capture different facets of system performance, and are divided into two levels. Answer-level metrics compare the system's generated answer against the reference answer in the golden dataset, measuring whether the content produced is semantically and lexically close to what was expected. Retrieval-level metrics instead examine whether the retrieval stage found the correct source documents, independently of whether the final answer was well-formed. This separation is important because it allows a poor result to be attributed to the right cause: a correct retrieval followed by a weak generation points to a language model issue, while a strong generation from the wrong source suggests a retrieval failure. Four metrics are used:

- **Semantic similarity** measures how close the generated answer is to the reference answer in meaning, using the same embedding model that powers the query pipeline to represent both as vectors and comparing them through cosine similarity. This captures whether the answer conveys the right information even when the exact wording differs.

- **Keyword overlap** complements semantic similarity with a surface-level check, computing the Jaccard similarity [30] between the sets of terms in the generated and reference answers after filtering out common stopwords. Jaccard similarity is defined as the size of the intersection of two sets divided by the size of their union, producing a score between zero and one that reflects how much the two sets share relative to their combined vocabulary. Before comparison, both answers are passed through a normalization step that removes structural content introduced by the language model at generation time: inline citations, Markdown formatting, list markers, framing intro phrases, and reference blocks. Tables are flattened by discarding header rows and reducing each data row to a plain line of cell content, ensuring that column labels do not inflate the token union without contributing to the intersection. After normalization, tokens are lowercased, accent-normalized, and filtered to remove short tokens and common stopwords. Tokens are then passed through the Removedor de Sufixos da Língua Portuguesa (RSLP) algorithm [31], a Portuguese stemmer which collapses inflected word forms to a common stem so that, for example, the verb forms *ligue*, *ligar*, and *ligado* are treated as the same term. Because Jaccard measures the intersection over the union, it penalises both missing terms and extraneous terms, catching cases where the meaning is close but key technical vocabulary is absent or diluted by irrelevant content.

- **Answer completeness** assesses whether the generated answer covers all the key points present in the reference answer. The reference answer is segmented into individual points,

and for each point the metric checks whether a sufficient proportion of its key terms appear in the generated answer, counting the point as covered only if enough of its vocabulary is present. This segmentation works most reliably when the reference answer is structured as a numbered sequence of steps or a semicolon-separated list, and degrades to a rougher heuristic when the reference is written as continuous prose. As a result, the metric is both most relevant and most reliable for procedural answers, where omitting a step could lead an operator to skip a critical part of a process, while for non-procedural questions its scores should be interpreted with more caution.

- **Retrieval hit rate** operates at the retrieval level, checking whether the expected source documents identified in the golden dataset appear among the source metadata of the chunks the system actually retrieved. The result is binary: a question scores a hit if any expected document was retrieved and a miss otherwise. This metric isolates retrieval performance from generation quality, making it possible to determine whether a failure originated in finding the right content or in producing the right answer from it.

3.4.3 LLM-as-a-judge

The automated metrics described in the previous subsection are deterministic and reproducible, but they operate at the level of surface overlap and embedding distance, which limits what they can assess. They cannot judge whether a generated answer is faithful to the retrieved context rather than introducing unsupported claims, whether it actually addresses the question that was asked rather than a related one, or whether the retrieved chunks were genuinely useful for answering rather than merely similar to the query. To capture these deeper qualitative dimensions, the system employs an LLM-as-a-judge: a lightweight language model, separate from the generation model, that reads the question, the retrieved context, and the generated answer, and scores the response along three dimensions.

The first one, faithfulness, measures the degree to which the generated answer is grounded in the retrieved context, penalising responses that introduce claims not supported by the provided material. The second, answer relevance, assesses whether the response addresses the specific question that was asked, rather than producing a technically correct but off-topic answer. The third, context relevance, evaluates whether the retrieved chunks were actually useful for answering the question, targeting the quality of the retrieval stage independently of how well the language model used the context it was given. Together, these three dimensions provide a structured qualitative assessment that the deterministic metrics cannot offer. Each dimension is scored as an integer from one to five, producing a compact, comparable output for every question in the golden dataset.

To prevent the judge from scoring entirely on subjective impression, the system provides it with quantitative signals tailored to each dimension. For faithfulness, the judge receives a similarity measure between the generated answer and the retrieved chunks, reflecting how closely the answer tracks the material it should be grounded in. For answer relevance, it receives a similarity measure between the question and the generated answer, indicating whether the response stayed

on topic. For context relevance, it receives a similarity measure between the question and the retrieved chunks, capturing whether the retrieval stage surfaced material that was actually pertinent. Each signal is accompanied by a calibration rubric that maps score ranges to expected quality levels, and the judge is instructed to use these signals as an anchor for its assessment rather than as a sole criterion, so that its scores remain grounded in measurable evidence while still reflecting the nuanced judgment that only a language model can provide. This calibration mechanism is intended to reduce the variance inherent in LLM-based evaluation and to produce scores that are more consistent across repeated runs.

3.4.4 Observability and tracing

While the metrics and the judge score the system's outputs, the observability layer serves a different purpose: it traces the internal behaviour of each query as it passes through the pipeline, recording what happened at each stage so that problems can be localised rather than merely detected. The system integrates with LangFuse [32], an open-source observability platform purpose-built for telemetry of LLM, which captures a structured record of every query, broken into spans that correspond to the main stages of the pipeline such as retrieval, reranking, context formatting, and generation. Each span records the intermediate results and metadata produced at that stage, making it possible to inspect, for example, which chunks were retrieved, how the reranker reordered them, and how long each step took. Each trace is identified by a unique identifier that persists beyond the diagnostic record itself, allowing other mechanisms to attach additional information to the same query, as described in the following subsection.

The observability layer is designed as an optional component that degrades gracefully when not configured. If the tracing platform credentials are not provided, the system continues to operate normally without recording traces, ensuring that the absence of an observability setup does not prevent the core functionality from running. This makes it possible to deploy the system in environments where external tracing infrastructure is not available, while retaining the ability to enable full diagnostic tracing when needed.

3.4.5 User feedback loop

After receiving an answer, the user can submit a star rating that captures their assessment of the response, as shown in Figure 3.6. The rating is always written to a local file regardless of whether the observability layer is active. When it is, the rating is additionally posted to the corresponding trace, grounding the qualitative signal in the full diagnostic context of that specific interaction, including the retrieved chunks, the reranking decisions, and the timing of each stage. This connection is what makes the feedback mechanism more than a detached log: a low rating can be investigated by inspecting the trace it is linked to, revealing whether the problem lay in retrieval, reranking, or generation.

Together, these mechanisms form an evaluation architecture that operates on two distinct levels: an offline loop in which the system is run against the golden dataset and its outputs are scored

by the automated metrics and the judge model, and a live level on which the observability layer and the user feedback mechanism capture diagnostic and qualitative signals from real interactions. Figure 3.4 presents this architecture, showing how the golden dataset anchors the offline evaluation and how the live mechanisms attach to the queries they observe.

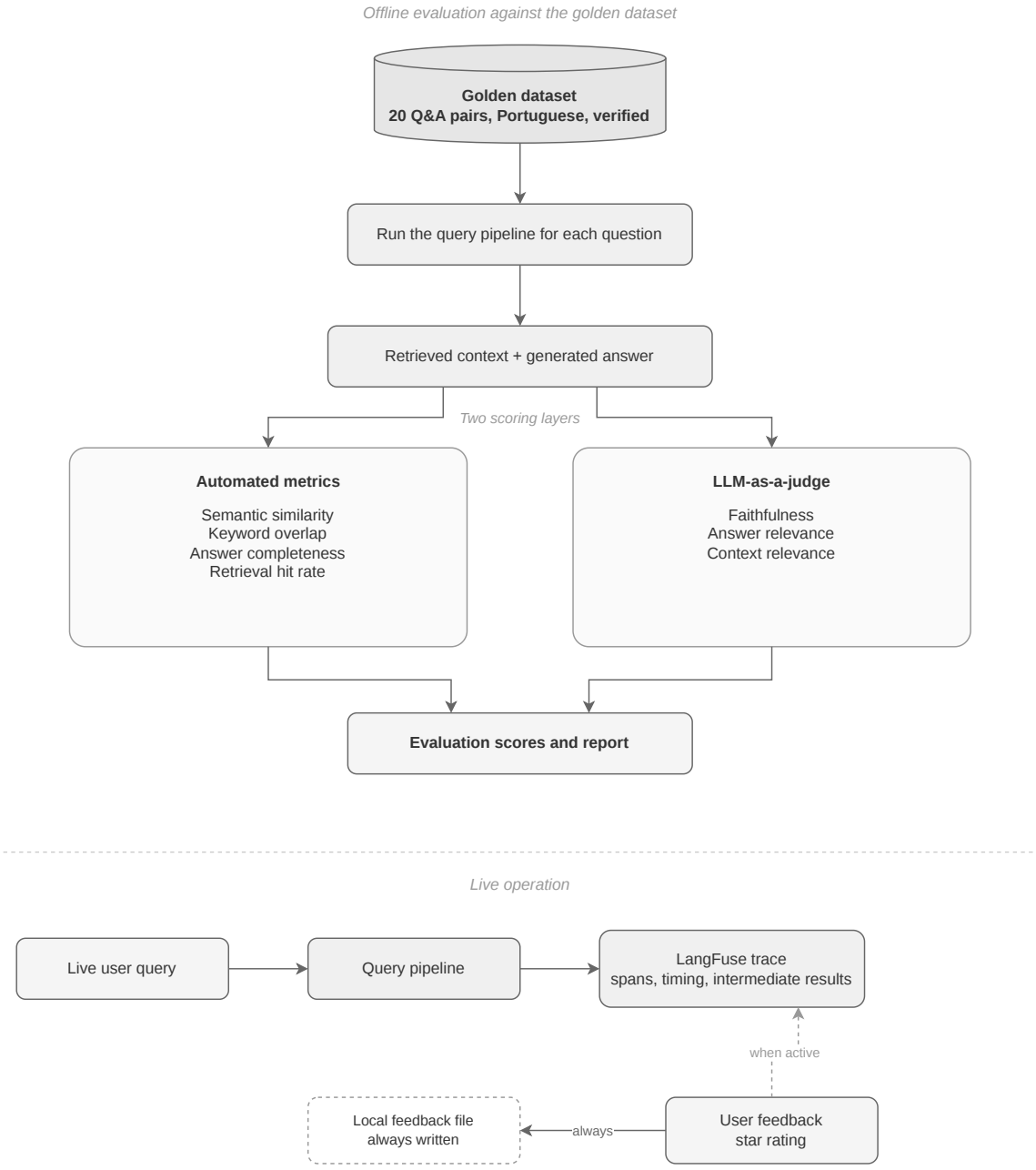

Figure 3.4: The evaluation architecture, separating the offline scoring of golden dataset runs from the live observability and feedback mechanisms.

3.4.6 DeepEval

DeepEval [33] is an open-source evaluation framework for large language model applications that formalizes many of the same evaluation concepts adopted in this dissertation into a reusable library. Rather than requiring a custom implementation for each metric, DeepEval provides pre-built metrics such as faithfulness, answer relevancy, and contextual relevancy, mirroring the three dimensions scored by the LLM-as-a-judge component. Each of these metrics is itself computed using an LLM as the evaluator, following the same underlying principle of using a language model to assess the quality of another model's output against a defined rubric.

A distinguishing feature of DeepEval is its integration with standard software testing tools, allowing evaluation metrics to be expressed as assertions within a `pytest` test suite. This positions LLM evaluation as a component of continuous integration rather than a separate offline process, enabling evaluation to run automatically whenever the system changes. DeepEval also supports synthetic dataset generation, in which a language model is used to generate question-answer pairs from a document corpus, offering a potential alternative to the manual curation process used to construct the golden dataset.

Adopting a framework such as DeepEval would trade the fine-grained control exercised in this dissertation, over the calibration scale, the rubric wording, and the judge model, for a standardized, community-maintained implementation with built-in reporting and dashboarding through its companion platform, Confident AI. This section situates DeepEval as a relevant point of comparison within the broader landscape of RAG evaluation tooling, and its use is discussed further as a direction for future work in Chapter 6.

3.5 Frontend interface

The frontend interface is the topmost layer of the architecture, through which all user interaction takes place. It is built with Gradio [34], a framework that allows a complete web interface to be constructed directly in Python without a separate frontend codebase. This choice keeps the system self-contained, since the interface, the pipeline, and the services all reside within a single project, and it allows the interface to evolve rapidly alongside the underlying system. The interface is organised as a single page that presents the conversation, the controls for submitting questions, and the supporting material that accompanies each answer.

At the top of the interface, a selector allows the user to scope a question to the documentation of a specific machine or to search across all available documents, with the available options populated automatically from the indexed knowledge base. Below it, a text field receives the user's question, and a pair of controls allow the question to be submitted or the conversation to be reset. The central element is a conversation panel that displays the full multi-turn exchange, preserving the history of questions and answers so that the dialogue can be followed and continued. Beneath the conversation, two separate galleries present any tables and figures that the answer refers to, and a chunks section lists the document passages that were retrieved to produce the answer, giving the

user visibility into the sources behind each response. A feedback control allows the user to rate the quality of the answer they received.

Several design decisions shape how these elements behave. The table and figure galleries are conditionally visible: they remain hidden unless the pipeline has actually resolved images for the current answer, so the interface never presents empty visual placeholders. The feedback control follows the same principle of progressive disclosure, remaining hidden until a question has been answered, so that it appears only when there is a response to evaluate. When a question is submitted, the input field is cleared automatically, leaving the interface ready for the next question while the conversation panel retains the history. Finally, the rating a user submits is linked internally to the trace of the specific query it concerns, so that feedback is always tied to the exact interaction it refers to without exposing any of that internal bookkeeping to the user.

The three main areas of the interface:

- **Figure 3.5:** the document selector, question input field, and conversation panel, showing the most recent question and answer.
- **Figure 3.6:** the contextual figure gallery, showing a resolved technical illustration, and the feedback control displayed after a response is produced. In this particular example, no table was shown, because no table was retrieved.
- **Figure 3.7:** the chunks section, listing the five retrieved passages used to construct the answer, each identified by its source document.

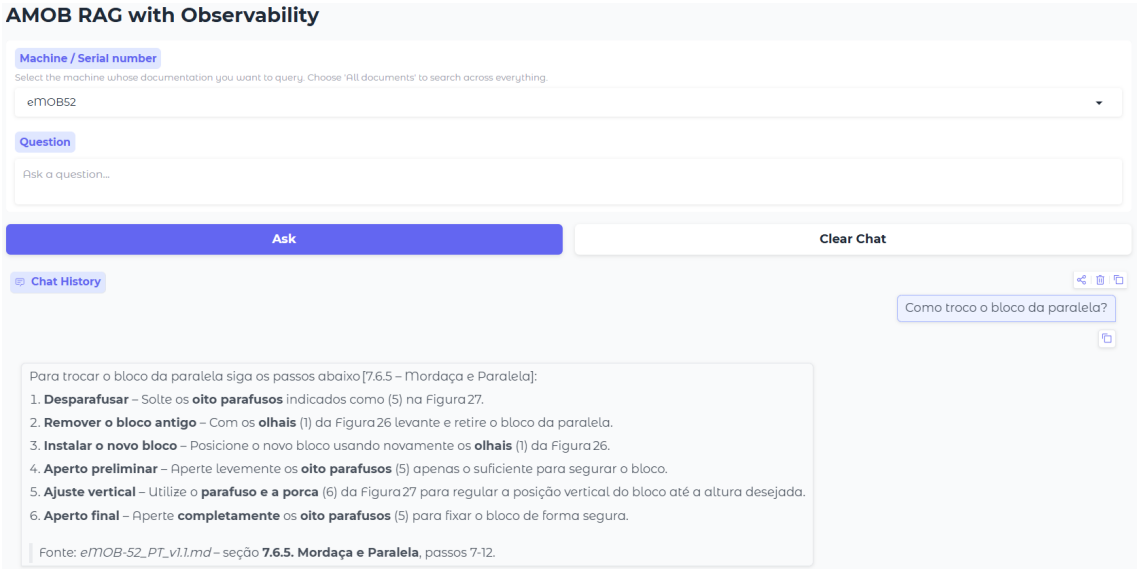

Figure 3.5: The document selector, question input field, and conversation panel.

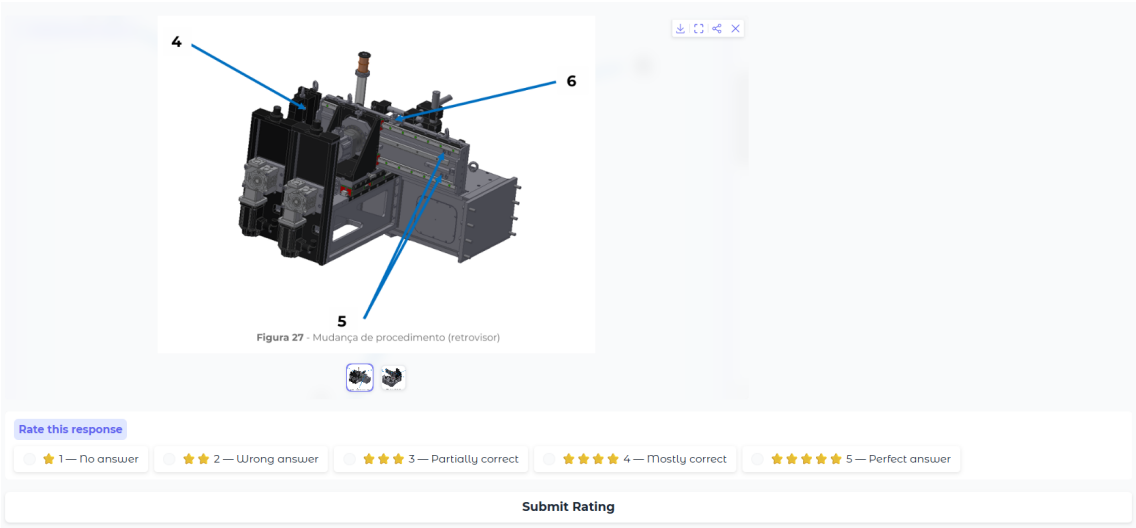

Figure 3.6: The contextual figure gallery and the feedback control displayed after a response is produced.

Figure 3.7: The chunks section, showing five retrieved passages.

Chapter 4

Implementation

This chapter describes the implementation of the retrieval-augmented generation system introduced in Chapter 3. The codebase comprises approximately 4000 lines of Python across the application, evaluation, and utility modules. Section 4.1 presents the project structure and the modularity principles that govern the codebase. Section 4.2 describes the configuration mechanism and the environment variables that control the system at runtime. Section 4.3 details the implementation of the ingestion pipeline, covering each processing stage in turn, and describes the inspection utilities that allow each stage to be observed without modifying the index. Section 4.4 covers the implementation of the query pipeline, including query preprocessing, retrieval, reranking, context assembly, and image resolution. Finally, Section 4.5 presents the evaluation framework implementation, covering the automated metrics, the evaluation runner, the LLM-as-a-judge module, and the evaluation command-line interface (CLI).

4.1 Project structure and modularity

The project is organized around a two-level Python package rooted at `src/`, complemented by `evals/` for evaluation tooling and `scripts/` for development utilities, with `app.py` at the repository root as the sole user-facing entry point. Table 4.1 summarizes the modules and subpackages that make up the codebase and their responsibilities.

Table 4.1: Module layout and responsibilities.

Module	**Responsibility**
`app.py`	User-facing entry point, Gradio interface
`src/rag.py`	Query orchestration
`src/config.py`	Configuration hub, environment variables
`src/models.py`	Shared data model (`QueryResult`)
`src/services/embeddings.py`	Embedding model initialization
`src/services/llm.py`	LLM and judge model initialization

Module	Responsibility
`src/services/vectorstore.py`	Document ingestion and vector store management
`src/services/retriever.py`	Similarity retrieval and reranking
`src/services/chunker.py`	Chunking strategy selection and block splitting
`src/services/preprocessor.py`	Markdown cleaning and chunk post-processing
`src/services/observability.py`	LangFuse tracing and callback handlers
`src/services/table_images.py`	Table and figure image resolution
`src/services/axis_synonyms.py`	Domain-specific query expansion
`evals/`	Evaluation sub-package
`scripts/`	Operational utilities

The public interface of the package is defined by `src/__init__.py`, which re-exports the symbols that both `app.py` and the `evals/` sub-package consume. Consumer code is expected to import through this facade so that neither depends on the internal module layout, meaning the service layer can be reorganized or extended without modifying any consumer of the package.

Table 4.2: Runtime concerns: public API symbols and cached factory functions.

Category	Symbol	Description
Public API	`query`	Main query entry point
	`load_documents`	Triggers document ingestion
	`list_source_documents`	Lists indexed source files
	`get_document_count`	Returns chunk count in vector store
	`clear_vectorstore`	Clears the Chroma collection
	`get_embeddings`	Returns the embedding model instance
	`QueryResult`	Shared response dataclass
Cached factories	`get_embeddings`	Embedding model
	`get_llm`	Generation LLM
	`get_judge_llm`	Judge LLM for evaluation
	`get_reranker`	CrossEncoder reranker
	`get_vectorstore`	Chroma vector store connection
	`get_langfuse_client`	LangFuse observability client
	`load_axis_synonyms`	Domain synonym table

Each factory follows the same structure: a single `@lru_cache` decorator above a function that constructs the resource from configuration constants on the first call and returns the cached instance on every subsequent call. Listing 4.1 shows this pattern for `get_embeddings`, which initializes the HuggingFace embedding model from `EMBEDDING_MODEL` and `EMBEDDING_DEVICE`.

Listing 4.1: Representative `@lru_cache` factory function.

```
@lru_cache
def get_embeddings():
```

```
    return HuggingFaceEmbeddings(
        model_name=EMBEDDING_MODEL,
        model_kwargs={"device": EMBEDDING_DEVICE}
    )
```

The dependencies between modules follow a clear directional structure with no circular imports, as illustrated in Figure 4.1. `src/config.py` acts as a shared configuration dependency for `vectorstore.py`, `rag.py`, and `axis_synonyms.py`, providing the runtime constants each of these modules requires. The ingestion and query chains are clearly separated: the ingestion chain flows from `vectorstore` through `preprocessor`, `chunker`, and `embeddings`, while the query chain flows from `rag` through `retriever`, `table_images`, `observability`, `llm`, and `vectorstore`. `retriever` additionally connects to `axis_synonyms` for domain-specific query expansion, which in turn shares a configuration dependency with `config.py`. `preprocessor.py` and `axis_synonyms.py` have no internal imports beyond their configuration dependency and can therefore be modified in complete isolation. This structure ensures that each service module can be replaced or reconfigured without cascading changes across the codebase, which is the structural property that makes the per-block controlled experiments presented later possible.

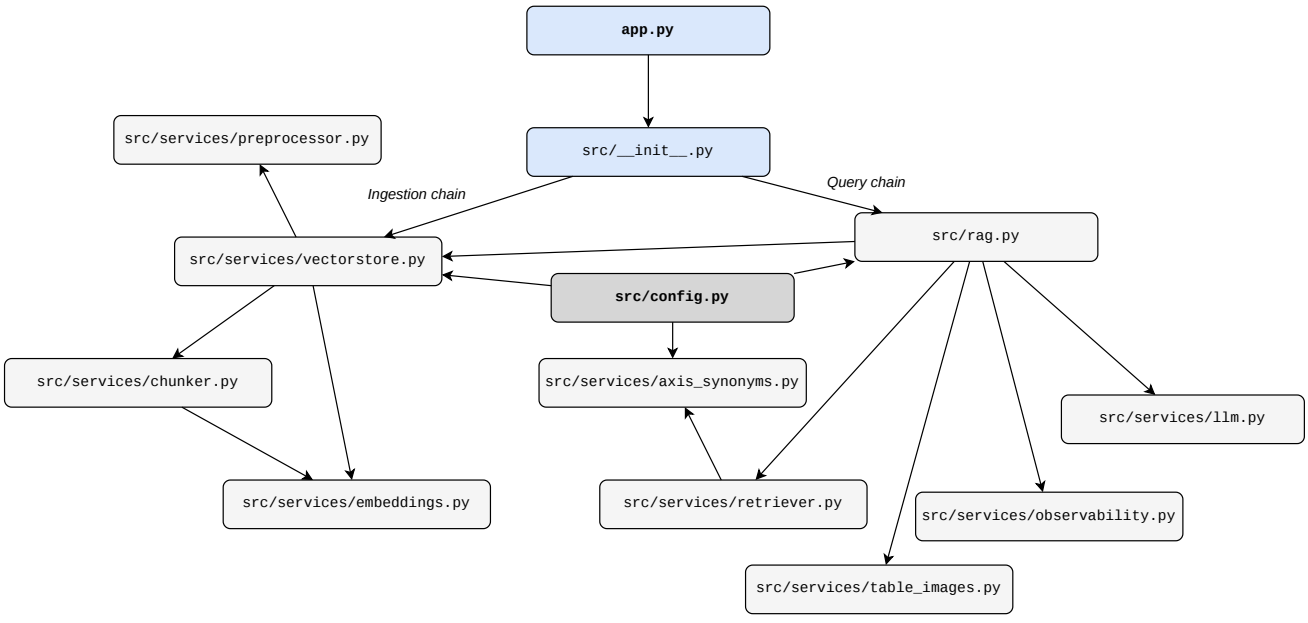

Figure 4.1: Module dependency graph of AMOB-RAG, showing the ingestion and query chains and the universal dependency on `src/config.py`.

A dedicated health-check script, `scripts/verification_check.py`, runs more than a dozen named checks spanning: six configuration parameter assertions, CrossEncoder loading, vector store population, an out-of-scope query guard, three blank-query contract checks covering return type, response text, and empty sources, a latency baseline against the golden dataset, and a

benchmark pass rate. The script exits with code 1 on any failed check, making it suitable for use as a pre-deployment regression gate.

4.2 Configuration values

All runtime parameters are defined in a single `.env` file at the repository root and loaded into `src/config.py` via `python-dotenv`. Each value is exposed as a module-level constant, which every other module imports directly. This centralizes all tunable parameters in one place and ensures that changing a value in `.env` propagates consistently across the entire system without modifying any module logic. Path constants, `DOCS_PATH`, `ALT_DOCS_PATH`, `TABLE_IMAGES_DIR`, and `FIGURE_IMAGES_DIR`, are derived programmatically from `BASE_DIR` at import time and are not overridable via `.env`.

Table 4.3 lists the environment variables grouped by concern, together with their default values and purpose.

Table 4.3: Environment variables grouped by concern.

Group	Variable	Purpose (default)
LLM	`GROQ_API_KEY`	Authentication key for the Groq API
	`LLM_MODEL`	Model used for answer generation (`openai/gpt-oss-120b`)
	`LLM_TEMPERATURE`	Sampling temperature for generation (`0.0`)
	`JUDGE_MODEL`	Model used by the LLM-as-a-judge evaluator (`llama-3.1-8b-instant`)
Embeddings	`EMBEDDING_MODEL`	HuggingFace model used for dense retrieval (`intfloat/multilingual-e5-base`)
	`EMBEDDING_DEVICE`	Torch device for embedding inference (`cpu`)
Vector store	`CHROMA_DIR`	Persistent storage directory for ChromaDB (`./data/chroma_db`)
	`CHROMA_COLLECTION`	Collection name used within ChromaDB (`documents`)
Retriever	`RETRIEVER_K`	Number of chunks retrieved by similarity search (`50`)
	`MINIMUM_SIMILARITY`	Similarity threshold below which no answer is returned (`0.10`)

Group	Variable	Purpose (default)
	PRE_RERANK_LIMIT	Maximum candidates passed to the reranker (15)
	RERANKER_TOP_N	Number of chunks kept after CrossEncoder reranking (5)
	NO_MATCH_RESPONSE	Fallback response returned when similarity is below threshold
Chunking	CHUNKING_STRATEGY	Selects SemanticChunker or RecursiveCharacterTextSplitter (semantic)
	CHUNK_SIZE	Maximum chunk size in characters for size-based chunking (1000)
	CHUNK_OVERLAP	Overlap in characters between consecutive size-based chunks (200)
	SEMANTIC_THRESHOLD_TYPE	Breakpoint detection method for semantic chunking (percentile)
	SEMANTIC_THRESHOLD_AMOUNT	Threshold value applied to the selected breakpoint method (90)
Paths	DOCS_PATH	Primary documentation directory (derived)
	ALT_DOCS_PATH	Alternative documentation directory (derived)
	TABLE_IMAGES_DIR	Directory for resolved table images (derived)
	FIGURE_IMAGES_DIR	Directory for resolved figure images (derived)
LangFuse	LANGFUSE_PUBLIC_KEY	Public key for LangFuse observability
	LANGFUSE_SECRET_KEY	Secret key for LangFuse observability
	LANGFUSE_HOST	LangFuse server endpoint (https://cloud.langfuse.com)
	LANGFUSE_ENVIRONMENT	Environment tag applied to all traces (development)

Several variables serve as direct toggles for the controlled experiments presented later. CHUNKING_STRATEGY is the primary ingestion toggle: the system is built around the semantic strategy, which activates SemanticChunker with percentile-based breakpoint detection by default (SEMANTIC_THRESHOLD_TYPE), while the size value substitutes RecursiveCharacterTextSplitter and is available as a comparison baseline. Re-indexing

after the change produces a different vector store against which the same evaluation run can be compared, provided `clear_vectorstore()` is called first to discard the existing index. On the retrieval side, `RETRIEVER_K`, `PRE_RERANK_LIMIT`, `RERANKER_TOP_N`, and `MINIMUM_SIMILARITY` are varied across runs to isolate the contribution of each retrieval stage. Because all of these are environment variables rather than hardcoded constants, each experimental condition can be reproduced exactly by restoring the corresponding `.env` file.

4.3 Ingestion pipeline implementation

Listing the exact call order across files gives a precise reference before the prose walkthrough that follows. Table 4.4 traces every step from the public entrypoint to the final persisted chunks.

Table 4.4: Ingestion pipeline call order across files.

#	Function	Location	Description
1	`load_documents()`	`vectorstore.py:71`	Public entrypoint; orchestrates the whole ingestion pipeline; short-circuits if the collection already has ids.
2	`get_vectorstore()`	`vectorstore.py:37`	Creates/returns cached Chroma collection.
3	`get_documents_path()`	`vectorstore.py:27`	Resolves `DOCS_PATH`/`ALT_DOCS_PATH`.
4	`DirectoryLoader(...).load()`	`vectorstore.py:99-106`	Loads all `*.md` files into `Document` objects.
5	`remove_table_of_contents()`	`preprocessor.py:51`	Strips leading ToC block.
6	`clean_markdown()`	`preprocessor.py:36`	Strips image placeholders/noise, collapses blank lines.
7	`MarkdownHeaderTextSplitter.split_text()`	`vectorstore.py:126`	Splits on `##` headers into section-tagged docs.
8	`merge_warnings_into_context()`	`preprocessor.py:66`	Merges "ATENÇÃO" sections into preceding doc.
9	`extract_md_tables()`	`preprocessor.py:90`	Extracts markdown tables, classifies as Type 1/2/3, returns `(clean_docs, table_chunks)`.

#	Function	Location	Description
10	`split_structural_blocks()`	`chunker.py:51`	Separates paragraph runs from atomic list/numbered-procedure blocks, tags `block_index`.
11	`get_chunker()`	`chunker.py:16`	Returns `SemanticChunker` or `RecursiveCharacterTextSplitter` per `CHUNKING_STRATEGY`.
12	`chunker.split_documents(paragraph_blocks)`	`vectorstore.py:144`	Splits only paragraph blocks (never atomic lists); assigns `sub_index`.
13	merge and sort	`vectorstore.py:155–159`	Combines semantic chunks, atomic blocks, and table chunks, sorts by `(block_index, sub_index)`.
14	`merge_small_chunks(min_chars=150)`	`preprocessor.py:335`	Merges small chunks (e.g. captions) into same-section neighbors.
15	header-prepend loop	`vectorstore.py:165–179`	Tags machine/type metadata, prepends section header to `page_content`.
16	`Chroma.add_documents(chunks)`	`vectorstore.py:182`	Persists chunks (with embeddings) to ChromaDB; returns chunk count.

The ingestion pipeline is implemented inside `load_documents()` in `src/services/vectorstore.py`. Before executing any stage, the function checks whether the ChromaDB collection already contains documents and returns immediately if it does, making repeated calls safe across restarts. This idempotency mechanism has one important consequence: switching `CHUNKING_STRATEGY` without first calling `clear_vectorstore()` silently leaves the old index in place, since the non-empty collection causes the function to exit before any new chunking logic runs.

4.3.1 Document preprocessing

The preprocessing stages described here were developed around the specific structure and noise patterns of the eMOB-52 machine documentation. The cleaning decisions, such as stripping table-of-contents blocks, removing HTML image placeholders, and merging warning sections, reflect the most common artefacts observed in that corpus. A different document set would likely require different preprocessing logic, and the pipeline should be treated as document-specific rather than general-purpose. The pipeline expects documents to be provided in Markdown format; the conversion from the original PDF sources to Markdown is performed upstream using Docling, as described in Section 3.2.1, and falls outside the project codebase.

The first two stages normalize the raw markdown before any structural splitting takes place. `remove_table_of_contents()` scans the document for the first numbered section header matching a regular expression (REGEX) pattern and discards all content before it, preventing manually written tables of contents from being indexed as retrievable text:

```
r'^#{1,2}\s+\d+[.\s]'
```

`clean_markdown()` then removes `<!-- image -->` placeholders left by the document export process, drops lines containing only a single digit, and collapses runs of three or more consecutive blank lines into two, producing clean Markdown ready for structural splitting.

After header splitting via `MarkdownHeaderTextSplitter`, `merge_warnings_into_context()` identifies any section whose header contains `ATENÇÃO` and appends its body to the preceding chunk with the prefix `"ATENÇÃO: "`, ensuring that safety warnings are never stored as isolated chunks detached from the context they qualify.

4.3.2 Table extraction

`extract_md_tables()` detects pipe-delimited Markdown tables using the following REGEX pattern and removes them from the text chunk before further splitting:

```
table_pattern = re.compile(r'((?:^\|.+\|[␣\t]*\n?)+)', re.MULTILINE)
```

The `re.MULTILINE` flag is required so that `^` anchors to the start of each line rather than the start of the entire string, allowing the pattern to match tables that appear anywhere in the document. Each detected table is classified into one of three types and converted into one or more `Document` objects carrying the parent section, an optional caption, and a `type=table_row` metadata flag.

- **Type 1 (description-value):** tables whose headers indicate a description-to-value relationship, such as component names paired with physical measurements. All rows are merged into a single natural-language sentence using a Portuguese template of the form `{descrição} pesa {valor} {unidade}.` As an example, a row with `Consola CNC | 50` from a table with headers `Descrição | Peso (daN)` produces the sentence *Consola CNC pesa 50 daN*. Figure 4.2 shows an example of a Type 1 table as it appears in the source document.

- **Type 2 (merged-cell):** tables with a high proportion of empty or repeated first-column values, or a detectable unit identifier in the header. Rows are consolidated into a key-value text block, with sub-headers detected and used as column labels where present. Figure 4.3 shows an example of a Type 2 table as it appears in the source document.

- **Type 3 (generic):** all remaining tables. One `Document` is emitted per data row in `HEADER: value` format.

Descrição	Peso (daN)
Máquina (padrão) (sem console CNC)	4500
Esgrima (se encomendado)	250
Consola CNC	50

Figure 4.2: Example of a Type 1 description-value table as it appears in the source document.

Descrição		Unidade de medida	Valor
Caraterísticas Físicas	Peso total da máquina com CNC (padrão)	daN	4500
	Dimensões (l x d x h)		Consulte o layout
	Cor	RAL	7005-7040
Condições de ambiente	Temperatura ambiente	°C	+5 / +40
	Temperatura de armazenamento	°C	-5 / +40
	Altitude de instalação	m	<1000 a.s.l. (sem desvalorização) 1000-5000 a.s.l. (com desvalorização)
	Humidade relativa	%	5 - 75 (máx. durante 24 h)
Sistema elétrico	Tensão de alimentação	V	400 ±10% (-15% máx. durante 1 minuto)
	Tipo de potência	-	Trifásico sem neutro
	Frequência	Hz	47-63
	Potência instalada	kVA	22
	Corrente nominal	Um	31
	Tensão auxiliar	V	24 dc
	Tensão de alimentação CNC	V	24 dc
	Grau de proteção do compartimento	-	IP55
Unidade de lubrificação de mandril mineral	Capacidade do tanque	Eu	18
	Caudal	l/min	De acordo com a configuração
	Consumo médio	l/min	De acordo com a configuração
Sistema Pneumático	Pressão de trabalho	bar	6
	Tratamento do ar		Unidade FR

Figure 4.3: Example of a Type 2 merged-cell table as it appears in the source document.

Caption lookup follows a two-pass strategy. The code first searches the 200 characters immediately preceding the table for a `Tabela N` caption. If none is found, it walks back through all captions seen so far in the document and takes the most recent one that appeared after the previous table. Tables for which neither pass finds a caption are assigned a synthetic identifier by a module-level counter, producing labels `S1`, `S2`, and so on. The resolved caption is parsed by

`extract_table_id()` to produce a normalised identifier such as `tabela_3`, which is stored as `table_id` metadata on each chunk. This identifier is the key used later in the query pipeline to locate the corresponding table image on disk via `resolve_table_images()`, which may then be included alongside the answer in the frontend. Rows that fail an internal low-value filter are silently dropped before the documents are emitted.

4.3.3 Structural block splitting

`split_structural_blocks()` scans each section chunk line by line and separates content into two block types: paragraph blocks, which are passed to the semantic chunker in the next stage, and atomic list blocks, which bypass chunking entirely and are stored as single `Document` objects. This distinction preserves the sequential structure of procedural steps, such as installation sequences or adjustment procedures, which would lose meaning if split mid-list by a chunker.

Numbered lists are detected by the following the REGEX:

```
r'^\s*\d+[\.\)]\s*'
```

and bullet lists by:

```
r'^\s*[-*]\s+'
```

Each block receives a monotonically increasing `block_index`; atomic blocks also receive `sub_index=0`. To preserve the context that introduces a list, `_extract_intro_sentence()` walks the lines of the preceding paragraph in reverse and prepends one line to the list chunk. The priority rule is as follows: if any line ends with a colon, that line is returned immediately, since a colon signals an explicit list introduction. For example, given a paragraph ending with `"To adjust the clamping pressure, proceed as follows:"`, that line would be prepended regardless of what came before it. If no colon-terminated line exists, the function falls back to the last non-empty line of the paragraph instead, for example `"The following components must be inspected before each cycle."` This ensures that even when the source document does not use a colon to introduce a list, the list chunk still carries enough surrounding context for retrieval to work correctly.

Atomic list blocks are merged back into the full chunk list at the merge and sort step of Table 4.4, sorted alongside the semantic chunks by `(block_index, sub_index)` to restore document order.

4.3.4 Semantic chunking

Paragraph blocks are split by `get_chunker()`, which reads `CHUNKING_STRATEGY` from the configuration and instantiates the appropriate splitter. As noted in Section 4.2, the system is built around the `semantic` strategy. Listing 4.2 shows the instantiation of `SemanticChunker` with its two configurable threshold parameters.

Listing 4.2: Instantiation of `SemanticChunker` inside `get_chunker()`.

```
SemanticChunker(
    embeddings,
    breakpoint_threshold_type=SEMANTIC_THRESHOLD_TYPE,
    breakpoint_threshold_amount=SEMANTIC_THRESHOLD_AMOUNT
)
```

The `percentile` threshold type instructs the chunker to compute the distribution of cosine distances between consecutive sentence embeddings and place a breakpoint wherever the distance exceeds the 90th percentile of that distribution. `SemanticChunker` applies cosine distance internally by default; no explicit metric is configured in `get_chunker()`. The similarity computation therefore depends entirely on the embedding model passed to it, which in this system is `intfloat/multilingual-e5-base` via `get_embeddings()`, meaning the chunking boundaries reflect the semantic space of that specific multilingual model. This means the number and size of chunks adapts to the semantic density of each section rather than being fixed by a character count. Resulting semantic chunks receive sequential `sub_index` values to preserve their within-block order before being merged with the atomic and table chunks at the merge and sort step of Table 4.4.

After merging and sorting, `merge_small_chunks()` walks the sorted list and merges any non-table chunk whose content is shorter than 150 characters into an adjacent chunk from the same section, preferring a backward merge and falling back to a forward merge. Table rows are never merged. The final stage prepends the section header to each chunk whose content does not already contain it, and enriches each chunk with `machine` metadata derived from the subdirectory name of the source file.

4.3.5 Development utilities

Two scripts support inspection of the ingestion pipeline without modifying the live index. `scripts/semantic_chunks.py` runs `SemanticChunker` directly on the raw documents and prints chunk previews to standard output, accepting `--threshold`, `--threshold-type`, `--limit`, and `--full` flags. It is the recommended tool for tuning `SEMANTIC_THRESHOLD_AMOUNT` before committing to a full re-index. `scripts/inspect_chunks.py` connects directly to the live ChromaDB collection and reads the documents and metadata that were actually stored after indexing, accepting `--limit`, `--full`, `--save`, and `--output` flags for controlling output scope and persistence.

The most diagnostic invocation is the following:

```
python scripts/inspect_chunks.py --full --save
```

This command dumps every chunk stored in the vector store to disk in full, making it straightforward to manually audit whether the ingestion pipeline is correctly curating the source documents. This output allows the developer to verify, for instance, that table rows were extracted as individual documents, that section headers were correctly prepended to each chunk, and that no

spurious content such as table-of-contents entries or HTML artefacts survived the cleaning step. Because the inspection targets the vector store directly rather than re-running the pipeline, it reflects exactly what the retriever will see at query time, making it the definitive tool for diagnosing retrieval quality issues at their root.

Together these two utilities cover the two most common diagnostic needs: previewing what the chunker would produce before indexing, and verifying what the vector store actually contains after indexing.

4.4 Query pipeline implementation

Listing the exact call order across files gives a precise reference before the prose walkthrough that follows. Table 4.5 traces every step from the orchestration entrypoint to the final returned result.

Table 4.5: Query pipeline call order across files.

#	Function	Location	Description
1	`query()`	`rag.py:33`	Main orchestration entrypoint.
2	`get_vectorstore()`	`vectorstore.py:37`	Returns cached Chroma collection.
3	`retrieve()`	`retriever.py:123`	Orchestrates normalization, synonym expansion, similarity search, threshold gating.
4	`_normalize_question()`	`retriever.py:88`	Lemmatizes via `VERB_LEMMAS`, appended to question if changed.
5	`expand_synonyms()`	`axis_synonyms.py:51`	Appends matching axis synonyms.
6	`load_axis_synonyms()`	`axis_synonyms.py:14`	Loads `Nomes_dos_eixos.xlsx` lookup (`lru_cached`).
7	`Chroma.similarity_search_with_score()`	`retriever.py:136`	Applies metadata filter, returns `(Document, distance)` pairs.
8	threshold gate	`retriever.py:138-145`	Computes similarity; below `MINIMUM_SIMILARITY` → `([], True)` → `NO_MATCH_RESPONSE`.
9	pre-rerank sort/truncate	`rag.py:112-113`	Sorts by distance, slices to `PRE_RERANK_LIMIT`.

#	Function	Location	Description
10	`rerank_documents()`	`retriever.py:102`	CrossEncoder (`BAAI/bge-reranker-v2-m3`) scoring, `TABLE_PENALTY` applied via `_is_table_query()`.
11	slice to top-N	`rag.py:120`	`docs[:RERANKER_TOP_N]`.
12	table-id extraction regex	`rag.py:124-140`	`re.finditer(r"Tabela\s+(\d+)")` over top docs, selects `table_id`.
13	`resolve_table_images()`	`table_images.py:14`	Scoped lookup under manual subfolder, root fallback.
14	`format_docs()`	`retriever.py:149`	Builds filename-tagged context string.
15	`get_prompt_template()`	`rag.py:25`	Builds `ChatPromptTemplate` with `MessagesPlaceholder`.
16	chat-history mapping	`rag.py:180-185`	Role dicts → `HumanMessage`/`AIMessage`.
17	`chain.invoke()`	`rag.py:173,187-190`	Chain composed as `prompt | llm | StrOutputParser()`; produces final answer string.
18	figure-id extraction regex	`rag.py:195-202`	`re.findall` over answer, fallback scan over `docs[:3]`.
19	`resolve_figure_images()`	`table_images.py:58`	Scoped glob `figura_{id}_*`, root fallback.
20	`format_references()`	`rag.py:224-236`	Called at end of `rag.py`; returns final `QueryResult`.

The query pipeline is orchestrated by `query()` in `src/rag.py`, which wires together query preprocessing, similarity retrieval, threshold gating, CrossEncoder reranking, context assembly, LLM generation, and image resolution into a single call that returns a `QueryResult`. The preprocessing and retrieval logic is implemented in `src/services/retriever.py`, axis-synonym expansion in `src/services/axis_synonyms.py`, and image resolution in `src/services/table_images.py`. The subsections that follow cover each stage in the order it executes at query time.

4.4.1 Query preprocessing: lemmatization and synonym expansion

Before retrieval, the user's question passes through two sequential preprocessing steps implemented in `src/services/retriever.py`: verb lemmatization and axis-synonym expansion.

The goal is to bridge the lexical gap between the phrasing a user chooses and the terminology used in the indexed documentation.

The first step is handled by `_normalize_question()`, which lowercases the question, strips punctuation, and replaces inflected Portuguese verb forms with their infinitive using the `VERB_LEMMAS` dictionary. Listing 4.3 shows the function body.

Listing 4.3: Query normalization function.

```
def _normalize_question(question: str) -> str:
    q = question.lower().strip()
    q = re.sub(r"[?!.]+", "␣", q)
    q = re.sub(r"[^\w\sçãõâêîôûáéíóúàèìòù]+", "␣", q)
    for form, lemma in VERB_LEMMAS.items():
        q = re.sub(rf"\b{re.escape(form)}\b", lemma, q)
    q = re.sub(r"\s+", "␣", q).strip()
    return q
```

`VERB_LEMMAS` maps inflected forms of eleven Portuguese verbs to their infinitives: *trocar*, *mudar*, *ajustar*, *operar*, *ligar*, *desligar*, *instalar*, *abrir*, *fechar*, *pressionar*, and *girar*. These verbs were selected based on their frequency in the queries observed during development against the eMOB-52 documentation. Each verb is covered by its first-person singular, third-person singular, past, third-person plural, first-person plural, and infinitive forms, giving a dictionary of 65 entries.

The normalized form is then combined with the original question into a single enriched query string, but only when normalization actually changed something. This concatenation is an intentional query expansion technique: the embedding model receives both the natural-language phrasing and the lemmatized form in a single input, which improves recall against chunks that were indexed using normalized terminology. When normalization produces no change, the original question is used as-is, avoiding redundant repetition. The result is then passed through `expand_synonyms()`, which appends any known domain-specific synonym variants before the query is submitted to retrieval:

```
expanded_question = _normalize_question(question)
if expanded_question and expanded_question != question.lower():
    expanded_question = f"{question}␣{expanded_question}"
else:
    expanded_question = question
expanded_question = expand_synonyms(expanded_question)
```

To illustrate this behavior concretely, consider the question *Qual é a pressão máxima do óleo (bar)?*. `_normalize_question()` lowercases it, strips the `"?"` and parentheses, and collapses whitespace, producing `"qual é a pressão máxima do óleo bar"`. Since this normalized form differs from the lowercased original, the concatenation branch is taken, yielding the enriched query:

```
Qual é a pressão máxima do óleo (bar)? qual e a pressao maxima do oleo bar
```

Note that the original and normalized forms overlap substantially here, since punctuation removal was the only change, so most content words are repeated verbatim rather than lemmatized to a distinct form. This duplication is an accepted side effect of the concatenation strategy: because `_normalize_question()` is applied uniformly regardless of how much a given question actually benefits from normalization, low-impact cases such as punctuation-only changes still trigger repetition, while high-impact cases involving verb inflection, for example *abro* normalized to *abrir*, gain the intended lexical bridging. The system favors recall over query concision, accepting redundant tokens in the embedding input as a trade-off for consistently surfacing chunks indexed with normalized terminology. The result is a single query string submitted to one retrieval call, not two separate searches. The second step is axis-synonym expansion, implemented in `src/services/axis_synonyms.py`.

4.4.2 Retrieval and similarity threshold gating

Retrieval is handled by `retrieve()` in `src/services/retriever.py`, which takes the original question, the vector store instance, and an optional `source_filter` string, and returns a `(docs_with_scores, below_threshold)` tuple. The function applies the full query preprocessing pipeline described in Section 4.4.1 internally, so the caller in `rag.py` passes the raw question and receives preprocessed results without managing the expansion steps directly.

If a `source_filter` is provided, it is passed unmodified as a string into a ChromaDB metadata filter that restricts retrieval to chunks whose `machine` field matches exactly:

```
if source_filter:
    search_kwargs["filter"] = {"machine": {"$eq": source_filter}}
```

This corresponds to the document-scoping selector in the frontend interface, where the user can restrict a query to the documentation of a specific machine. When no filter is provided, retrieval runs across the full collection. The filtering mechanism becomes particularly relevant as the knowledge base grows: in a deployment with hundreds of indexed manuals, restricting retrieval to a specific machine prevents the similarity search from scanning the entire collection, avoiding both performance degradation and the risk of retrieving contextually irrelevant chunks from unrelated documents.

Similarity search is performed by `similarity_search_with_score()`, which returns up to `RETRIEVER_K` chunks together with their ChromaDB cosine distances. Because ChromaDB returns distances rather than similarities, the threshold comparison is made against the converted value of the top-ranked result:

```
best_similarity = 1 - docs_with_scores[0][1]
if best_similarity < MINIMUM_SIMILARITY:
    return [], True
```

Only the top hit is used for the threshold decision. If `best_similarity` falls below `MINIMUM_SIMILARITY` (default `0.10`), the function returns an empty list and `True` as the

below-threshold flag, discarding all retrieved documents. If the collection returns no results at all, `best_similarity` is set to `0`, which also triggers the below-threshold path.

Back in `rag.py`, the return value is unpacked and the flag is checked after retrieval:

```
docs_with_scores, below_threshold = retrieve(question, vs, source_filter=
    source_filter)
```

If `below_threshold` is `True`, `query()` returns a `QueryResult` with `answer` set to `NO_MATCH_RESPONSE`, empty references, and an empty sources list, terminating the pipeline before any reranking or generation takes place. This gate ensures that the LLM is never invoked on a question the retriever cannot ground in the documentation.

4.4.3 CrossEncoder reranking and table penalty

Before reranking, the `RETRIEVER_K` chunks returned by the vector store are sorted by cosine distance in ascending order and sliced to the top `PRE_RERANK_LIMIT` candidates:

```
docs_with_scores.sort(key=lambda x: x[1])
docs_with_scores = docs_with_scores[:PRE_RERANK_LIMIT]
```

ChromaDB returns `(doc, distance)` pairs, so a lower distance corresponds to a higher similarity. The explicit re-sort guarantees ascending order by distance since ChromaDB does not contractually guarantee ordering across all query paths. The design rationale for retrieving 50 chunks before slicing to 15 is recall: vector similarity alone can miss relevant chunks due to bi-encoder limitations and paraphrase mismatch, so a wide initial net is cast and the more expensive CrossEncoder is then applied only to the shortlist, trading retrieval breadth for reranking precision.

These candidates are then passed to `rerank_documents()` in `src/services/retriever.py`, which scores each chunk against the query using a CrossEncoder and returns all documents sorted by score in descending order. The `RERANKER_TOP_N` cap is then applied by the caller in `rag.py`, keeping only the highest-scoring chunks for context assembly.

The CrossEncoder is instantiated via `get_reranker()`, decorated with `@lru_cache` so the model is loaded only once per process:

```
@lru_cache
def get_reranker():
    return CrossEncoder("BAAI/bge-reranker-v2-m3")
```

The model `BAAI/bge-reranker-v2-m3` is a multilingual cross-encoder that scores query-document pairs directly, unlike the bi-encoder used for retrieval, which encodes query and document independently. This makes the CrossEncoder more accurate but also more expensive, which is why it operates only on the pre-filtered `PRE_RERANK_LIMIT` candidates rather than the full `RETRIEVER_K` retrieved set.

Scoring is performed over pairs of the normalized question and each chunk's page content:

```
pairs = [(normalized_question, doc.page_content) for doc in docs]
scores = reranker.predict(pairs)
```

A table penalty heuristic is applied before sorting. When the query does not mention any of the table-related keywords in `TABLE_KEYWORDS`:

```
TABLE_KEYWORDS = {"tabela", "manutenção", "plano", "intervalo",
                  "periodicidade", "frequência", "lubrificação"}
```

Table chunks are penalized by subtracting `TABLE_PENALTY = 0.2` from their CrossEncoder score:

```
penalize = not _is_table_query(question)
for doc, score in zip(docs, scores):
    if penalize and doc.metadata.get("type") == "table_row":
        score -= TABLE_PENALTY
```

The rationale is that table rows, which are stored in `HEADER: value` format as described in Section 4.3.2, are less useful as context for free-text questions than for explicit table lookups. When the query does contain a table keyword, the penalty is lifted and table chunks compete on equal terms with prose chunks. The keyword list was assembled from table-related terms that commonly appear in maintenance and specification queries against the eMOB-52 documentation, following the same corpus-driven approach applied to `VERB_LEMMAS`.

4.4.4 Context assembly and prompt construction

Once reranking is complete, the top `RERANKER_TOP_N` documents are formatted into a single context string by `format_docs()` in `src/services/retriever.py`. Each document is rendered as a filename-tagged line, and chunks are joined by blank lines:

```
def format_docs(docs: list) -> str:
    return "\n\n".join([
        f"[{Path(d.metadata.get('source', '')).name}]: {d.page_content}"
        for d in docs
    ])
```

The filename tag gives the LLM explicit source attribution for each chunk, allowing it to cite the originating document in its answer. The resulting string is passed as the `context` variable to the prompt template.

The prompt template is constructed by `get_prompt_template()` and follows a three-message structure:

```
def get_prompt_template():
    return ChatPromptTemplate.from_messages([
        ("system", RAG_SYSTEM_PROMPT),
        MessagesPlaceholder(variable_name="chat_history", optional=True),
```

```
        ("human", "{question}"),
    ])
```

The system message carries `RAG_SYSTEM_PROMPT`, defined in `src/config.py`:

```
RAG_SYSTEM_PROMPT = """Answer based on the context below. If you can't
    find the answer, say so.

Context:
{context}

Be concise and cite sources when relevant."""
```

The `{context}` variable is interpolated inside the system prompt itself, so the retrieved chunks are injected at the system level rather than in the human turn. This instructs the model to answer based on the provided context, acknowledge when an answer cannot be found, be concise, and cite sources. The `MessagesPlaceholder` with `optional=True` inserts the conversation history between the system message and the human question when present, and is silently omitted when the conversation has no prior turns.

Multi-turn chat history is constructed by mapping role dictionaries from the frontend into LangChain message objects, converting `"user"` entries to `HumanMessage` and `"assistant"` entries to `AIMessage`, before the chain is invoked. It is important to note that `chat_history` provides conversational coherence across turns, while `context` is always freshly retrieved for the current question only. Prior turns do not trigger their own retrieval pass, so the model can refer back to what was said earlier in the session but cannot access retrieved chunks from previous turns unless those same chunks are retrieved again for the current question.

The chain is composed using LangChain's pipe operator (`|`), which overloads Python's `|` symbol to connect pipeline components in sequence, passing the output of each component as the input to the next. The chain is invoked with the three payload keys the template expects:

```
chain = get_prompt_template() | get_llm() | StrOutputParser()

answer = chain.invoke(
    {"context": context, "question": question, "chat_history":
        history_messages},
    config={"callbacks": callbacks},
)
```

The `callbacks` list carries the LangFuse handler when observability is configured, or is empty otherwise, so the chain composition is identical regardless of whether tracing is active. The `StrOutputParser()` at the end of the chain extracts the raw string from the LLM response object, which is the value assigned to `answer` and subsequently processed for image resolution before being returned as part of the `QueryResult`.

4.4.5 Image resolution

After the answer is generated, `query()` in `src/rag.py` resolves any table and figure images that should accompany the response. The two resolution paths are independent and handled sequentially.

The table identifier is selected in two passes. The first pass scans the top three retrieved chunks for any mention of a table number using a case-insensitive pattern, building a set of mentioned identifiers:

```
mentioned_ids = set()
for doc in docs[:3]:
    for m in re.finditer(r"Tabela\s+(\d+)", doc.page_content, re.
        IGNORECASE):
        mentioned_ids.add(m.group(1))
```

The second pass walks the pre-rerank similarity-sorted list, capped at `PRE_RERANK_LIMIT`, and selects the first chunk whose `type` metadata is `table_row` and whose `table_id` is either in `mentioned_ids` or no mentioned IDs were found. This means a table chunk pushed out of the top `RERANKER_TOP_N` by the reranker can still supply a `table_id` for image resolution. A consequence of this design is that the resolved table image may occasionally not correspond directly to the content of the answer, since the supplying chunk was not among those used for generation. In practice this is an acceptable trade-off: the user can quickly identify a mismatch, and the benefit of surfacing a potentially relevant table outweighs the cost of an occasional false positive. The selection logic is as follows:

```
table_id = None
for doc, _ in docs_with_scores:
    if doc.metadata.get("type") == "table_row":
        candidate_id = doc.metadata.get("table_id", "")
        if candidate_id and (not mentioned_ids or candidate_id in
            mentioned_ids):
            table_id = candidate_id
            break
```

If no `table_row` chunk is found, a REGEX fallback scans the top three chunks directly for a `Tabela N` pattern. Once a `table_id` is resolved, the source document path is extracted from the matching chunk's metadata and passed to `resolve_table_images()` together with the identifier.

`resolve_table_images()` in `src/services/table_images.py` searches for image files in a manual-specific subfolder first, derived from the source path stem via `_manual_stem()`, then falls back to the root `TABLE_IMAGES_DIR`. Within each candidate directory it attempts a multi-part loop, looking for files named `tabela_{id}_{part}{ext}` with `part` incrementing from 1, to handle tables whose image was split across multiple files. If the multi-part loop finds nothing, it falls back to a single-file lookup for `tabela_{id}{ext}`. Supported extensions are `.png`, `.jpg`, `.jpeg`, and `.webp`.

Figure identifiers are extracted from the generated answer using an order-preserving deduplication pattern:

```
figure_ids = list(dict.fromkeys(re.findall(r"[Ff]igura\s+(\d+)", answer)))
```

If the answer contains no figure references, the same pattern is applied as a fallback to the top three retrieved chunks, stopping at the first chunk that yields at least one identifier. For each resolved identifier, `resolve_figure_images()` is called with the source path of the first retrieved document. It searches the manual-specific subfolder first, then the root `FIGURE_IMAGES_DIR`, globbing for files matching `figura_{id}_*{ext}`. Unlike `resolve_table_images()`, there is no single-file fallback: a figure file must carry a suffix after the identifier to be matched. The resolution loop breaks after the first figure identifier that returns at least one image, so at most one figure's images are attached to any given answer, preventing the system to hallucinate content.

4.5 Evaluation framework implementation

The evaluation framework implements the architecture described in Section 3.4 across four components. The automated metrics module computes four complementary scores for each question in the golden dataset and applies a pass/fail rule based on three of them. The evaluation runner loads the dataset, iterates over questions, aggregates results into a summary, and writes timestamped reports. The LLM-as-a-judge module scores each answer independently across three qualitative dimensions using a separate judge model. Finally, two command-line interfaces expose the metric-based and judge-based evaluations as standalone processes with configurable flags and CI-compatible exit codes.

4.5.1 Automated metrics

The automated metrics module is implemented in `evals/metrics.py` and provides four functions that score a generated answer against an expected answer from the golden dataset. Each function operates independently and can be called in isolation.

Semantic similarity computes the cosine similarity between the embeddings of the generated and expected answers using the same embedding model as the query pipeline. Both answers are embedded via `embeddings_model.embed_query()` and compared using `sklearn.metrics.pairwise.cosine_similarity`. The score ranges from 0 to 1, where 1 indicates identical semantic content.

Keyword overlap measures the Jaccard similarity between the keyword sets of the generated and expected answers. Before comparison, both answers are passed through a normalization step that removes structural content introduced by the language model at generation time: inline citations, Markdown formatting, list markers, framing intro phrases, and reference blocks. Tables are flattened by discarding header rows and reducing each data row to a plain line of cell content, ensuring that column labels do not inflate the token union without contributing to the intersection.

After normalization, each answer is lowercased, stripped of punctuation, and filtered against a hardcoded bilingual stop-word set covering common English and Portuguese function words. For example, *pressão* is normalized to *pressao* so that accented and unaccented forms match. Tokens are then stemmed using the Removedor de Sufixos da Língua Portuguesa (RSLP) algorithm, the standard stemmer for Portuguese, which collapses inflected word forms to a common stem so that *ligue*, *ligar*, and *ligado* are all treated as the same term. The Jaccard score is defined as:

$$J(A,B) = \frac{|A \cap B|}{|A \cup B|} \tag{4.1}$$

where A and B are the keyword sets of the generated and expected answers, respectively. The score returns 0.0 if either set is empty after filtering.

Answer completeness measures how many key points from the expected answer are covered by the generated answer. For each point in the expected answer, key terms of four or more characters are extracted and a point is considered matched if the generated answer contains at least 50% of those terms. The function returns a dictionary with the overall score, matched count, missed count, and the matched and missed point texts.

Retrieval accuracy compares the set of retrieved source filenames against the expected source documents from the golden dataset entry, computing precision, recall, and a binary hit rate. The hit rate is 1.0 if at least one expected source was retrieved and 0.0 otherwise.

Table 4.6 summarizes the four metrics, their computation method, pass threshold, and whether they participate in the pass/fail gate.

Table 4.6: Summary of automated evaluation metrics.

Metric	Method	Pass threshold	In gate
Semantic similarity	Cosine similarity over embeddings	≥ 0.7	Yes
Keyword overlap	Jaccard over normalized keyword sets	≥ 0.3	Yes
Answer completeness	Key-term coverage at 50% match	—	No
Retrieval accuracy	Binary hit rate over source filenames	> 0	Yes

The pass/fail decision for each question is applied in `evaluate_single()` in `evals/evaluate.py`:

```
passed = (
    sem_sim >= similarity_threshold and  # default 0.7
    kw_overlap >= keyword_threshold and  # default 0.3
    retrieval["hit_rate"] > 0
)
```

Answer completeness is computed and included in the report but is not part of the pass/fail gate. When a question fails, `generate_markdown_report()` appends one or more diagnostic labels to the failure entry: `"Low similarity"` when semantic similarity falls below the threshold, `"Low keyword"` when keyword overlap falls below the threshold, and `"Retrieval`

`miss"` when the hit rate is zero. These labels allow failed questions to be triaged quickly without inspecting the full output.

After each question is evaluated, `evaluate_single()` posts five scores back to LangFuse when `result.trace_id` is present, meaning the query was executed with tracing enabled: `semantic_similarity`, `keyword_overlap`, `retrieval_hit_rate`, `completeness`, and `passed`. Any error at the score-posting level is swallowed silently so that evaluation continues normally regardless of LangFuse availability.

4.5.2 Evaluation runner and report generation

The evaluation runner is implemented in `evals/evaluate.py` and orchestrates the full evaluation pass over the golden dataset. The entry point is `run_evaluation()`, which accepts a dataset path, an optional output path, the two pass/fail thresholds, and a verbosity flag. On invocation, it first calls `load_documents()` to ensure the vector store is populated, then loads the golden dataset from the JSON file and retrieves the embedding model via `get_embeddings()`.

Questions are iterated in order. For each entry, `evaluate_single()` is called with the query function, the embedding model, and the configured thresholds. The function times the query, computes all four metrics, applies the pass/fail rule described in Section 4.5.1, and returns an `EvalResult` dataclass. Retrieval is restricted to the eMOB-52 manual for every question in the golden dataset, ensuring that the evaluation reflects system performance on the document the benchmark was constructed against and that cross-document contamination does not affect the results. When LangFuse credentials are configured in the `.env` file, the five evaluation scores are posted back to the corresponding query trace, making the full evaluation output visible in the LangFuse interface without any additional steps. The fields of `EvalResult` and the aggregate `EvalSummary` are listed in Table 4.7.

Table 4.7: Fields of the `EvalResult` and `EvalSummary` dataclasses.

Dataclass	Field	Type
`EvalResult`	`question_id`	`int`
	`question`	`str`
	`generated_answer`	`str`
	`expected_answer`	`str`
	`semantic_similarity`	`float`
	`keyword_overlap`	`float`
	`completeness_score`	`float`
	`retrieval_hit_rate`	`float`
	`retrieval_precision`	`float`
	`retrieval_recall`	`float`
	`latency_ms`	`float`
	`retrieved_sources`	`List[str]`
	`expected_sources`	`List[str]`
	`passed`	`bool`
`EvalSummary`	`timestamp`	`str`
	`total_questions`	`int`
	`passed_count`	`int`
	`failed_count`	`int`
	`avg_semantic_similarity`	`float`
	`avg_keyword_overlap`	`float`
	`avg_completeness`	`float`
	`avg_retrieval_hit_rate`	`float`
	`avg_latency_ms`	`float`
	`pass_rate`	`float`

After all questions have been evaluated, `run_evaluation()` aggregates the individual `EvalResult` objects into an `EvalSummary`, computing pass count, fail count, pass rate, and per-metric averages. Report generation is conditional on whether an output path was provided. When one is present, a JSON file is written to that path containing the summary, all individual results, and the configuration used for the run. A sibling Markdown report is then written to the same directory, with the path derived by replacing the `.json` extension with `.md`. The Markdown report is produced by `generate_markdown_report()` and includes a summary table, configured thresholds, per-question results separated into passed and failed tables, and a detailed failure section showing truncated answers, per-metric scores, and retrieved sources for each failed question.

The command-line interface is provided by `evals/run_eval.py`. When `-save` is passed without an explicit `-output` path, the output path is auto-generated as `evals/results/eval_model_slug_timestamp.json`. The `-model` flag overrides the active LLM for the run without modifying the `.env` file: it mutates `LLM_MODEL` in `src.services.llm` and calls `get_llm.cache_clear()` to force the cached instance to be rebuilt with the new model on the next call, allowing runs

against alternative models such as `llama-3.3-70b-versatile` without changing the default `openai/gpt-oss-120b` configuration. The remaining flags, `-dataset`, `-similarity-threshold`, and `-keyword-threshold`, map directly to the corresponding parameters of `run_evaluation()`, while `-quiet` is negated to produce the `verbose` argument. The process exits with code 1 if the pass rate falls below 0.5, making the runner suitable for use in a continuous integration pipeline.

4.5.3 LLM-as-a-judge

The LLM-as-a-judge module is implemented in `evals/llm_judge.py` and scores each question across three qualitative dimensions that the automated metrics cannot capture. Each dimension is evaluated independently by a separate prompt to the judge model, and its response is parsed as a structured JSON object.

The three dimensions are faithfulness, answer relevance, and context relevance. Faithfulness measures whether every claim in the generated answer is directly supported by the retrieved context. Answer relevance measures whether the answer actually addresses the question asked; notably, this dimension is evaluated without the retrieved context, using only the question and the answer. Context relevance measures whether the retrieved chunks contain the information needed to answer the question, and is evaluated without the generated answer. Each dimension uses a dedicated prompt containing a rubric anchored at five levels, from 1 (completely wrong or irrelevant) to 5 (fully correct or fully relevant), with explicit descriptions for each level. Providing a rubric rather than a bare scoring instruction is essential: without concrete anchors, different invocations of the same model tend to interpret the scale inconsistently, making scores across questions incomparable. The pass threshold is 4 for all three dimensions, corresponding to answers that are mostly correct with only minor gaps or unsupported details; a threshold of 5 would be too strict for a system operating over technical documentation where answers may be accurate but incomplete, while a threshold of 3 would tolerate responses with significant missing or ungrounded content.

Quantitative calibration signals are appended to each prompt as an optional block when similarity scores are available. The block reports per-chunk cosine similarity values together with their maximum and average, and provides a mapping from similarity ranges to expected scores, shown in Table 4.8. The judge is explicitly instructed to override this mapping when textual content warrants a different assessment. The calibration reference differs per dimension: faithfulness uses the maximum answer-to-chunk similarity, because an answer may be grounded in a single highly relevant chunk even if others are less similar; context relevance uses the average question-to-chunk similarity, because retrieval quality should reflect the overall relevance of the retrieved set rather than its best element; and answer relevance uses the scalar question-to-answer similarity, because this dimension does not involve multiple chunks and is captured by a single embedding comparison between the question and the generated answer.

The judge model defaults to `llama-3.1-8b-instant`, retrieved via `get_judge_llm()` at temperature 0.0 to ensure deterministic and reproducible scoring across repeated evaluation runs. A smaller, separate model is deliberately chosen for two reasons: structured rubric-based

Table 4.8: Similarity-to-score calibration mapping injected into each judge prompt.

Cosine similarity range	Suggested score
> 0.85	5
$0.70 - 0.85$	4
$0.50 - 0.70$	3
$0.30 - 0.50$	2
< 0.30	1

scoring is a simpler task than open-ended answer generation, so a lighter model is sufficient; and keeping the judge distinct from the generation model avoids a situation where a model evaluates its own output. Using the same 120-billion-parameter generation model would also make evaluation runs significantly slower and more costly given that they iterate over the full golden dataset and are invoked repeatedly during development. The judge model is configured independently via the `JUDGE_MODEL` environment variable, allowing both models to be changed without modifying any code. Each prompt invocation is handled by `_call_judge()`, which strips any Markdown code fences from the response before parsing the JSON score and reason. Up to three attempts are made per dimension, with a one-second sleep between attempts; if all attempts fail, the dimension returns `None` rather than raising an error.

The result of each evaluation is stored in an `LLMJudgeResult` dataclass containing a score, a one-sentence reason, and a passed flag for each of the three dimensions, an `overall_passed` field, and the judge model name. `overall_passed` is `True` only when all dimensions for which a score was obtained have passed; dimensions that could not be scored are excluded from the check rather than counted as failures. If no dimension produced a score, `overall_passed` is `None`.

4.5.4 Evaluation CLIs

The evaluation framework exposes two independent command-line interfaces: `evals/run_eval.py` for metric-based evaluation and `evals/llm_judge_eval.py` for LLM-as-a-judge evaluation. Both are designed to be run as modules and produce timestamped reports without requiring changes to the `.env` file.

`run_eval.py` accepts the flags listed in Table 4.9. The `-save` flag triggers automatic output path generation in the form `evals/results/eval_model_slug_timestamp.json`; if `-output` is also provided, `-save`'s path auto-generation is skipped and the explicit path is used instead. The `-model` flag overrides the active LLM for the run by mutating `LLM_MODEL` in `src.services.llm` and calling `get_llm.cache_clear()`, forcing the cached instance to be rebuilt with the new model without modifying the `.env` file. The process exits with code 1 if the pass rate falls below 0.5.

`llm_judge_eval.py` accepts the flags listed in Table 4.10. When `-ids` is provided, only the questions with the listed identifiers are evaluated and `-limit` is ignored; when only `-limit` is provided, evaluation is restricted to the first N questions in the dataset. The `-ids` flag is particularly useful in combination with the metric-based evaluation: after a full

Table 4.9: CLI flags for `run_eval.py`.

Flag	Short	Default	Effect
`-dataset`	`-d`	`data/golden_dataset.json`	Path to the golden dataset
`-output`	`-o`	`None`	Explicit JSON output path
`-similarity-threshold`	`-s`	`0.7`	Minimum semantic similarity to pass
`-keyword-threshold`	`-k`	`0.3`	Minimum keyword overlap to pass
`-model`	`-m`	`None`	Overrides the LLM model at runtime
`-save`	—	`False`	Auto-generates a timestamped output path
`-quiet`	`-q`	`False`	Suppresses per-question progress output

run of `run_eval.py` identifies failing questions by their identifier, those specific questions can be passed to `llm_judge_eval.py` to obtain qualitative scores and one-sentence reasons per dimension, helping to determine whether the failure reflects a retrieval gap, a generation issue, or a mismatch between the generated and expected answers that the automated metrics penalize but the judge considers acceptable. Reports are written to `evals/reports/` as a pair of sibling files sharing a timestamp: `llm_judge_report_{timestamp}.md` and `llm_judge_report_{timestamp}.json`. The process exits with code 1 if the overall pass rate falls below 0.5.

Table 4.10: CLI flags for `llm_judge_eval.py`.

Flag	Short	Default	Effect
`-dataset`	`-d`	`data/golden_dataset.json`	Path to the golden dataset
`-limit`	`-n`	`None`	Evaluate only the first *N* questions
`-ids`	—	`None`	Evaluate only the listed question IDs
`-quiet`	`-q`	`False`	Suppresses per-question progress output

The two CLIs write to different output directories: `run_eval.py` writes to `evals/results/` and `llm_judge_eval.py` writes to `evals/reports/`. Both directories are created automatically if absent. Both tools apply the same exit code convention: the process exits with code 1 when the pass rate falls below 0.5, making them suitable for use in a continuous integration pipeline.

Chapter 5

Evaluation and Results

5.1 Golden dataset

5.1.1 Dataset description

The evaluation is conducted against a golden dataset of 20 question-answer pairs, all written in Portuguese, grounded exclusively in the eMOB-52 CNC machine documentation. The eMOB-52 manual was the first document integrated into the system during development and served as the primary reference throughout the implementation phase, making it the natural choice for constructing a representative and well-understood evaluation benchmark. Each entry contains a question, a reference answer, the expected source documents, the expected source section, and a verified flag that indicates whether the entry has been manually reviewed. All 20 entries in this dataset carry `verified: true`, meaning each question and its corresponding reference answer have been manually reviewed for accuracy against the source manual, with the reference answer being the primary object of verification. When the evaluation is executed against this dataset, retrieval is restricted to the eMOB-52 manual, ensuring that retrieved chunks are drawn exclusively from the document the questions and reference answers were constructed against. The full list of questions is reproduced in Appendix A.

The dataset is organized into six categories, reflecting the range of query types an operator is likely to direct at the system: procedural, technical specifications, maintenance, troubleshooting, safety, and operation. Table 5.1 shows the distribution of questions across categories.

Table 5.1: Question distribution across categories in the golden dataset.

Category	**Questions**
Procedural	8
Technical specs	4
Maintenance	4
Troubleshooting	2
Safety	1
Operation	1
Total	**20**

The distribution is deliberately weighted toward procedural questions, which account for eight of the twenty entries. This reflects the primary use case the system is intended to support, namely an operator consulting the manual for step-by-step instructions during machine setup, tool changes, die replacement, and technical maintenance procedures on the CNC controller. Technical specifications and maintenance each contribute four entries, covering factual lookups such as axis speeds and electrical ratings as well as scheduled maintenance tasks and their intervals. The remaining categories, troubleshooting, safety, and operation, are represented more sparsely, but they ensure that the benchmark exercises interaction modes beyond procedural retrieval.

A dataset of 20 pairs is small by general NLP benchmarking standards, and metric differences between configurations should be interpreted with that in mind: with 20 questions, each individual result corresponds to a five percentage point shift in pass rate, so small numerical gaps between configurations do not carry strong statistical weight. Furthermore, the dataset is grounded in a single document, which means the results reflect system performance on the eMOB-52 manual specifically and may not generalize to the full range of documents the system could be deployed against in the future. The dataset is nonetheless meaningful for this application for two reasons. First, the question categories provide structured coverage of the main query types the system is expected to handle, ensuring that evaluation is not concentrated in a single interaction mode. Second, the human verification step ensures that each reference answer is factually correct and answerable from the documentation, removing ambiguous or unanswerable questions that would introduce noise into the results.

5.1.2 Results

Beyond the category distribution, the qualitative character of the questions determines what the benchmark actually exercises. The entries span two broad answer shapes: procedural and maintenance questions expect structured, multi-step reference answers, while technical specification questions expect dense factual enumerations. Table 5.2 presents one representative question from each of three categories, paired with the source section that grounds it.

Table 5.2: Representative questions across categories with their grounding sections.

Category	Question	Source section
Procedural	*Quais são os passos para trocar a matriz?*	7.6.1. Matriz
Technical specs	*Quais são as velocidades máximas dos eixos da máquina eMOB-52?*	3.1. Especificações Técnicas
Troubleshooting	*Quais são as causas prováveis e soluções para rugas de curvatura internas?*	7.10. Principais Anomalias da Máquina

Running the full evaluation against the golden dataset with `openai/gpt-oss-120b` as the default model yields a pass rate of 80%, with 16 of the 20 questions passing all gating conditions. Retrieval is restricted to the eMOB-52 manual, consistent with the golden dataset being grounded exclusively in that document. Average semantic similarity across all questions is 92.4%, average keyword overlap is 51.1%, and the retrieval hit rate is 100%. Table 5.3 reports the per-question outcome across all entries.

Table 5.3: Per-question evaluation results. Sem: semantic similarity; KW: keyword overlap; Pass: all gating conditions met.

ID	Category	Sem (%)	KW (%)	Pass
Q1	Procedural	94.4	59.5	**Yes**
Q2	Procedural	95.5	67.6	**Yes**
Q3	Procedural	92.2	31.9	**Yes**
Q4	Procedural	93.8	65.5	**Yes**
Q5	Procedural	93.0	51.5	**Yes**
Q6	Procedural	94.8	36.6	**Yes**
Q7	Procedural	91.1	50.0	**Yes**
Q8	Procedural	92.2	66.1	**Yes**
Q9	Technical specs	84.1	12.5	No
Q10	Technical specs	92.8	90.9	**Yes**
Q11	Technical specs	92.3	46.2	**Yes**
Q12	Technical specs	94.9	42.9	**Yes**
Q13	Maintenance	91.5	39.0	**Yes**
Q14	Maintenance	91.4	16.1	No
Q15	Maintenance	95.3	81.8	**Yes**
Q16	Maintenance	91.4	25.0	No
Q17	Troubleshooting	95.1	93.3	**Yes**
Q18	Troubleshooting	92.1	63.0	**Yes**
Q19	Safety	91.5	63.6	**Yes**
Q20	Operation	88.2	19.4	No

The two metrics, semantic similarity and keyword overlap, do not always move in the same direction, and examining four representative questions across the score spectrum reveals how each metric captures a different dimension of answer quality. The four examples below cover the combinations of high and low scores for both metrics, illustrating when the two measures agree, when

they diverge, and what each divergence means in practice.

High semantic similarity, high keyword overlap — Q17. Q17, which asks *Quais são as causas prováveis e soluções para rugas de curvatura internas?*, achieves a semantic similarity of 95.1% and a keyword overlap of 93.3%, the highest keyword overlap score in the dataset. Table 5.4 shows the reference and generated answers side by side.

Table 5.4: Reference answer versus generated answer for Q17, illustrating high agreement on both metrics.

Reference answer	**Generated answer**
Causas prováveis: Mandril excessivamente retraído; Mandril inadequado; Uso inadequado do bico de pato. Soluções: Avançar o mandril; Reajustar o bico de pato; Entre em contato com a AMOB.	Causas prováveis — Mandril excessivamente retraído; Mandril inadequado; Uso inadequado do bico de pato. Soluções — Avançar o mandril; Reajustar o bico de pato; Entrar em contato com a AMOB.

The near-verbatim reproduction of a short, bounded factual list leaves little room for paraphrase, so both token overlap and semantic similarity are near-maximal. This represents the ideal case for the metric pair: they agree strongly when the reference is short and enumerable and the retrieved chunk contains the answer almost verbatim.

High semantic similarity, low keyword overlap — Q14. Q14, which asks *Com que frequência devem ser lubrificados os fusos de esferas e que lubrificante deve ser usado?*, achieves a semantic similarity of 91.4% but a keyword overlap of only 16.1%, falling below the 30% threshold and resulting in a failure. Table 5.5 shows the reference and generated answers side by side.

Table 5.5: Reference answer versus generated answer for Q14, illustrating the effect of retrieval insufficiency on generated vocabulary.

Reference answer	**Generated answer**
Os sistemas de fusos de esferas devem ser lubrificados a cada 170 horas ou uma vez por semana. O lubrificante a utilizar é *Kernite NGLI 2 (graxa)*. A máquina deve estar parada durante esta operação e é necessário 1 operador. Os eixos abrangidos são: Mandril, Posicionador, Movimento Horizontal, Movimento Vertical, Mordaça, Paralela, Acompanhamento da Paralela e Braço de Curvatura.	Os fusos de esferas devem ser lubrificados manualmente a cada 170 horas de operação (equivalente a aproximadamente uma vez por semana). Para essa lubrificação deve-se aplicar *gel lubrificante* (gel específico para fusos de esferas) conforme indicado nos procedimentos de manutenção. Frequência: 170 h (ou 1 vez/semana). Lubrificante: gel lubrificante para fusos de esferas.

The frequency is correct in both answers, but the product name *Kernite NGLI 2 (graxa)*, the eight axis names, and the operational conditions are entirely absent from the generated answer because the chunks containing them never reached the model. The embedding model registers the shared topic and correctly assigns high semantic similarity, while the Jaccard metric registers the absence of the specific product name and axis vocabulary as a low intersection, producing the divergence between the two scores.

Low semantic similarity, high keyword overlap — Q19. Q19, which asks *Que equipamento de proteção individual deve ser usado e que acessórios são proibidos durante a manutenção?*, achieves a keyword overlap of 63.6% but a semantic similarity of 91.5%, among the lowest semantic similarity scores among the passing questions, tied with Q13 and only marginally above Q7's 91.1%. Table 5.6 shows the reference and generated answers side by side.

Table 5.6: Reference answer versus generated answer for Q19, illustrating structural rewording with strong lexical overlap.

Reference answer	**Generated answer**
Os operadores e o pessoal de manutenção devem utilizar equipamento de proteção individual, por exemplo, óculos, botas e luvas. Não usar acessórios ou joias, incluindo pulseiras, relógios, anéis ou colares, ao realizar operações de manutenção em que a operação em área desprotegida seja necessária, para evitar riscos mecânicos como arrasto, aprisionamento ou outros. Usar calçados de segurança e luvas de proteção durante todas as etapas operacionais, trocas de ferramentas, fornecimento de carregadores, manuseio de peças e configurações da máquina.	Equipamento de Proteção Individual (EPI) a ser usado — Óculos de segurança; Botas ou calçados de segurança; Luvas de proteção (para todas as etapas operacionais, troca de ferramentas, fornecimento de carregadores, manuseio de peças e ajustes da máquina). Acessórios proibidos durante a manutenção — Qualquer tipo de joia ou acessório pessoal, como pulseiras, relógios, anéis ou colares, que possam ser arrastados, presos ou causar outros riscos mecânicos ao operar em áreas desprotegidas. Em resumo, durante a manutenção o operador deve usar óculos, calçado de segurança e luvas de proteção, e deve evitar o uso de joias ou acessórios que possam representar risco de arrasto ou aprisionamento.

The answer reuses many exact reference terms, producing a strong lexical match. However, the model restructures the content into labelled subsections, reorders material relative to the reference, and appends a summarizing paragraph not present in the reference. This structural rewording discounts cosine similarity even though no fact is wrong or missing, making Q19 the mirror image of the Q14-style cases: there, wording diverges while facts match, keeping semantic similarity high but keyword overlap low; here, wording and terms match closely but structural rewriting still reduces semantic similarity somewhat, showing the two metrics do not move in lockstep in either direction.

Low semantic similarity, low keyword overlap — Q9. Q9, which asks *Quais são as especificações técnicas de alimentação elétrica e potência da máquina eMOB-52?*, achieves a semantic similarity of 84.1%, the lowest in the dataset though still above the 70% gate, and a keyword overlap of 12.5%, resulting in a failure. Table 5.7 shows the reference and generated answers side by side.

Table 5.7: Reference answer versus generated answer for Q9, illustrating total omission of content due to retrieval failure.

Reference answer	**Generated answer**
Tensão de alimentação: 400 V ±10% (-15% máx. durante 1 minuto). Tipo de potência: Trifásico sem neutro. Frequência: 47-63 Hz. Potência instalada: 22 kVA. Corrente nominal: 31 Um. Tensão auxiliar: 24 Vdc. Tensão de alimentação CNC: 24 Vdc. Grau de proteção do compartimento: IP55.	Desculpe, mas nas informações fornecidas não há detalhes sobre as especificações técnicas de alimentação elétrica (tensão, frequência, corrente) nem sobre a potência da máquina eMOB-52. Para obter esses dados, consulte a seção correspondente do manual técnico ou entre em contato com o suporte da AMOB S.A.

This is a refusal rather than a paraphrase. The model states outright that the requested data was not present in its context, so the generated text shares almost no tokens with the dense factual reference, producing a keyword overlap of 12.5%, and is also semantically distant since the response is a deflection rather than a factual answer, producing the lowest semantic similarity in the dataset at 84.1%. This is a cleaner low/low case than the other failures because the failure mode is total omission rather than vocabulary drift: the relevant section did not reach the final context, and both metrics register the complete absence of the expected content.

All four failures share the same surface pattern: semantic similarity remains above the 70% threshold in every case, ranging from 84.1% to 91.4%, while keyword overlap falls below the 30% threshold in all four. The retrieval hit rate is 100%, meaning the eMOB-52 document was present in the retrieved chunks for every question. Two distinct failure modes account for the four cases, though they affect a different proportion of the dataset than the document-level retrieval hit rate alone would suggest: retrieval or reranking insufficiency accounts for three of the four failures, with vocabulary divergence at generation responsible for only one.

The first and dominant failure mode is retrieval or reranking insufficiency, where the chunks or sections containing the specific terminology needed to answer precisely are never surfaced to the model, despite the correct document being retrieved overall. Q14, which asks *Com que frequência devem ser lubrificados os fusos de esferas e que lubrificante deve ser usado?*, is the clearest instance: the chunk naming the lubricant as *Kernite NGLI 2 (graxa)* ranks 20th in the similarity search, and the table row enumerating the eight axes covered by this lubrication task is similarly absent from the retrieved chunks, both falling outside the top 15 candidates passed to the reranker and therefore never available to the model at generation time. Lacking access to the specific product name and the axis list, the model produced a plausible but generic substitute, *gel lubrificante*, as already shown in Table 5.5.

Q16, which asks *Que cuidados e verificações são definidos para motores e codificadores?*, shows the same pattern at a finer level: the generated answer describes the same maintenance tasks as the reference but uses different phrasing throughout, for example replacing *tomada do codificador* with *ficha do codificador*, and enumerates an additional six-month inspection task absent from the reference, diluting the Jaccard intersection without introducing factual error. Q20, which asks *Quais são os três modos de funcionamento da máquina e como se distinguem?*, presents

a subtler variant: the model retrieved content from a different section of the manual covering the same three operating modes but using different vocabulary, so terms such as *seletor de movimento manual* and *potenciómetro* appear in the generated answer while the reference uses *calibrar dispositivos mecânicos* and *teclado*. Both descriptions are factually correct but share few exact tokens, and the Jaccard metric penalizes every term in the generated answer that does not appear in the reference regardless of semantic equivalence.

The third instance of this same failure mode appears at Q9, which asks *Quais são as especificações técnicas de alimentação elétrica e potência da máquina eMOB-52?*. The relevant section did not reach the final context, as already shown in Table 5.7. The model declined to answer, correctly stating that the specifications were not available in the provided context. The specification data is stored as a single large table chunk containing all electrical parameters together, and this dense chunk ranked outside the top 15 candidates before reranking, meaning it was excluded before the CrossEncoder ever scored it. This points to a limitation of the current table ingestion strategy: large specification tables stored as a single chunk produce an embedding that averages across all their parameters, diluting the signal for any individual parameter query and making it harder for the similarity search to surface the chunk in response to a specific question. Splitting such tables into smaller, more focused chunks at ingestion time would allow each parameter group to be retrieved independently, improving keyword matching precision and reducing the likelihood of this type of exclusion.

Taken together, these results show that the system generates semantically faithful answers across the dataset, with every question scoring above 80% semantic similarity regardless of pass or fail status, but they also reveal that retrieval hit rate alone does not guarantee that the right content reaches the model. Three of the four failures are attributable to chunk-level or section-level ranking gaps that the document-level hit rate metric cannot detect, and a finer-grained chunk-level metric would be needed to surface this failure mode directly in future evaluation work. Q16 remains the only failure where the relevant content reached the model and generation-side vocabulary divergence was the sole cause of the metric penalty.

5.1.3 Latency profile

Figure 5.1 shows the end-to-end query latency for each of the 20 questions in the golden dataset. Latency is measured as the time from query submission to final result return, covering the complete pipeline: vector retrieval, CrossEncoder reranking, and LLM generation over the Groq API.

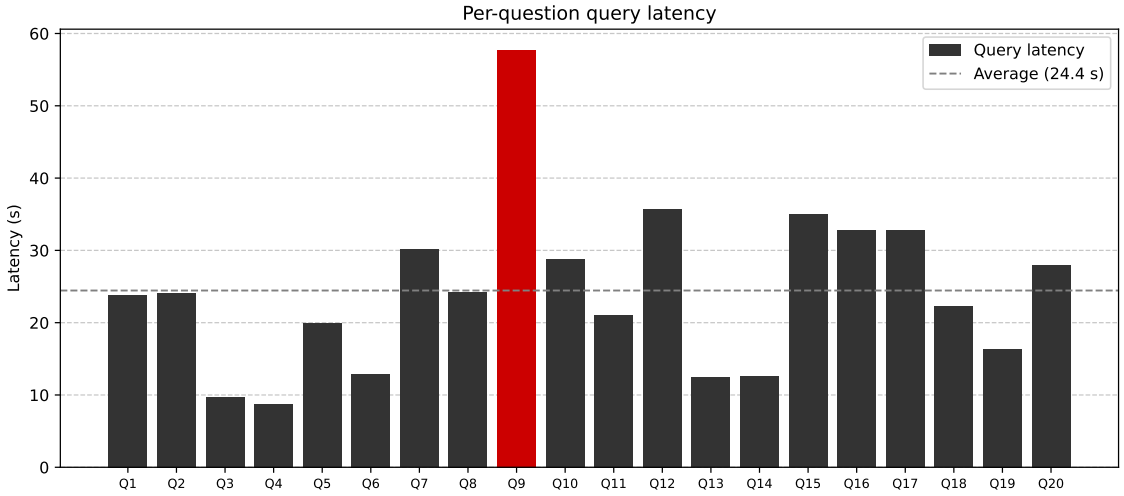

Figure 5.1: Per-question query latency across the golden dataset.

Table 5.8 reports a stage-level breakdown across five representative queries, revealing where time is actually spent within the pipeline.

Table 5.8: Stage-level latency breakdown across five representative queries.

Question	**Retrieval (s)**	**Rerank (s)**	**Generation (s)**	**Total (s)**
Como troco a paralela?	0.377	11.525	2.179	14.455
Como soluciono excesso de ovalização?	0.048	29.235	1.294	30.792
O que controla o movimento on/off?	0.041	19.584	0.752	20.584
Quais são os procedimentos da manutenção mecânica?	0.049	6.305	1.647	8.172
Qual o peso da máquina?	0.037	24.641	1.363	26.209
Average	0.110	18.258	1.447	20.042
Share of total	0.5%	91.0%	7.2%	100%

The CrossEncoder reranker, `BAAI/bge-reranker-v2-m3`, running on CPU accounts for approximately 91% of total latency on average. Vector retrieval is negligible at 0.5% and LLM generation via the Groq API contributes only 7.2%. Reranking time varies considerably across queries, from 6.3 s to 29.2 s, depending on the length and content of the fifteen candidate chunks being scored, not on the length or correctness of the final answer.

The average end-to-end latency across all 20 golden dataset questions is 24.4 s, with a range from 8.7 s at Q4 to 57.7 s at Q9. Q9 is a clear outlier: it is the only question where the model declined to answer, and its latency is more than twice the average, suggesting that significant reranking time was spent on candidate chunks that, although drawn from the correct source document, did not contain the specific electrical specification table the question required. The elevated latency at Q9 is therefore a symptom of the chunking gap diagnosed in Section 5.1.2, where the relevant table was not captured in any of the top-ranked chunks passed to the model.

Outside of Q9, latency does not correlate with pass or fail status. The longest passing responses, Q15 at 35.0 s and Q17 at 32.8 s, are detailed multi-part answers that happen to use the correct domain vocabulary. The shortest failure is Q14 at 12.6 s, a question where the model produced a quick answer using the correct frequency but substituted a generic lubricant description for the specific product name. For an industrial documentation assistant, an average of 24.4 s is an acceptable cost for a look-up tool where the operator submits a question and waits for a complete answer, but would be noticeable in a conversational flow requiring rapid back-and-forth. The primary lever for latency reduction is the reranker itself, since it accounts for approximately 91% of total query time, and replacing it with a lighter reranking model would yield the largest improvement, at the cost of potentially reduced retrieval precision.

5.2 Baseline evaluation: standalone LLM without retrieval

The end-to-end results in the previous section establish that the system answers most questions correctly, but they do not by themselves prove that retrieval is responsible for that performance. A reasonable question is whether the language model already holds enough knowledge about tube bending machinery to answer the golden dataset questions on its own, making the retrieval stage redundant. This section addresses that question directly by evaluating the same model on the same dataset with retrieval removed, isolating the contribution of the documentation to the quality of the answers.

5.2.1 Methodology

The baseline experiment runs the language model in isolation, bypassing the query pipeline entirely. The script never queries the vector store or invokes the retriever; instead, each question is passed directly to the LLM model through a minimal prompt that supplies only the question text, with no system prompt, no retrieved context. Listing 5.1 shows the chain construction.

Listing 5.1: Baseline chain: the model receives only the raw question.

```
BASELINE_PROMPT = ChatPromptTemplate.from_messages([
    ("human", "Answer the following question: {question}"),
])
chain = BASELINE_PROMPT | get_llm() | StrOutputParser()
answer = chain.invoke({"question": q["question"]})
```

Every other element of the experiment is held fixed so that the comparison isolates the effect of retrieval alone. The baseline calls the same `get_llm()` factory used by the full pipeline, which means it runs the same model, `openai/gpt-oss-120b`, at the same temperature of 0.0. It iterates over the same 20 questions from the golden dataset and scores the answers with the same metric functions, namely semantic similarity, keyword overlap, and answer completeness. The retrieval hit rate is the only metric omitted, since it has no meaning when nothing is retrieved. A question is counted as passing under the same gating conditions as the end-to-end evaluation:

semantic similarity at or above 0.70 and keyword overlap at or above 0.30. This design constitutes a fair baseline, because the model is given every advantage except access to the documentation: the same weights, the same decoding settings, and the same questions, differing only in whether the eMOB-52 manual is available to ground the answer.

5.2.2 Results

The standalone model passes none of the 20 questions. Table 5.9 contrasts the aggregate baseline metrics against the retrieval-augmented configuration from Section 5.1.

Table 5.9: Aggregate metrics for the standalone baseline against the retrieval-augmented pipeline.

Metric	**RAG**	**Baseline**
Pass rate	80% (16/20)	0% (0/20)
Average semantic similarity	92.4%	86.4%
Average keyword overlap	51.1%	4.8%
Average completeness	85.4%	40.8%

Figure 5.2 shows the per-question breakdown across keyword overlap and answer completeness for both configurations, making the distribution of gains visible across all 20 questions.

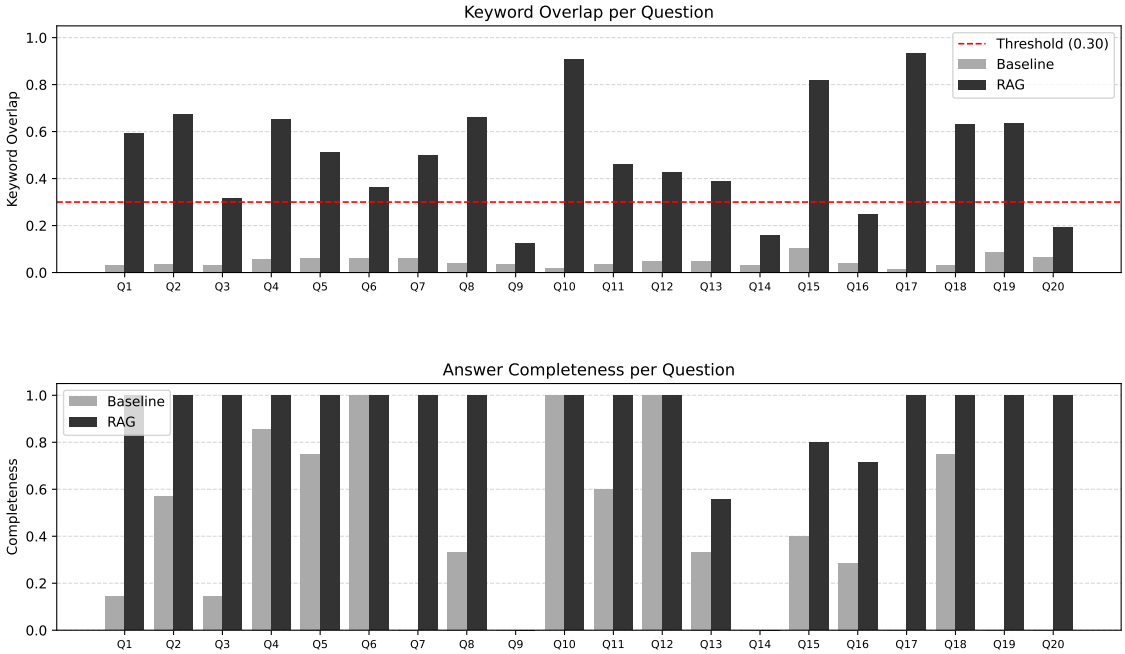

Figure 5.2: Per-question keyword overlap and answer completeness for the standalone baseline and the retrieval-augmented pipeline. The dashed red line in the upper panel marks the keyword overlap pass threshold of 0.30.

The drop in average completeness from 85.4% to 40.8% is particularly consequential in this domain. Answer completeness measures whether the generated answer covers all the key points

present in the reference answer, and it is most reliable and most meaningful for procedural questions, where each uncovered point corresponds to a step that the operator would simply not be told to perform. A baseline answer that covers 40% of the reference points for a die replacement or lubrication procedure is not a partial answer that can be supplemented by intuition; it is a procedure with missing steps, and in an industrial environment operating with servo-controlled axes and hydraulic clamping forces, a missing step is a direct path to machine damage or personal injury.

The headline result is unambiguous: removing retrieval drops the pass rate from 80% to zero. What makes the result diagnostic rather than merely negative is the shape of the failure. Average semantic similarity remains high at 86.4%, only six percentage points below the retrieval-augmented configuration, which means the standalone model is talking about the right general topic for nearly every question. Keyword overlap, however, collapses from 51.1% to 4.8%. The model produces plausible, fluent answers about the subject category, but it does not use the machine-specific terminology, part names, axis designations, measurements, or procedural steps that exist only in the eMOB-52 documentation. The semantic similarity metric registers the topical match while the keyword metric exposes the absence of grounded detail, and it is precisely this grounded detail that an operator consulting the manual needs. Notably, several questions score 1.0 completeness in the baseline, meaning the model covered the right topics at a conceptual level, yet their keyword overlap remains near zero, confirming that topic awareness and vocabulary grounding are distinct dimensions of answer quality that the baseline fails to bridge.

The baseline failures span three qualitatively different risk levels. Table 5.10 shows the RAG-generated answer alongside the baseline answer for three representative questions, making the contrast between grounded and ungrounded generation concrete.

Table 5.10: RAG answer versus baseline answer for three representative questions. The baseline answer for Q1 names no machine axis or component, and the answer for Q10 lists only three axes with incorrect units and values that do not correspond to the eMOB-52 specification.

ID	RAG answer	Baseline answer
Q1	Passos para trocar a matriz (seção 7.6.1): 1. Mover o eixo Y para 90°; 2. Abrir totalmente a paralela (P) e a mordaça (C); 3. Mover a cabeça para baixo (posicionar o eixo H no valor máximo); 4. Soltar totalmente a porca de fixação; 5. Remover as matrizes e demais espaçadores, substituindo-as por novos componentes; ao inserir as novas matrizes de curvatura, colocar a chave entre cada matriz; 6. Apertar completamente a porca de bloqueio da matriz de curvatura.	Como trocar a matriz (de forma geral): 1. Leia o manual do equipamento; 2. Reúna as ferramentas corretas (chave de fenda, chave allen, pinças antiestáticas, luvas, álcool isopropílico); 3. Desconecte a energia; 4. Remova as tampas ou painéis de proteção; 5. Desconecte cabos e retire a matriz; 6. Instale a nova matriz e reconecte os cabos; 7. Recoloque os painéis e teste o funcionamento.
Q9	*"Nas informações fornecidas não há detalhes sobre as especificações técnicas de alimentação elétrica nem sobre a potência da máquina eMOB-52. Consulte a seção correspondente do manual técnico ou entre em contato com o suporte da AMOB S.A."*	*"Desculpe, mas não disponho de informações específicas sobre as características de alimentação elétrica e potência da máquina eMOB-52."*
Q10	Velocidades máximas: X 1500 mm/s, Y 100 °/s, Z 672 °/s, W 312 mm/s, H 96 mm/s, M 192 mm/s, C 115 mm/s, P 115 mm/s, E 455 mm/s. Fonte: Tabela 2 do documento eMOB-52_PT_v1.1.md.	Velocidades máximas: X/Y 60 m/min em rapid, 30 m/min em corte; Z 30 m/min em rapid, 15 m/min em corte.

The stakes of these failures are not uniform, but they are consistently high in an industrial context. The eMOB-52 is a rotary draw bending machine operating with servo-controlled axes, hydraulic clamping forces, and tooling that moves at speeds of up to 1500 mm/s. An operator acting on incorrect procedural or specification information risks machine damage, tooling failure, or, most importantly, personal injury. Q1 illustrates a failure where the answer is obviously unhelpful, since no operator would follow a generic procedure that names no axis and no component, but the risk remains real if the operator, under time pressure, attempts to infer the correct procedure from the generic steps. Q10 is the most dangerous case: the baseline answer presents fabricated axis speeds with specific numeric values and units, formatted as a reference table, with no indication that the information does not correspond to the actual machine, covering only three of the nine axes with incorrect units and values that bear no relation to the eMOB-52 specification. An operator configuring axis parameters from this answer would be working with values that are wrong in unit, wrong in count, and wrong in magnitude. Q9 is the only case where the risk is contained: both systems declined to answer, and the operator is correctly directed to consult the source documentation.

These results provide the empirical justification for the retrieval-augmented design. The language model alone, however capable, cannot answer questions whose correct answers exist only in documentation it was never trained on, and in an industrial setting the cost of a confidently

wrong answer is not merely inconvenience but potential harm. Retrieval is not an incremental enhancement to the system but the component that makes domain-specific answers both possible and safe.

5.3 LLM-as-a-judge evaluation

5.3.1 Methodology

The automated metrics established in Section 5.1 measure answer quality through string overlap and embedding distance, both of which are sensitive to vocabulary and phrasing choices that do not necessarily reflect correctness. The LLM-as-a-judge framework complements these metrics by scoring each answer on three qualitative dimensions that a deterministic metric cannot capture: faithfulness, answer relevance, and context relevance.

Faithfulness measures whether every claim in the generated answer is directly supported by the retrieved context, penalizing answers that introduce information not present in the retrieved chunks. Answer relevance measures whether the answer addresses the question that was asked, independently of grounding. Context relevance measures whether the retrieved chunks contained the information needed to answer the question, providing a retrieval-side quality signal that is separate from generation quality. Each dimension is scored on an integer scale from 1 to 5, with rubric anchors defined at each point. A score of 5 on faithfulness indicates that every claim is directly supported by the context; a score of 1 indicates that the answer contradicts or completely ignores it. The pass threshold for all three dimensions is 4, corresponding to answers that are mostly correct with only minor gaps, and a question is considered to have passed the judge evaluation only when all three dimensions individually meet the threshold.

The judge runs on a separate model, `llama-3.1-8b-instant`, at temperature 0.0. Using a model that is distinct from the generation model and running it at zero temperature ensures that the judge scores are deterministic and independent of the system's own generation choices. The choice of a lighter model for this role is deliberate: unlike the generation model, which must synthesize a complete, well-structured answer to an open-ended user question from retrieved context, the judge performs a narrower classification task, scoring three predefined dimensions against a fixed rubric rather than producing free-form content. This lower demand on generative capability makes a smaller, faster model an appropriate and cost-effective choice for the evaluation role. Each judge prompt includes a calibration scale that maps cosine similarity ranges to expected integer scores, providing a quantitative anchor that reduces variance in how the judge interprets the rubric. The mapping is: similarity above 0.85 corresponds to a score of 5, 0.70–0.85 to 4, 0.50–0.70 to 3, 0.30–0.50 to 2, and below 0.30 to 1, with the instruction to override based on textual judgement when the content warrants it. If the judge response cannot be parsed after three attempts, the dimension returns `None` and is excluded from the overall pass decision rather than counted as a failure.

5.3.2 Results

The judge evaluation was run against all 20 questions using `openai/gpt-oss-120b` as the generation model, consistent with the end-to-end evaluation in Section 5.1. Table 5.11 reports the per-question scores across all three dimensions.

Table 5.11: LLM-as-a-judge scores per question. F: faithfulness; AR: answer relevance; CR: context relevance. Pass threshold is 4 for all dimensions.

ID	Category	F	AR	CR	Judge	Automated
Q1	Procedural	5	5	5	**Pass**	**Pass**
Q2	Procedural	5	5	5	**Pass**	**Pass**
Q3	Procedural	5	5	5	**Pass**	**Pass**
Q4	Procedural	5	5	5	**Pass**	**Pass**
Q5	Procedural	5	5	5	**Pass**	**Pass**
Q6	Procedural	5	5	5	**Pass**	**Pass**
Q7	Procedural	5	5	5	**Pass**	**Pass**
Q8	Procedural	5	5	5	**Pass**	**Pass**
Q9	Technical specs	2	4	5	Fail	Fail
Q10	Technical specs	5	5	5	**Pass**	**Pass**
Q11	Technical specs	5	5	5	**Pass**	**Pass**
Q12	Technical specs	5	5	5	**Pass**	**Pass**
Q13	Maintenance	5	5	5	**Pass**	**Pass**
Q14	Maintenance	5	5	5	**Pass**	Fail
Q15	Maintenance	5	5	5	**Pass**	**Pass**
Q16	Maintenance	5	5	5	**Pass**	Fail
Q17	Troubleshooting	5	5	5	**Pass**	**Pass**
Q18	Troubleshooting	5	5	5	**Pass**	**Pass**
Q19	Safety	5	5	5	**Pass**	**Pass**
Q20	Operation	5	5	5	**Pass**	Fail

The judge passes 19 of the 20 questions, with average scores of 4.85 for faithfulness, 4.95 for answer relevance, and 5.00 for context relevance. The most informative aspect of these results is not the aggregate pass rate but the reasoning the judge attaches to each dimension, since the faithfulness dimension in particular identifies the specific retrieved chunk that grounds each claim, providing a traceable link between the generated answer and its source that the deterministic metrics cannot express.

Two passing questions illustrate this behavior clearly. Q5, which asks *Como se substitui o lubrificante da caixa de velocidades do eixo de curvatura?*, and Q10, which asks *Quais são as velocidades máximas dos eixos da máquina eMOB-52?*, both received a perfect score on all three dimensions, and in each case the judge's faithfulness justification names the exact chunk from which the answer was drawn. Tables 5.12 and 5.13 report the per-dimension scores and the judge's stated reasoning for these two questions.

Table 5.12: LLM-as-a-judge report for Q5, showing per-dimension scores and reasoning.

Dimension	Score	Judge reasoning
Faithfulness	5 / 5	Every claim in the answer is supported by the retrieved context, with the steps and details matching those found in Chunk 3.
Answer relevance	5 / 5	The answer fully addresses the question, providing a step-by-step guide to replacing the gearbox lubricant of the bending axis.
Context relevance	5 / 5	The retrieved context contains the complete step-by-step instructions needed to answer the question.

Table 5.13: LLM-as-a-judge report for Q10, showing per-dimension scores and reasoning.

Dimension	Score	Judge reasoning
Faithfulness	5 / 5	Every claim in the answer is supported by the retrieved context, specifically Tabela 2, machine specifications (part 2), located in Chunk 1.
Answer relevance	5 / 5	The answer directly addresses the question by providing a table of the maximum velocities of each axis.
Context relevance	5 / 5	The retrieved context lists the maximum velocity of each axis, fully covering the information the question requires.

In both cases the ability of the judge to point to the originating chunk, rather than merely returning a numeric verdict, is the property that makes this form of evaluation useful as a diagnostic tool, since it turns each score into an explanation that can be inspected and challenged.

Q9, which asks *Quais são as especificações técnicas de alimentação elétrica e potência da máquina eMOB-52?*, is the only question that both frameworks fail, and it also exposes a limitation of the judge. The judge assigned a context relevance score of 5, indicating that the retrieved chunks contained the electrical specification data, and a faithfulness score of 2, penalizing the model's refusal for not drawing on that data. However, the trace inspection reported in Section 5.1 established that the relevant specification chunk never reached the model, so the context the judge scored as fully sufficient did not in fact contain the answer. The judge's high context relevance score is therefore inconsistent with the actual content passed to the model, and the low faithfulness score appears to follow from that same misjudgement rather than from a genuine fault in the model's behavior, which was in fact an appropriate refusal given the context it received. This case suggests that the judge scores, while informative, should be read as a further signal rather than as ground truth.

Several aspects of the evaluation design may push the judge toward high scores, and the results are therefore reported with corresponding caution rather than as definitive measures of quality. The judge model, `llama-3.1-8b-instant`, is a lightweight model evaluating the output of a considerably larger generation model, and this capability asymmetry is a known source of excessive permissiveness in LLM-as-a-judge setups [26]. The calibration scale injected into each prompt may compound this, since the reranker selects the most similar chunks and the resulting high

similarity values anchor the judge toward scores of 4 or 5 before any content is read. The pass threshold of 4 is also permissive by design, admitting minor unsupported details or gaps, and the uniform context relevance average of 5.00 across all questions, together with the Q9 inconsistency above, suggests the judge does not discriminate finely along that dimension. For the three questions where the judge and the automated metrics disagree, Q14, Q16, and Q20, the character of each answer has already been established in Section 5.1, and the judge's uniformly high scores are better read as a reflection of this excessive permissiveness than as an independent verdict on those answers.

Despite these limitations, the LLM-as-a-judge framework offers something the deterministic metrics of the golden dataset cannot. Semantic similarity and keyword overlap operate purely on surface form and embedding distance, so they cannot explain why an answer succeeds or fails, only that it diverges from the reference. The judge, by contrast, reasons over the actual content of the question, the retrieved context, and the generated answer, and it can attribute a claim to a specific source chunk or identify where grounding breaks down. This makes it a valuable complement to the deterministic metrics rather than a replacement for them, particularly for diagnosing individual cases where a numeric score alone leaves the cause of a result ambiguous. A more robust configuration, using a stronger judge model, removing the numerical calibration anchors, raising the faithfulness threshold, and adding an explicit reasoning step before scoring, would reduce the excessive permissiveness observed here and is a natural direction for future evaluation work.

5.4 Response analysis

The following examples illustrate system behavior across representative query types. For the first four examples, screenshots from the chat interface show the full interaction as it appears to the user, with the top retrieved chunk shown alongside the generated answer to make visible the connection between the retrieved context and the response the model produced. The fifth example, illustrating multi-turn conversation with chat history, is presented as a table pairing each question with its generated answer rather than as interface screenshots.

Die replacement procedure

The first question shown asks for the steps to replace the bending die: *Quais são os passos para trocar a matriz?*. Figure 5.3 shows the generated answer as it appears in the chat interface.

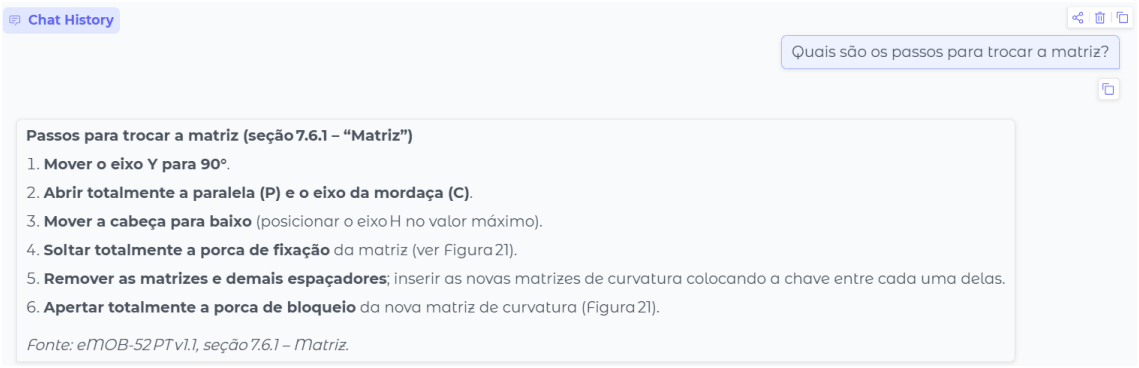

Figure 5.3: Generated answer for the die replacement question as displayed in the chat interface.

The model produced a six-step numbered procedure using the axis designations and component names from the source text, appending references to Figura 21 on the steps where the manual itself cites the figure. Figure 5.4 shows the top retrieved chunk that grounded this answer, drawn from section 7.6.1 of the eMOB-52 manual.

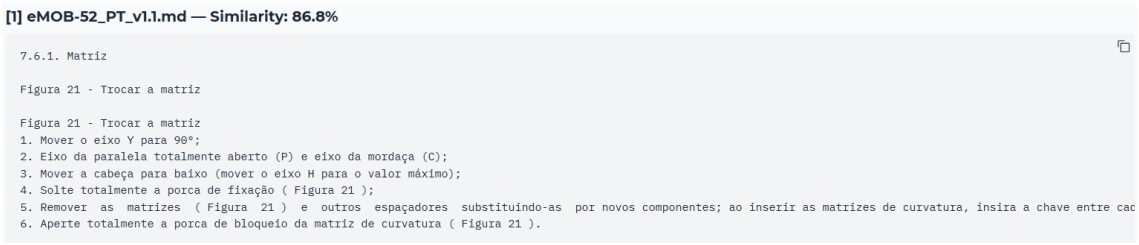

Figure 5.4: Top retrieved chunk for the die replacement question.

The chunk contains the complete procedure with three inline references to Figura 21, providing the model with all the information needed to produce the answer without relying on any prior knowledge. Because the model explicitly referenced Figura 21 in its response, the primary figure resolution path was triggered: the system extracted the figure number from the answer text, located the corresponding image file, and rendered it alongside the answer in the interface. Figure 5.5 shows the resolved image as presented to the user.

Figure 5.5: Figura 21 resolved and rendered alongside the answer.

Sensor maintenance tasks

The second example demonstrates table resolution. The question asks which maintenance tasks are defined for position sensors and at what frequency: *Que tarefas de manutenção são definidas para os sensores de posição e com que frequência?*. Figure 5.6 shows the generated answer as it appears in the chat interface.

Figure 5.6: Generated answer for the sensor maintenance question as displayed in the chat interface.

The model produced a structured table listing three maintenance tasks, their periodicity, and their estimated duration, drawn directly from Tabela 5 of section 7.2.1. Figure 5.7 shows the top retrieved chunk, a `table_row` document carrying `table_id = tabela_5`.

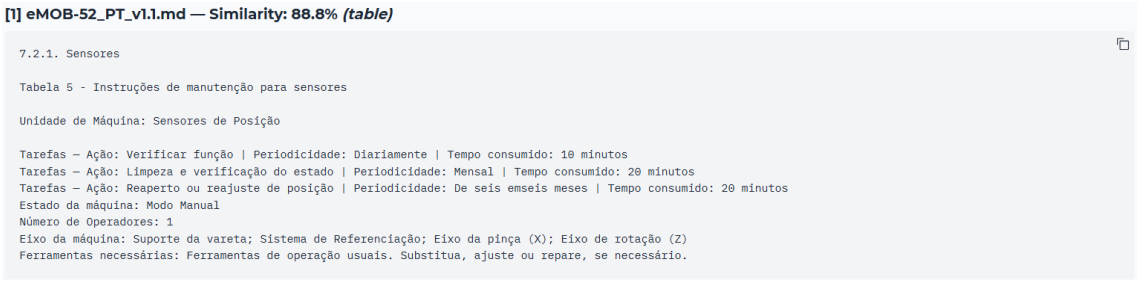

Figure 5.7: Top retrieved chunk for the sensor maintenance question.

Because the top-ranked chunk is a `table_row` type and the chunk content explicitly names Tabela 5, the primary table resolution path fires: the system matches the `table_id` from the chunk metadata against the table identifiers mentioned in the top context chunks, locates the corresponding image file, and renders it alongside the answer. Figure 5.8 shows the resolved table image as presented to the user.

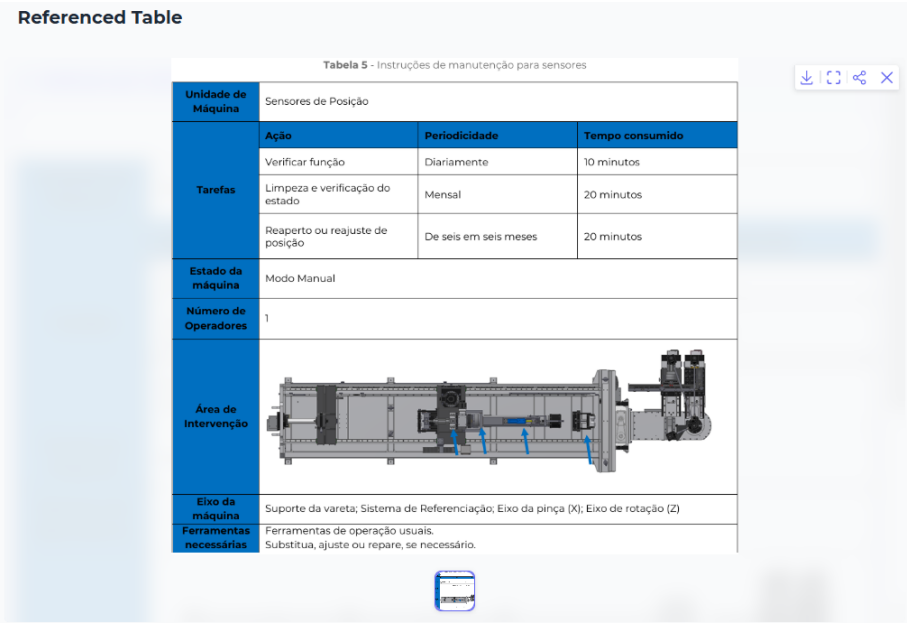

Figure 5.8: Tabela 5 resolved and rendered alongside the answer.

Ball screw lubrication

The third example revisits Q14, already examined in Section 5.1.2 as an instance of high semantic similarity paired with low keyword overlap. The question asks how frequently ball screws should be lubricated and which lubricant should be used: *Com que frequência devem ser lubrificados os fusos de esferas e que lubrificante deve ser usado?*. Figure 5.9 shows the generated answer as it appears in the chat interface.

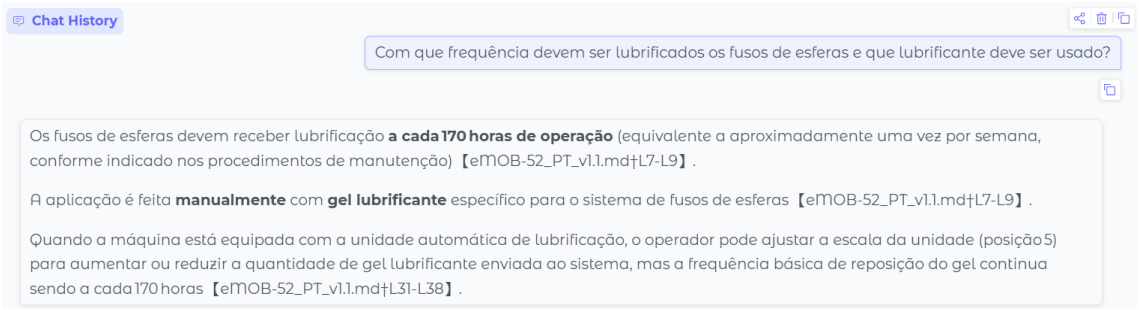

Figure 5.9: Generated answer for the ball screw lubrication question as displayed in the chat interface.

Figure 5.10 shows the top retrieved chunk, confirming that the lubrication frequency was surfaced but the specific product name and axis list were not among the chunks that reached the model, consistent with the retrieval insufficiency diagnosed in Section 5.1.2.

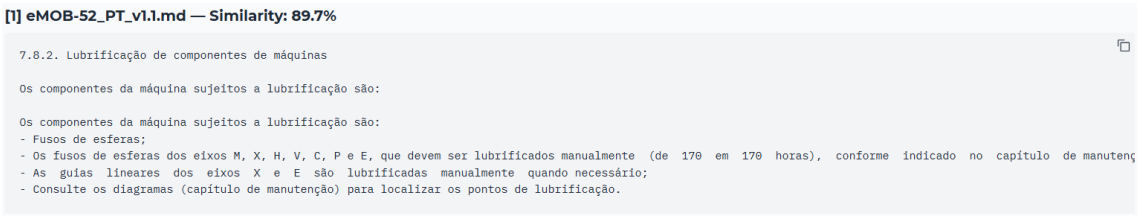

Figure 5.10: Top retrieved chunk for the ball screw lubrication question.

Electrical supply specifications

The fourth example revisits Q9, already examined in Section 5.1.2 as an instance of low semantic similarity paired with low keyword overlap and as the clearest case of retrieval or reranking insufficiency in the dataset. The question asks for the technical specifications of the electrical supply and power of the eMOB-52 machine: *Quais são as especificações técnicas de alimentação elétrica e potência da máquina eMOB-52?*. Figure 5.11 shows the generated answer as it appears in the chat interface.

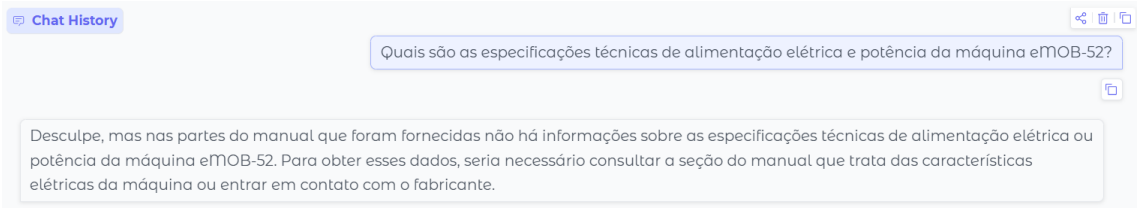

Figure 5.11: Generated answer for the electrical specifications question as displayed in the chat interface.

Figure 5.12 shows the top retrieved chunk, drawn from the machine configuration overview rather than the technical datasheet section containing the electrical parameters, confirming that the relevant specification data never reached the model, as diagnosed in Section 5.1.2.

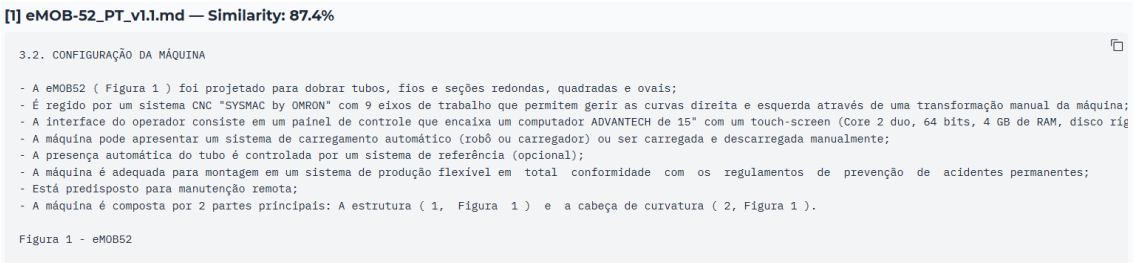

Figure 5.12: Top retrieved chunk for the electrical specifications question.

Multi-turn conversation with chat history

The fifth example demonstrates the system's use of chat history across a multi-turn exchange, splitting the content of Q19 into two consecutive questions within the same session rather than posing it as a single query. Table 5.14 shows both turns of the exchange alongside the generated answers.

Table 5.14: Multi-turn exchange splitting Q19 into two consecutive questions within the same chat session.

Turn	Generated answer
Que equipamento de proteção individual deve ser usado durante a manutenção?	Durante as operações de manutenção o operador deve usar, no mínimo, os seguintes equipamentos de proteção individual (EPI): óculos de segurança, calçados de segurança, luvas de proteção, e vestuário adequado que cumpra os requisitos das Diretivas comunitárias 89/656/CE e 89/686/CE e da legislação local. Esses EPIs são exigidos tanto nas etapas operacionais como nas trocas de ferramentas, fornecimento de carregadores, manuseio de peças e ajustes da máquina.
E que acessórios são proibidos?	Os acessórios que não podem ser usados durante as operações de manutenção são: pulseiras, relógios, anéis, colares, ou qualquer outro tipo de joia ou acessório que possa prender-se nas partes móveis da máquina. Esses itens devem ser retirados para evitar riscos de arrasto, aprisionamento ou outros perigos mecânicos.

The model correctly inferred that *acessórios* in the second turn refers to items prohibited during maintenance rather than to a new, unrelated topic, both answers being grounded in section 5.7 of the manual. This exchange illustrates that the chat history mechanism allows the system to resolve an elliptical follow-up question using the context carried over from the prior turn within the same session, producing a response equivalent in content to the single-turn Q19 answer discussed in Section 5.1.2, despite the question being split across two separate turns rather than asked as one. This behavior depends on the chat history being preserved and the two turns occurring within the same session; if the history is cleared or a new session is started, the second question would lack the context needed to resolve the reference.

5.5 Multi-model comparison

5.5.1 Methodology

The multi-model comparison evaluates whether the choice of generation model meaningfully affects system performance when all other pipeline components are held fixed. Two models available through the Groq API are compared: `openai/gpt-oss-120b`, the configured default, and `llama-3.3-70b-versatile`, run as the comparison candidate. Both models are evaluated at temperature 0.0 against the same 20 questions in the golden dataset, with retrieval restricted to the eMOB-52 manual as in the end-to-end evaluation.

The `-model` flag in `evals/run_eval.py` enables runtime model switching without modifying the `.env` file or any configuration constant. When the flag is supplied, the runner patches `LLM_MODEL` directly in `src.services.llm` and calls `get_llm.cache_clear()` to force the cached instance to rebuild with the new model on the next invocation:

Listing 5.2: Runtime model override in the evaluation CLI.

```
if args.model:
    import src.services.llm as _llm_svc
    _llm_svc.LLM_MODEL = args.model
    _llm_svc.get_llm.cache_clear()
    active_model = args.model
```

The override affects only the generation model. The embedding model (`intfloat/multilingual-e5-base`), the CrossEncoder reranker (`BAAI/bge-reranker-v2-m3`), the retrieval parameters (`RETRIEVER_K` = 50, `RERANKER_TOP_N` = 5), the similarity thresholds, the system prompt, and the golden dataset all remain identical across runs. This ensures that any difference in results is attributable to the generation model alone.

5.5.2 Results

Table 5.15 reports the aggregate metrics for both models.

Table 5.15: Aggregate evaluation results for both models.

Metric	**llama-3.3-70b-versatile**	**openai/gpt-oss-120b**
Pass rate	90% (18/20)	80% (16/20)
Average semantic similarity	93.8%	92.4%
Average keyword overlap	60.5%	51.1%
Average completeness	72.0%	85.4%
Average retrieval hit rate	100%	100%
Average latency	24.0 s	24.4 s

`llama-3.3-70b-versatile` outperforms the default model on the three gating metrics, achieving a pass rate of 90% against 80%, a higher average semantic similarity, and a notably higher average keyword overlap of 60.5% against 51.1%. The keyword overlap gap is the most diagnostic difference: llama tends to stay closer to the source document's phrasing, while `openai/gpt-oss-120b` produces more elaborate answers that paraphrase or extend the reference vocabulary. This is confirmed by the completeness scores, where the default model scores 85.4% against llama's 72.0%, indicating that `openai/gpt-oss-120b` includes more detail per answer but at the cost of vocabulary divergence from the terse reference answers.

Figure 5.13 shows the per-question breakdown across all three metrics, making the distribution of differences visible across the full dataset rather than in aggregate alone.

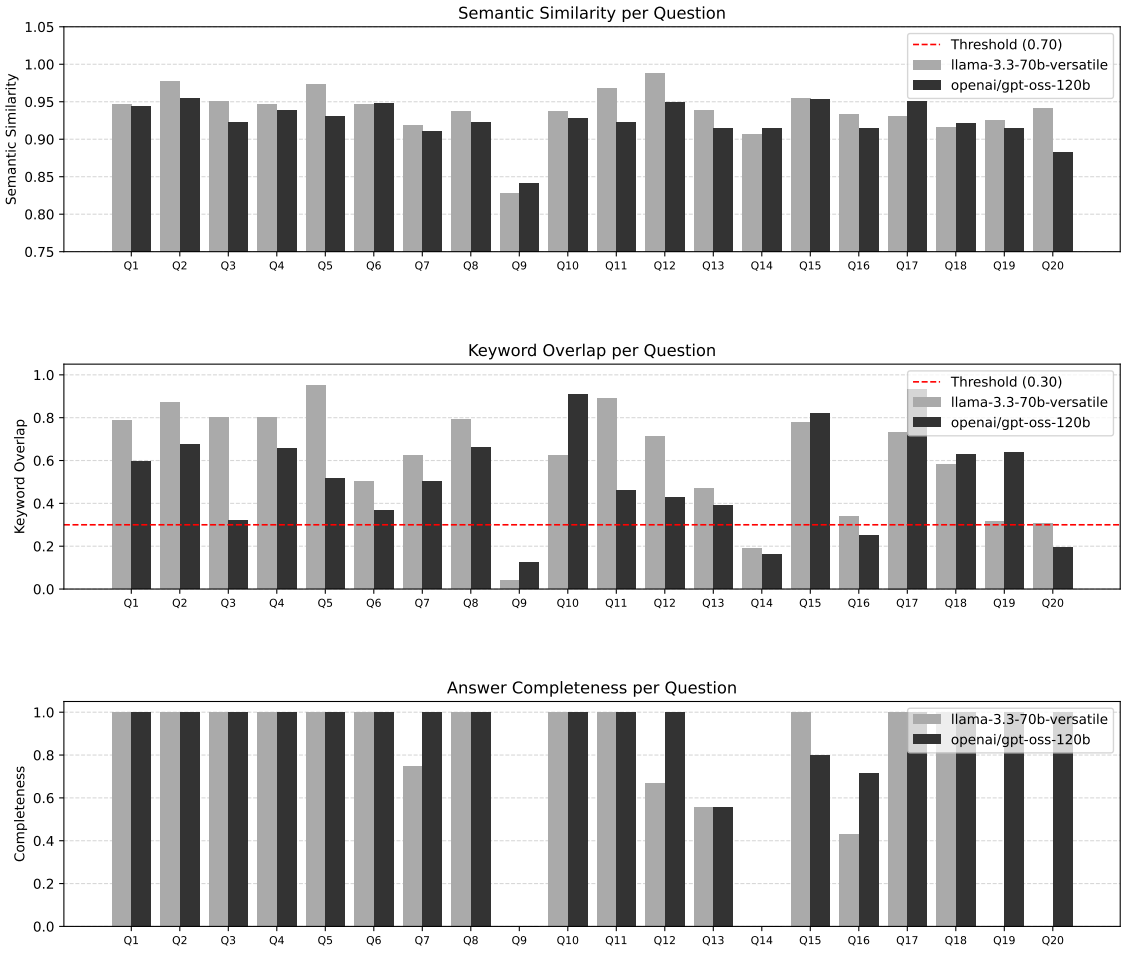

Figure 5.13: Per-question semantic similarity, keyword overlap, and answer completeness for both models. Dashed red lines mark the pass thresholds in the upper two panels.

Table 5.16 reports the exact per-question results for both models.

Table 5.16: Per-question results for both models. Sem: semantic similarity; KW: keyword overlap; P: pass.

ID	Category	llama Sem	llama KW	llama P	gpt Sem	gpt KW	gpt P
Q1	Procedural	94.7	78.8	**Yes**	94.4	59.5	**Yes**
Q2	Procedural	97.7	87.1	**Yes**	95.5	67.6	**Yes**
Q3	Procedural	95.0	80.0	**Yes**	92.2	31.9	**Yes**
Q4	Procedural	94.6	80.0	**Yes**	93.8	65.5	**Yes**
Q5	Procedural	97.3	95.1	**Yes**	93.0	51.5	**Yes**
Q6	Procedural	94.6	50.0	**Yes**	94.8	36.6	**Yes**
Q7	Procedural	91.8	62.5	**Yes**	91.1	50.0	**Yes**
Q8	Procedural	93.7	79.2	**Yes**	92.2	66.1	**Yes**
Q9	Technical specs	82.8	3.8	No	84.1	12.5	No
Q10	Technical specs	93.7	62.5	**Yes**	92.8	90.9	**Yes**
Q11	Technical specs	96.8	88.9	**Yes**	92.3	46.2	**Yes**
Q12	Technical specs	98.8	71.4	**Yes**	94.9	42.9	**Yes**
Q13	Maintenance	93.8	47.1	**Yes**	91.5	39.0	**Yes**
Q14	Maintenance	90.7	18.8	No	91.4	16.1	No
Q15	Maintenance	95.4	77.6	**Yes**	95.3	81.8	**Yes**
Q16	Maintenance	93.3	34.0	**Yes**	91.4	25.0	No
Q17	Troubleshooting	93.0	73.3	**Yes**	95.1	93.3	**Yes**
Q18	Troubleshooting	91.6	58.1	**Yes**	92.1	63.0	**Yes**
Q19	Safety	92.5	31.5	**Yes**	91.5	63.6	**Yes**
Q20	Operation	94.1	30.6	**Yes**	88.2	19.4	No

The two models disagree on two questions. Q16 is the most instructive case: llama passes with a keyword overlap of 34.0%, just above the 30% threshold, while `openai/gpt-oss-120b` fails at 25.0%. The llama answer stayed closer to the document's phrasing for the motor and encoder maintenance tasks, while the default model produced a more elaborate response that diverged from the reference vocabulary. Q20 shows a similar pattern: llama passes with a semantic similarity of 94.1% and keyword overlap of 30.6%, while the default model fails with a keyword overlap of 19.4%.

Two questions are failed by both models. On Q9, the electrical specification values were absent from the retrieved chunks despite the correct source document being returned, and both models correctly declined to answer rather than fabricating figures. On Q14, both models omitted the specific lubricant designation *Kernite NGLI 2*, driving keyword overlap below the threshold for both, with llama at 18.8% and the default model at 16.1%.

Latency is nearly identical between the two models, with llama averaging 24.0 s against 24.4 s for the default model. Given that reranking accounts for approximately 91% of total query time as shown in Section 5.1.3, the generation model has minimal influence on end-to-end latency under the current configuration.

Despite `llama-3.3-70b-versatile` outperforming the default model on pass rate, semantic similarity, and keyword overlap, `openai/gpt-oss-120b` remains the configured default. The default model produces more complete answers on average, scoring 85.4% complete-

ness against 72.0%, and its tendency toward elaboration may be preferable in deployment scenarios where an operator needs a comprehensive explanation rather than a terse reproduction of the manual's phrasing. The choice of default model is therefore a deliberate trade-off between keyword fidelity, where llama leads, and answer completeness, where the default model leads, rather than an oversight.

5.6 Observability and trace inspection

5.6.1 Trace structure and available parameters

Every query processed by the system optionally produces a LangFuse trace, gated on the presence of valid API credentials in the environment. When `LANGFUSE_PUBLIC_KEY` and `LANGFUSE_SECRET_KEY` are both set, `is_langfuse_enabled()` returns `True` and the observability layer activates. When either key is absent, every function in `src/services/observability.py` returns `None` or `False` and the query pipeline continues unaffected, with no import errors or runtime exceptions. This graceful degradation means observability is an optional instrumentation layer rather than a hard dependency.

Each trace is created at the start of `query()` with the question text and the active source filter as input fields. The environment string from `LANGFUSE_ENVIRONMENT` is written to the SDK-native `LANGFUSE_TRACING_ENVIRONMENT` variable and also attached as a tag on the trace object itself, so traces from development and production environments are filterable in the LangFuse interface both by environment field and by tag. The `trace_id` generated at this point is carried through the pipeline and returned as a field of `QueryResult`, making it available to both the evaluation runner and the Gradio frontend.

Two spans are recorded within each trace. The retrieval span captures three fields: `retrieved_count` (the number of chunks returned by the similarity search), `below_threshold` (a boolean indicating whether the top similarity score fell below `MINIMUM_SIMILARITY`), and `top_scores` (the similarity values for the top three retrieved chunks, computed as `1 - distance`). The generation span captures only `answer_length` in characters; the full answer text is logged on the trace itself rather than the span.

When the evaluation runner processes a question, five automated metric scores are posted back to the corresponding trace via `trace_id`: `semantic_similarity`, `keyword_overlap`, `retrieval_hit_rate`, `completeness`, and `passed`. This means the LangFuse interface shows the automated evaluation scores alongside the trace that produced the answer, creating a single-pane view for offline analysis without any manual cross-referencing. The score post-back is wrapped in a silent `try/except` block so LangFuse errors never interrupt evaluation.

The Gradio frontend closes a second feedback loop. The `trace_id` returned by `query()` is stored in Gradio state and attached to a five-point rating widget. When the user submits a rating,

the score is posted to LangFuse as a `user_feedback` score on the same trace and simultaneously written to `data/feedback.jsonl` with the timestamp, question, and rating label. When LangFuse is disabled, only the local file write occurs, ensuring feedback is never silently lost.

5.6.2 Trace-guided failure diagnosis

The retrieval span fields provide the primary diagnostic signal for failed questions. For a question that returns the no-match response, the `below_threshold` flag is `True` and the `top_scores` values confirm that no retrieved chunk exceeded `MINIMUM_SIMILARITY`. For a question that retrieves correctly but generates a poor answer, `below_threshold` is `False` and the `top_scores` values are high, pointing the diagnosis toward the generation stage rather than retrieval. When automated evaluation scores are also attached to the trace, the `passed` and `keyword_overlap` fields immediately identify which questions failed and on which metric, without needing to open the evaluation report separately.

Several aspects of the pipeline are not visible in LangFuse under the current instrumentation. The CrossEncoder reranking step has no dedicated span: reranker scores and reranking latency are printed locally but not captured in any trace. The generation span records only answer length, so token counts, LLM call latency, and the model name do not appear in that manually built span; this data is instead captured separately by the LangChain callback handler, which is passed into every chain invocation whenever LangFuse is enabled and automatically creates its own auto-instrumented generation observation with token usage, latency, and model name. The similarity threshold gate is recorded as a field in the retrieval span output rather than as a named span, so it does not appear as a distinct step in the trace timeline. Finally, automated metric scores are only posted during evaluation runs: a production query in the Gradio app produces a trace with a user feedback score if the user rates it, but never receives `semantic_similarity` or `keyword_overlap` scores. These gaps mean that LangFuse provides sufficient diagnostic resolution to distinguish retrieval failures from generation failures at a coarse level, but does not expose the intermediate reranking decisions that would be needed for finer-grained diagnosis of borderline cases.

Beyond diagnostic visibility, the current implementation carries two categories of unaddressed operational risk. The first concerns cost and compute exposure: no component of the codebase tracks token usage, API cost, or applies any rate limiting, timeout, or retry logic around calls to the Groq API, and neither `get_llm()` nor `get_judge_llm()` sets a `max_tokens` bound on generation. While the LangChain callback handler passed into every chain invocation does capture token usage automatically when LangFuse is enabled, this data is only ever inspected manually within the LangFuse interface; nothing in the system aggregates it into a cost figure, enforces a budget, or alerts when usage grows unexpectedly, for example due to a malformed or repeated query. The CrossEncoder reranker, which Section 5.1.3 identified as responsible for roughly 91% of query latency, runs on CPU with no device configuration or Graphics Processing Unit (GPU) acceleration path implemented, meaning the current compute footprint is not optimized and any

scaling to concurrent users would multiply an already CPU-bound bottleneck without any throttling mechanism in place to protect the service.

The second concerns the absence of content-specific safety guardrails for an industrial deployment. Beyond the similarity-based retrieval guard that triggers `NO_MATCH_RESPONSE` when no chunk exceeds `MINIMUM_SIMILARITY`, the system applies no additional verification, disclaimer, or human-in-the-loop step to answers concerning machine operation, electrical specifications, or safety procedures. A syntactically well-formed but factually incorrect answer to a question about, for instance, electrical supply voltage or a maintenance safety procedure would be returned to the operator with the same confidence and formatting as any other answer, since the pipeline has no mechanism to flag safety-critical categories for stricter verification or to append a cautionary disclaimer recommending the operator cross-check the physical manual before acting. Given that Section 5.2 already demonstrated the model's capacity to produce fabricated but plausible-looking specification values when ungrounded, and that retrieval failures such as those diagnosed for Q9 and Q14 in Section 5.1.2 show that grounding is not always guaranteed even with retrieval active, the lack of any content-aware safety layer is a meaningful gap for a system intended to support decisions on live industrial equipment.

Chapter 6

Conclusions and Future Work

6.1 Conclusions

This dissertation set out to design, implement, and evaluate AMOB-RAG, a Retrieval-Augmented Generation system for querying the technical documentation of industrial CNC tube bending machines in natural language. The work was structured around four objectives defined in Section 1.3, and this section revisits each of them in light of the results obtained.

The first objective concerned the collection and preprocessing of AMOB's eMOB-52 documentation into a form suitable for ingestion. The source manual was converted from PDF to Markdown, manually curated to correct conversion artifacts, and processed through an ingestion pipeline that introduced several design decisions tailored to the structure of industrial technical documentation. Rather than applying a single uniform segmentation strategy, the pipeline distinguishes between prose blocks, procedural lists, and tabular content, treating each differently. Prose blocks are semantically chunked using embedding-based boundary detection, ensuring that topical coherence is preserved without relying on arbitrary size thresholds in size-based chunking. Procedural lists are preserved as atomic blocks, keeping numbered steps intact so that retrieval never surfaces an incomplete sequence of instructions. Markdown tables are extracted at the row level and converted into natural-language sentences using a domain-specific Portuguese template, making tabular specifications directly searchable by embedding similarity.

The second objective was the design of a query pipeline that minimizes context loss and hallucination risk. The baseline experiment, in which the same model answered the dataset without retrieval, passed none of the twenty questions and saw keyword overlap collapse from 51.1% to 4.8%, while still producing fluent and confident answers. This demonstrates that the language model alone cannot ground its responses in proprietary documentation, and that retrieval is the component that suppresses confident but unfounded answers rather than an incremental enhancement, proving that the retrieval step is not a peripheral addition to the pipeline but the fundamental mechanism that makes correct and grounded LLM answering possible in domain-specific settings. On the query side, Portuguese lemmatization and domain-specific synonym expansion were introduced to bridge the vocabulary gap between operator queries and indexed documentation, ensuring

that terms resolve to the same content regardless of which name the operator uses, important for keyword matching efficiency.

The third objective was the development of an interesting UI exposing the pipeline to end users. This was realized as a chat interface that allows natural language querying, restricts retrieval to a selected machine's documentation, and automatically resolves and displays the tables and figures referenced in each answer, alongside a feedback mechanism that records user ratings for later analysis and constant system improvement. A consistent finding across the evaluation was that the system's failures were attributable to generation-side vocabulary divergence rather than retrieval failures, since the correct source document was present in the retrieved chunks for every question in the dataset, confirming that the application reliably surfaces the right content and that remaining gaps are localized to generation behavior rather than the retrieval or interface layer. The principal limitation identified was latency, which averaged 24.4 seconds per query and is dominated by the CrossEncoder reranking stage at approximately 91% of total query time, representing the most concrete target for improvement in a future production deployment.

The fourth objective concerned on the validation of the case study against real-world usage. The system was evaluated against a golden dataset of twenty verified question-answer pairs grounded in the eMOB-52 manual, covering procedural, technical specification, maintenance, troubleshooting, safety, and operation query types, with results demonstrating strong retrieval reliability and generation faithfulness across all configurations tested. Beyond the automated evaluation, the application was demonstrated to and validated by the AMOB professional designated to work with the system in its deployment phase, whose initial feedback was positive, expressing confidence in the system's practical utility for day-to-day documentation queries. Security and response reliability are recognized as critical requirements for any system deployed in an industrial environment, and the prototype addresses these concerns at this stage by grounding every answer exclusively in the retrieved documentation, declining to respond when the retrieved context does not support an answer, and operating within the behavioral constraints defined in the system prompt. This early validation confirms that the system's behavior is recognizable and useful to the domain expert it was designed to serve, while laying a responsible foundation for the deployment phase ahead.

Taken together, these results show that a carefully designed RAG system can serve as a reliable and practically useful interface to dense industrial documentation, grounding model outputs in verified source material and offering a credible alternative to manual navigation of lengthy technical manuals. The design choices introduced in this work, from atomic list preservation to row-level table extraction and domain-specific query expansion, each addressed a concrete failure mode that a generic RAG implementation would not have resolved, and their combined effect is visible in the evaluation results. The work also demonstrated that evaluation methodology matters in this domain: combining retrieval-level metrics, generation-level metrics, and qualitative scoring across multiple model configurations revealed failure modes with enough precision to trace them back to specific pipeline stages, making the evaluation framework itself a reusable contribution for future work in this area. Among these, the LLM-as-a-judge component stands out as

a particularly promising direction, since its ability to trace a claim back to the specific chunk that supports it offers a form of explainability that deterministic metrics cannot provide, and refining its configuration to reduce excessive permissiveness could make it a central diagnostic tool in future iterations of the system.

6.2 Future work

Modular document ingestion interface Adding a drag-and-drop interface for adding, replacing, and removing source documents through the frontend.

Improved table handling Refining the row-level table extraction responsible for the dataset's chunking-related false negatives, so that specification tables are retrieved and reproduced with their full content.

Query-Question expansion Generating a synthetic answer with the language model and embedding it in place of the raw question to improve recall when query and document vocabulary diverge.

Multi-Hop reasoning Extending the pipeline to retrieve and refine context over several iterations for queries that require combining information from non-adjacent sections of the documentation.

Observability dashboards Configuring LangFuse dashboard views that aggregate the per-trace scores over time, segment by query category, and surface latency outliers for further production monitoring.

Vision language models Feeding the already-resolved figure and table images to a vision language model so the system can reason over diagrams and drawings rather than only displaying them alongside the answer.

Stale-index auto-detection Automating the re-ingestion process so that changes to the chunking configuration or source documents trigger an automatic update of the ChromaDB index without requiring manual intervention.

Prompt engineering Extending the system prompt beyond its current minimal template to include explicit behavioral rules, domain-specific constraints, and safety guidelines that normalize response structure and ensure the model consistently declines to speculate beyond the retrieved context.

Deployment Transitioning the application from a local development environment to a production-ready deployment that runs continuously, making the system available to AMOB operators around the clock without requiring manual startup or maintenance.

References

[1] AMOB Group. *Sobre Nós*. https://amobgroup.com/pt-pt/sobre-nos. Accessed: 2026-04-16.

[2] Zohaib Jan et al. "Artificial intelligence for industry 4.0: Systematic review of applications, challenges, and opportunities". In: *Expert Systems with Applications* 216 (2023), p. 119456. DOI: 10.1016/j.eswa.2022.119456.

[3] Samuel Kernan Freire et al. "Knowledge sharing in manufacturing using LLM-powered tools: user study and model benchmarking". In: *Frontiers in Artificial Intelligence* 7 (2024), p. 1293084. DOI: 10.3389/frai.2024.1293084.

[4] Silvia Colabianchi, Francesco Costantino, and Nicolò Sabetta. "Assessment of a large language model based digital intelligent assistant in assembly manufacturing". In: *Computers in Industry* 162 (2024), p. 104129. DOI: 10.1016/j.compind.2024.104129.

[5] Maria Nunes e Almeida. "Study of Tube Rotary Draw Bending Process: Experimental and Numerical Analysis for Different Metallic Materials". Accessed: 2026-07-30. Master's Thesis. Faculdade de Engenharia da Universidade do Porto, 2025. URL: https://repositorio-aberto.up.pt/bitstream/10216/168852/2/735646.pdf.

[6] Rui L. Amaral et al. "Analysis of Springback in Rotary Draw Bending Process: Numerical Modelling with Experimental Validation". In: *MATEC Web of Conferences*. Vol. 408. 2025, p. 01077. DOI: 10.1051/matecconf/202540801077.

[7] Omni-X. *Rotary Draw Bending*. 2024. URL: https://www.omni-x.com/rotary-draw-bending (visited on 02/11/2025).

[8] Linda Borchmann et al. "Control of Material Flow Using Measuring Methods for Wrinkle and Crack Detection During Rotary Draw Bending". In: *Procedia CIRP* 118 (2023), pp. 857–862. ISSN: 2212-8271. DOI: 10.1016/j.procir.2023.06.147.

[9] Cristiano Mosqueira Gomes. "Finite Element Analysis of Flexible Pipes: Bending Combined with Tensile Load". Programa de Engenharia Civil. PhD Thesis. Rio de Janeiro, Brazil: Universidade Federal do Rio de Janeiro, COPPE, Oct. 2017.

[10] Juha Tulonen and SSAB. *Precision Steel Tube Handbook*. 2nd. Compilation of training material for bending, welding, and coating. SSAB. 2017.

[11] Ashish Vaswani et al. "Attention Is All You Need". In: *Advances in Neural Information Processing Systems*. Vol. 30. Accessed: 2026-07-30. 2017. URL: https://arxiv.org/abs/1706.03762.

[12] Tom Brown et al. "Language Models are Few-Shot Learners". In: *Advances in Neural Information Processing Systems*. Vol. 33. Accessed: 2026-07-30. 2020, pp. 1877–1901. URL: https://arxiv.org/abs/2005.14165.

[13] Long Ouyang et al. “Training Language Models to Follow Instructions with Human Feedback”. In: *Advances in Neural Information Processing Systems*. Vol. 35. Accessed: 2026-07-30. 2022, pp. 27730–27744. URL: https://arxiv.org/abs/2203.02155.

[14] Hugo Touvron et al. “LLaMA: Open and Efficient Foundation Language Models”. In: *arXiv preprint arXiv:2302.13971* (2023). Accessed: 2026-07-30. URL: https://arxiv.org/abs/2302.13971.

[15] OpenAI. *gpt-oss-120b & gpt-oss-20b Model Card*. Accessed: 2026-07-30. 2025. arXiv: 2508.10925 [cs.CL]. URL: https://arxiv.org/abs/2508.10925.

[16] Jason Wei et al. “Chain-of-Thought Prompting Elicits Reasoning in Large Language Models”. In: *Advances in Neural Information Processing Systems*. Vol. 35. Accessed: 2026-07-30. 2022, pp. 24824–24837. URL: https://arxiv.org/abs/2201.11903.

[17] Lei Huang et al. “A Survey on Hallucination in Large Language Models: Principles, Taxonomy, Challenges, and Open Questions”. In: *ACM Transactions on Information Systems* 43.2 (2025), pp. 1–55. DOI: 10.1145/3703155.

[18] Nils Reimers and Iryna Gurevych. “Sentence-BERT: Sentence Embeddings using Siamese BERT-Networks”. In: *Proceedings of the 2019 Conference on Empirical Methods in Natural Language Processing*. Accessed: 2026-07-30. Association for Computational Linguistics, 2019, pp. 3982–3992. URL: https://arxiv.org/abs/1908.10084.

[19] Aparajitha Allamraju et al. “Breaking It Down: Domain-Aware Semantic Segmentation for Retrieval Augmented Generation”. In: *arXiv preprint arXiv:2512.00367* (2024). Accessed: 2026-07-30. URL: https://arxiv.org/abs/2512.00367.

[20] Yunfan Gao et al. “Retrieval-Augmented Generation for Large Language Models: A Survey”. In: *arXiv preprint arXiv:2312.10997* (2023). Accessed: 2026-07-30. URL: https://arxiv.org/abs/2312.10997.

[21] Arnau Perez and Xavier Vizcaíno. “Advanced ingestion process powered by LLM parsing for RAG system”. In: *arXiv preprint arXiv:2412.15262* (2024). DOI: 10.48550/arXiv.2412.15262.

[22] Patrick Lewis et al. “Retrieval-Augmented Generation for Knowledge-Intensive NLP Tasks”. In: *Advances in Neural Information Processing Systems*. Vol. 33. Accessed: 2026-07-30. 2020, pp. 9459–9474. URL: https://arxiv.org/abs/2005.11401.

[23] Rodrigo Nogueira and Kyunghyun Cho. “Passage Re-ranking with BERT”. In: *arXiv preprint arXiv:1901.04085* (2019). Accessed: 2026-07-30. URL: https://arxiv.org/abs/1901.04085.

[24] Aleksei Dorkin and Kairit Sirts. “GliLem: Leveraging GliNER for Contextualized Lemmatization in Estonian”. In: *Proceedings of the Joint 25th Nordic Conference on Computational Linguistics and 11th Baltic Conference on Human Language Technologies*. Accessed: 2026-07-30. University of Tartu Library, 2025, pp. 86–97. URL: https://arxiv.org/abs/2412.20597.

[25] Shahul Es et al. “RAGAS: Automated Evaluation of Retrieval Augmented Generation”. In: *Proceedings of the 18th Conference of the European Chapter of the Association for Computational Linguistics: System Demonstrations*. Accessed: 2026-07-30. Association for Computational Linguistics, 2024, pp. 150–158. URL: https://arxiv.org/abs/2309.15217.

[26] Lianmin Zheng et al. “Judging LLM-as-a-Judge with MT-Bench and Chatbot Arena”. In: *Advances in Neural Information Processing Systems*. Vol. 36. Accessed: 2026-07-30. 2023. URL: https://arxiv.org/abs/2306.05685.

[27] Liming Dong, Qinghua Lu, and Liming Zhu. “AgentOps: Enabling Observability of LLM Agents”. In: *arXiv preprint arXiv:2411.05285* (2024). Accessed: 2026-07-30. URL: https://arxiv.org/abs/2411.05285.

[28] Docling Project. *Docling: Get Your Documents Ready for Gen AI*. https://github.com/docling-project/docling. Open-source document conversion toolkit. Accessed: 2026. 2024.

[29] Chroma. *Chroma: The AI-Native Open-Source Embedding Database*. 2024. URL: https://www.trychroma.com (visited on 06/01/2026).

[30] Paul Jaccard. “The Distribution of the Flora in the Alpine Zone”. In: *New Phytologist* 11.2 (1912), pp. 37–50. DOI: 10.1111/j.1469-8137.1912.tb05611.x.

[31] Viviane Moreira Orengo and Christian Huyck. “A Stemming Algorithm for the Portuguese Language”. In: *Proceedings of the Eighth Symposium on String Processing and Information Retrieval (SPIRE)*. 2001, pp. 186–193. DOI: 10.1109/SPIRE.2001.10024.

[32] Langfuse. *Langfuse: Open Source LLM Engineering Platform*. https://github.com/langfuse/langfuse. Open-source LLM observability and evaluation platform. Accessed: 2026. 2024.

[33] Confident AI. *DeepEval: The Open-Source LLM Evaluation Framework*. Accessed: 2026-07-30. 2024. URL: https://github.com/confident-ai/deepeval.

[34] Gradio Team. *Gradio: Build Machine Learning Web Apps in Python*. https://github.com/gradio-app/gradio. Open-source UI framework for machine learning applications. Accessed: 2026. 2023.

Appendix A

Golden Dataset Questions

This appendix lists the twenty questions that make up the golden dataset described in Section 3.4.1 and evaluated in Section 5.1. The questions are reproduced verbatim in Portuguese, the language in which the dataset was written and in which the system is queried, and are presented in the same order and with the same identifiers used throughout the evaluation, so that each entry can be matched directly against the per-question results reported in Table 5.3. All twenty entries are grounded exclusively in the documentation of the eMOB-52 rotary draw bending machine and carry the `verified: true` flag, meaning that each question and its corresponding reference answer were manually reviewed against the source manual. The reference answers themselves are not reproduced here, as several of them span structured multi-step procedures and dense specification enumerations that would not survive tabular presentation.

Table A.1: The twenty questions of the golden dataset, in evaluation order.

ID	Category	Question
Q1	Procedural	Quais são os passos para trocar a matriz?
Q2	Procedural	Quais são os passos para substituir a vareta?
Q3	Procedural	Quais são os passos para ajustar a posição da mordaça?
Q4	Procedural	Quais são os passos para ajustar a paralela?
Q5	Procedural	Como se substitui o lubrificante da caixa de velocidades do eixo de curvatura?
Q6	Procedural	Qual é o procedimento de teste de rotina do botão de paragem de emergência?
Q7	Procedural	Quais são os passos para substituir a bateria do CPU da série NJ?

continued on next page

Table A.1 (continued)

ID	Category	Question
Q8	Procedural	Que operações preliminares devem ser realizadas antes de manobrar a máquina?
Q9	Technical specs	Quais são as especificações técnicas de alimentação elétrica e potência da máquina eMOB-52?
Q10	Technical specs	Quais são as velocidades máximas dos eixos da máquina eMOB-52?
Q11	Technical specs	Quais são os eixos de trabalho servocontrolados da máquina e a função de cada um?
Q12	Technical specs	Qual é a vida útil das diferentes peças do Servo Motor série G5 modelo R88M-K?
Q13	Maintenance	Que tarefas de manutenção são definidas para os sensores de posição e com que frequência?
Q14	Maintenance	Com que frequência devem ser lubrificados os fusos de esferas e que lubrificante deve ser usado?
Q15	Maintenance	Que tarefas de manutenção devem ser executadas no quadro elétrico sem tensão e com que frequência?
Q16	Maintenance	Que cuidados e verificações são definidos para motores e codificadores?
Q17	Troubleshooting	Quais são as causas prováveis e soluções para rugas de curvatura internas?
Q18	Troubleshooting	Quais são as causas prováveis e soluções para quebras de tubo durante a curvatura?
Q19	Safety	Que equipamento de proteção individual deve ser usado e que acessórios são proibidos durante a manutenção?
Q20	Operation	Quais são os três modos de funcionamento da máquina e como se distinguem?