POLITECNICO
MILANO 1863

SCUOLA DI INGEGNERIA INDUSTRIALE
E DELL'INFORMAZIONE

MathRAG: An Agentic Question Answering Architecture for Mathematical Problems based on Structured Knowledge Graphs

Tesi di Laurea Magistrale in
Computer Science and Engineering - Ingegneria Informatica

Davide Morelli, 10962902

Abstract: Large Language Models (LLMs) have demonstrated strong capabilities in natural language understanding and reasoning, yet they remain unreliable in mathematical problem solving due to hallucinations, arithmetic inconsistencies, and limited transparency in intermediate reasoning steps. This thesis proposes a structured Retrieval-Augmented Generation framework, termed *MathRAG*, designed to improve reliability and precision in mathematical question answering by integrating knowledge graph retrieval with deterministic tool execution.
The proposed system extracts formulas from domain-specific documents, converts them into structured entities enriched with semantic metadata, and stores them in a knowledge graph. Upon receiving a user query, the framework performs semantic retrieval over embedded formula representations, selects the most relevant formula, and dynamically synthesizes a computational tool from a validated Python template. Numerical computation is then executed deterministically, separating symbolic reasoning from language generation and thereby reducing hallucination risks. This whole process of question answering is encapsulated inside a ReAct-style reasoning loop, in order to take advantage of the CoT-reasoning ability of the language model as much as possible.
The system is evaluated on a dataset of 600 mathematical questions spanning Physics and Chemistry domains. Results show that the primary bottleneck lies in tool recognition and retrieval rather than in numerical execution. While overall accuracy across the full dataset reaches 28%, conditional accuracy rises above 58% when the correct tool is instantiated, demonstrating the robustness of deterministic computation once the appropriate formula is selected. A domain-level analysis reveals a significant performance gap between the two questions' domains, highlighting the impact of formula structure, metadata consistency, and semantic alignment on retrieval effectiveness.
The findings confirm that combining structured knowledge representation with executable tools substantially improves computational reliability and interpretability compared to purely generative approaches. However, system performance remains strongly dependent on retrieval precision, metadata quality, and ingestion robustness. The thesis concludes that hybrid knowledge-grounded architectures such as MathRAG constitute a promising direction for trustworthy mathematical reasoning systems, provided that future work further enhances semantic matching, document standardization, and knowledge graph consistency.

Advisor:
Prof. Marco Brambilla

Co-advisors:
Riccardo Campi
Mathyas Giudici

Academic year:
2024-2025

Key-words: RAG, graphRAG, mathematical formulas, ReAct, tool calling, CoT reasoning, knowledge extraction

1. Introduction

Large Language Models (LLMs) in recent years have demonstrated remarkable capabilities across a wide range of natural-language understanding and generation tasks, including question answering, summarization, and conversational assistance. Their rapid adoption has also extended to domains traditionally dominated by symbolic or rule-based systems, such as scientific reasoning, technical consulting, and mathematical problem solving.
However, despite their expressive power, LLMs remain fundamentally probabilistic systems, a characteristic that poses significant challenges when correctness, reproducibility, and verifiability are required. This limitation is particularly evident in mathematical question-answering, where even minor inconsistencies or arithmetic inaccuracies produced by hallucinations can invalidate an otherwise plausible response [1].
The lack of trustworthiness in LLM-generated answers for these exact reasons can prove to be a critical obstacle to their deployment in knowledge-intensive real-world applications.
Previous work has shown that LLMs are prone to hallucinate formulas, misapply known equations, or perform unreliable arithmetic reasoning, even when the underlying problem is conceptually simple [1]. Moreover, these shortcomings are exacerbated when the interrogation from the user covers newly discovered mathematical knowledge or the application of formulas coming from very specific and niche domains.
From a user-interaction perspective, this creates a fundamental challenge: how can systems expose the flexibility and accessibility of LLM-driven interfaces while guaranteeing the correctness of mathematically grounded answers? Trying to find a solution to this challenge is at the core of this thesis.
Retrieval-Augmented Generation (RAG) and Graph-based RAG (GraphRAG) architectures have been presented as a partial mitigation to hallucinations in language models by grounding responses in external knowledge [8, 9]. Although classical RAG approaches improve factual consistency, they remain inadequate for mathematical reasoning tasks, where correctness depends not only on retrieving relevant information but also on applying it through precise and deterministic computational procedures.
Even in GraphRAG systems, with the introduction of structured knowledge representations in order to enable richer reasoning over entities and relationships, these approaches typically terminate at information retrieval and continue to rely on the language model to perform the final calculation, thereby reintroducing the very sources of error they seek to eliminate.
This thesis addresses these limitations by proposing **MathRAG**, a graph-augmented, agent-based reasoning framework explicitly designed for trustworthy mathematical question answering.
The central idea is to decouple the retrieval of mathematical knowledge from the execution of mathematical formulas. Instead of prompting a language model to implicitly infer formulas and subsequently carry out the calculations, MathRAG stores formalized mathematical knowledge, expressed as RDF-encoded formulas, constants, and parameter schemas, within a structured knowledge graph. An autonomous agent orchestrates the interaction between the user query, the knowledge graph, and the language model through a ReAct-style reasoning loop, dynamically synthesizing and invoking deterministic computational tools when required.
This approach is based on the observation that a large class of mathematical problems can be reduced to a well-defined mapping of the form

$$\textit{input values} + \textit{known formula} \rightarrow \textit{exact output}$$

By externalizing the *known formulas*, extracting the *input values* from the user message, and executing them through symbolic means, MathRAG is able to provide the *exact output* and mitigate the phenomenon of mathematical hallucination while preserving a natural-language interaction paradigm. The agent explicitly reasons about when to retrieve knowledge, when to construct executable tools, and when to invoke them, all while keeping track of episodic states in order to enforce coherent workflows and prevent invalid action sequences.
MathRAG is presented with the goal of being a domain-agnostic system, and this is reflected in the ontology that it uses which is purposefully simple enough to be able to adapt to as many use cases as possible.
Formulas are stored and linked only to the *Chunks* of the *Documents* they were extracted from as well as to the *Constants* they use in their calculations. Formulas can also be linked to other Formulas they use, are derived from, are specializations of or generalizations of. By leveraging this simple structure, the system is able to adapt to whatever domain the final user whishes to infer on, from physics to algebra, from Calculus to the calculation of energy-consumption metrics derived from technical documentations.
The main contributions of this thesis are threefold:

- the development of a formula-extraction pipeline that takes in input documents and process them in order to extract as many mahematical informations as possbile as well as the design of the underlying knowledge representation structure that supports the whole framework;
- the development of an agent-based reasoning architecture that integrates GraphRAG with dynamic tool synthesis and deterministic computation;
- an empirical evaluation demonstrating improved reliability over baseline language models on both general mathematical queries and tasks known to expose systematic LLM failures, such as modular arithmetic.

2. Background and Related Works

2.1. Language Models and their limitations

Since their introduction, Large Language Models (LLMs) have seen a significant expansion in both their scale and range of applications.
As their reasoning capabilities improved, they received increasing attention from both academic fields and industrial ones, to the point that it is not rare today to see them deployed even in applications that concern very specific domains of application.
Early studies demonstrated the capabilities of LLMs in solving a wide range of arithmetic and algebraic problems when presented in natural language form, in particular when these resembled cases seen during training. This initial success further motivated the development of specialized benchmarks and fine-tuned models aimed at improving performances in mathematically intensive domains [10].
Despite these advancements, subsequent research has highlighted several fundamental limitations. LLMs often struggle with tasks that require strict adherence to formal mathematical rules, precise arithmetic, and symbolic manipulation [22].
These difficulties become especially evident in problems involving multi-step derivations, when intermediate results have to be stored and propagated unchanged across multiple successive reasoning steps [1].
The underlying cause of these problems and limitations can be traced back to the statistical nature of LLMs. While such a nature makes these models well suited for natural language tasks where approximation and redundancy are often acceptable, it makes them fundamentally misaligned with the deterministic nature of mathematics, where ambiguity is not tolerated, and correctness is strongly binary.
Within this context, the phenomenon commonly referred to as *hallucinations* has been widely discussed.
When it comes to factual question answering, hallucinations typically manifest as the invention of external non-existing facts or unsupported claims [14, 17]. In the mathematical context, however, hallucinations can take on another form.
Models may apply valid formulas outside their domain of applicability, combine multiple correct identities into a composite invalid transformation, or silently violate some underlying assumption required for a derivation [18].
All of these behaviors highlight a deeper structural challenge: LLMs often struggle to maintain full global logical context across an entire solution.
Previous research has shown that such hallucinations are particularly pronounced in tasks involving modular arithmetic, symbolic algebra and large numerical values [6, 13, 20]. All of these settings require exact computation, careful handling of constraint and the ability to strictly enforce rules. All things that are not naturally supported by probabilistic text generation.
What emerges is that these problems do not constitute rare edge cases but rather systematic failure modes that undermine significantly the model's trustworthiness in mathematically grounded applications. As a consequence, increasing research efforts have been directed toward solutions and methods that extend further beyond the model themselves, bringing to the development and introduction of external mechanisms and architectural changes to address these reasoning limitations [15].
Early attempts to mitigate these reasoning problems in Language Models focused primarily on prompt engineering and other prompt-based solutions. Among these, Chain-of-Thought (CoT) prompting was proposed as a method to encourage LLMs to explicitly generate intermediate reasoning steps rather than directly producing a final answer [19]. Through this externalized reasoning process, empirical evidence showed that models could achieve higher performances on certain multi-step reasoning tasks.
However it quickly became evident that, while such techniques can enhance transparency and sometimes improve performance, they do not fundamentally change the reasoning mechanisms at the core of language models. The generated reasoning chains remain probabilistic in nature and therefore remain susceptible to hallucinations and logical inconsistencies.
Moreover prompt-based solutions impose a significant burden on prompt design. They require careful and task-specific tuning that does not scale well across diverse problem domains. These limitations ultimately led researchers to explore new approaches that incorporate external reasoning and computation components into LLM-based systems.

2.2. RAG and graph-RAG as possible solutions

Retrieval-Augmented-Generation (RAG) architectures were introduced as a means of grounding language models' responses in external knowledge sources without requiring retraining of the underlying model. One of the key motivations for RAG lies in addressing the so called *closed-world* assumption of pre-trained language models,

which cannot reliably access information that was not present during its training [9]. This specific limitation is particularly evident in mathematical tasks involving formulas, identities, or domain-specific laws that the model has not encountered before, leading to a significant decrease in accuracy.
In a typical RAG setup, relevant documents are retrieved from a corpus based on the user's input query and provided to the model as additional context when generating the output answer. This design allows the model to leverage up-to-date and domain-specific knowledge, effectively decoupling knowledge storage from language generation.
RAG architectures have therefore been widely adopted in question answering systems, conversational agents, and knowledge-intensive applications [8].
Despite their success, RAG systems exhibit notable limitations when it comes to mathematical knowledge [11]. Retrieved documents are often unstructured and may contain multiple formulas, alternative derivations, or informal explanation. The responsibility for identifying the correct formula and applying it appropriately in this way is once again left entirely to the LLM. As a result, the core problem of hallucinations is only partially addressed, as incorrect selection or misuse of the retrieved information can still lead to invalid results.
To address and overcome the limitations imposed by unstructured retrieval, recent research has increasingly focused on structured knowledge representation. Graph-based Retrieval-Augmented-Generation (GraphRAG) extends the traditional RAG paradigm by implementing knowledge graphs instead of plain documents to store the external information [5].
In this setting, information is stored as structured entities and relations, enabling for more precise retrieval and clearer semantic interpretation. By organizing the knowledge in graph form, GraphRAG systems reduce the risk of retrieving irrelevant or misleading context and allow for finer-grained control over the applicability of the retrieved information.
This structured approach is particularly well suited for the mathematical domain, where relationships between concepts, formulas and constraints can be explicitly modeled.
While GraphRAG significantly improves the quality and precision of retrieval, it does not by itself guarantee correct application of the retrieved knowledge, thus leaving room for further architectural improvements.

2.3. External computation and Agent-based orchestration

The integration of external computation tools into LLM-based systems represents another significant step toward reliable reasoning. Tool calling, also referred to as function calling, allows language models to delegate specific tasks, such as numerical computation, database querying, or symbolic manipulation, to external functions with well-defined and deterministic behaviors [7].
Tool-augmented systems emerged from the recognition that certain classes of tasks are inherently better handled by deterministic algorithms rather than by language models.
In the mathematical context, tools can perform exact calculations, enforce formal constraints and finally return verifiable results. This enables a clear separation of responsibilities: the tool ensures computational correctness while the LLM focuses on interpretation and high-level reasoning. Nevertheless, tool integration introduces its own challenges. The model must still recognize when a specific tool is needed and how to invoke it correctly.
Without a fine orchestration work, tools may be underutilized, misapplied, or invoked at inappropriate stages of the reasoning process. These challenges have motivated the development of agent-based frameworks that explicitly manage interactions between LLMs and external components.
Another advancement in this direction is represented by the ReAct paradigm [21]. This reasoning mechanism forces the language model to interleave reasoning steps with actions and observations. Under this paradigm, the model periodically halts text generation to issue the execution of an external action such as querying a knowledge base or invoking a computational tool. Control is temporarily transferred to an agent that executes the requested action and returns the result to the model as an observation, which is then incorporated into the subsequent reasoning process.
This explicit-loop method contrasts with earlier approaches where tool usage was either hard-coded or loosely integrated.
In an agent-based system, the control flow is coordinated by an orchestrator. This component decides when to override the LLM, perform retrieval or computation, return the result to the model as observation, and finally when to give back the control to the model to produce the final answer for the user to see [21].
This specific architecture is particularly effective for mathematical problem solving, where exactness and strict adherence to rules are essential. Collectively, these research developments cited above have given rise to a class of knowledge-enhanced LLM systems that aim to mitigate hallucinations by grounding model outputs in external sources and enforcing correctness through deterministic components and strict reasoning procedures.
While many existing works in this area explore symbolic reasoning, knowledge graphs and tool integration to improve factual accuracy, relatively few explicitly target the unique challenges of mathematical reasoning in this sense. Existing systems often rely on passive information retrieval or limited tool integration, leaving significant

room for improvement.
The approach proposed in this thesis builds upon these research directions while aiming to advance the state of the art. Unlike traditional RAG systems, it relies on a structured RDF-based knowledge graph to store mathematical formulas, definitions and constraints. Retrieval is performed at the level of formal knowledge in the form of formula templates, rather than unstructured text.
Furthermore, the proposed system integrates GraphRAG with a ReAct-style agent that actively orchestrates reasoning, retrieval and computation. By dynamically synthesizing deterministic tools from retrieved knowledge and delegating execution to them, the approach minimizes the contribution of the LLM in exact computation, limiting it to the role of producing the thought pattern that guides the solution process.
This design directly targets known hallucination failure modes in mathematical reasoning and distinguishes the proposed work from prior knowledge-enhanced LLM systems.

2.4. Related Works

Early work on math-aware information retrieval focused on the extraction and structural representation of formulas from scientific documents, enabling search over operator trees and spatial symbol relations rather than plain text (e.g., Tangent-v) [4] and large-scale pipelines for formula detection and parsing from PDFs [16].
A fundamental enabling technology for large-scale mathematical knowledge extraction from documents is Optical Character Recognition (OCR). OCR refers to the automated conversion of scanned or image-based documents into machine-readable text. In the context of scientific and technical material, OCR systems must go beyond plain character recognition and correctly preserve structural elements such as section headers, mathematical expressions, and layout information. Accurate OCR is therefore a prerequisite for any downstream pipeline that aims to extract formulas, symbols, and semantic relationships from PDF documents in a reliable and scalable manner.
Recent advances in OCR for scientific documents have led to models that directly convert PDFs into structured Markdown representations. Marker[1] is one such system, designed specifically for high-fidelity PDF-to-Markdown conversion. Unlike traditional OCR pipelines that primarily focus on character-level recognition, Marker preserves document structure, including headings, paragraphs, tables, and mathematical expressions.
Internally, the system integrates multiple specialized deep learning components. A layout analysis model first segments each page into semantic regions and determines reading order. An OCR component processes scanned or image-based content to ensure compatibility with non-digitally-born PDFs. A dedicated mathematical recognition model reconstructs equations and outputs syntactically valid LaTeX expressions. In addition, a table recognition module infers row and column structure to accurately reproduce tabular content in Markdown format. These intermediate outputs are then consolidated into a unified, hierarchically structured representation of the document.
This structured output is particularly suitable for subsequent semantic processing and knowledge graph construction, as it enables section-aware segmentation and more reliable formula extraction.
More recently, the limitations of pure chain-of-thought reasoning for precise computation have motivated hybrid neuro-symbolic methods in which language models delegate execution to external programs. Program-Aided Language Models (PAL) and Program-of-Thoughts prompting generate executable code for mathematical operations, effectively decoupling symbolic computation from natural-language reasoning and yielding large gains on arithmetic and algebraic benchmarks [3, 7].
Similarly, ReAct-style frameworks interleave reasoning and tool invocation, allowing models to retrieve relevant knowledge or call Python code-interpreters during multi-step problem solving [21]. These approaches demonstrate that explicit action traces and environment interaction significantly improve faithfulness and numerical accuracy in mathematical question answering. Together, these lines of work indicate that combining structured knowledge representations, math-aware retrieval, and deterministic tool implementation is a promising direction for building reliable LLM-based systems for formal reasoning.

3. Main Idea of the Thesis

3.1. Overview and Design Principles

The work presented in this thesis is centered around the development of a system designed to assist language models in handling mathematical question answering more reliably and accurately. The core objective in this sense is not that of replacing the language model itself, but rather to augment its reasoning process with structured knowledge and deterministic computation in order to mitigate well-known failure modes in

[1] Source code available at: `https://github.com/datalab-to/marker`.

mathematical reasoning.
In order to achieve this goal, the system must first be able to ingest, store, and later retrieve the mathematical knowledge needed to later answer the user's queries.
During the design phase, it therefore became evident that the overall framework could be naturally decomposed into three main conceptual blocks as shown in Figure1:

- a mathematical knowledge extraction pipeline;
- a knowledge representation and storage layer;
- a graph-augmented retrieval and reasoning agent.

Each of these components addresses a distinct stage of the end-to-end process, from raw knowledge ingestion to grounded reasoning and deterministic computation. In this section, we present the conceptual design underlying each one of these blocks, deliberately abstracting away from specific implementation choices, which will instead be discussed in later sections of this thesis.

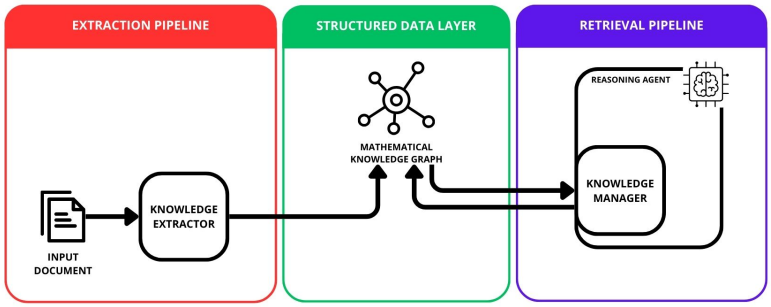

Figure 1: Schema representing the three main blocks of the MathRAG framework. While the data extraction block and the data retrieval block are actual pipelines, the data rapresentation block is a persistent data-representation layer that is used by the other two (built by the first and queried by the last).

3.2. Document Parsing and Knowledge Graph Construction Pipeline

The first major block of the proposed architecture is the mathematical knowledge extraction pipeline, whose purpose is to transform unstructured documents into a structured, semantically enriched, and computationally usable knowledge base.
Building a system such as MathRAG requires acknowledging that mathematical knowledge appears in a wide variety of forms and formats. Formulas may be embedded inline within paragraphs, presented as standalone equations, encoded as images, or described verbally in natural language. Even when limiting attention only to PDF documents, one of the most widely adopted formats for scientific material, regulations, and technical documentation, the heterogeneity of representations remains substantial.
PDFs are inherently unstructured artifacts: reading order, layout hierarchy, and semantic organization are often implicit rather than explicitly encoded. These challenges are further amplified in the presence of scanned or image-based PDFs, where textual information must be reconstructed through optical character recognition.
As a deliberate design choice, the scope of this work focuses on the extraction of mathematical knowledge from digitally-born PDF documents. This restriction allows for a controlled exploration of the extraction problem while maintaining practical relevance. This first extraction pipeline operates primarily in an offline or pre-processing phase. The document is fed into the system so that the knowledge can be extracted prior to the actual user interaction with the system and will be later available for the reasoning agent to use. The full pipeline is shown in Figure 2

3.2.1 Document Preprocessing and Marker-Based PDF Conversion

The very first step of the pipeline consists of ingesting the PDF documents and converting them into a machine-readable and structurally coherent representation. For this purpose, the system employs *Marker*, a state-of-the-

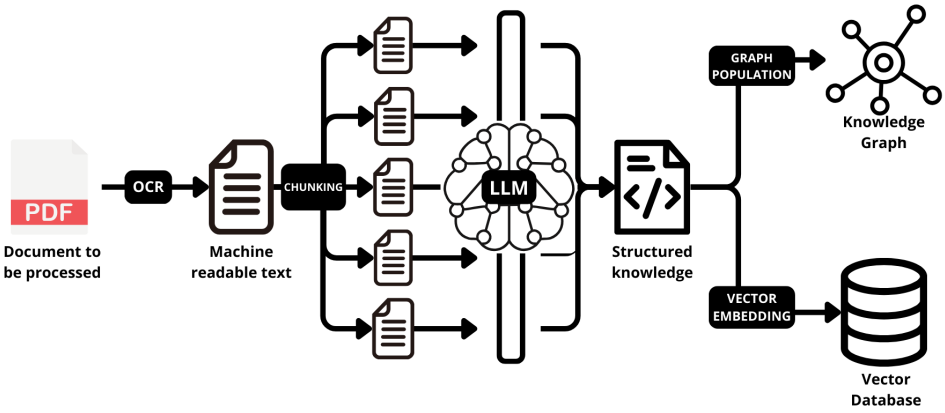

Figure 2: The full pipeline that turns ingested PDF documents into ready-to-query RDF graph and VectorDB.

art PDF-to-Markdown converter developed by Datalab[2].
Marker differs substantially from traditional PDF extraction libraries, which typically operate by extracting raw text fragments without preserving document structure or mathematical notation. In contrast, Marker leverages multiple deep learning models to analyze document layout and reconstruct semantic organization.
At a high level, Marker is organized as a modular pipeline in which a set of specialized deep learning models are responsible for different document-understanding tasks. A layout analysis model first detects the structural components of the page, such as multi-column regions, section headings, tables, and figures. The resulting blocks are then processed by a reading-order reconstruction model that infers the correct logical sequence of the text across complex layouts. In parallel, a dedicated formula recognition model identifies mathematical expressions and converts them into valid LaTeX representations.
When the input document is not digitally born, an OCR model is selectively activated to transcribe text from scanned or image-based pages. The outputs of these models are finally merged into a unified representation, producing a Markdown document in which headings, paragraphs, tables, figures, and, most importantly, mathematical formulas are preserved with their structural and semantic relationships.
This step is fundamental for the reliability of the entire pipeline. Traditional PDF extractors often flatten fractions, distort superscripts and subscripts, or scramble multi-column layouts, producing text representations that are difficult for language models to interpret accurately. By contrast, Marker preserves formulas directly in LaTeX form, thereby providing the language model with clean symbolic input. The resulting markdown file constitutes the processed and structured representation that can proceed into the next phases of the extraction process.

3.2.2 Section-aware Chunking Strategy

Large language models operate under limited context windows, making it impractical to process entire documents in a single pass. Consequently, the structured markdown document is segmented into smaller chunks to prevent content truncation and ensure reliable downstream extraction. This design choice is a direct consequence of the way the Marker OCR model outputs structured text: since Marker produces well-formed Markdown with explicit section headers and paragraph boundaries, the chunking strategy leverages this structure to preserve semantic coherence during segmentation.
In similar settings, a common approach consists of dividing the entire document into fixed-length segments with partial overlap in order to minimize the risk of separating semantically related concepts. In this work, however, a different strategy was adopted, one that leverages the highly predictable structure in which the OCR system outputs documents, where sections and paragraphs are explicitly marked and easily identifiable. For this reason, a section-aware chunking strategy was introduced.
The process operates as follows: (i) the document is first divided according to Markdown section headers (e.g., `##` or `###`), ensuring that semantically coherent sections are treated as unified blocks; (ii) if a section exceeds the

[2] `https://github.com/datalab-to/marker`

maximum allowed size, it is further split along paragraph boundaries, rather than by arbitrary character counts; (iii) paragraphs are incrementally grouped until the size constraint is reached, preserving complete explanatory context for each formula.
This strategy ensures that formulas remain coupled with their corresponding descriptions, variable definitions, and explanatory text, and at the same time that the context-window character limit is always respected. By prioritizing structural boundaries over raw character limits, the system aims at maintaining contextual integrity first and foremost. The resulting chunks therefore contain coherent and semantically self-contained portions of the document, improving the reliability of downstream formula extraction and knowledge graph construction.
The total number of generated chunks is thus not fixed and depends on both the length of the document and the density of its structural annotations. A short, well-sectioned document may yield one chunk per section, whereas a longer or poorly segmented document may produce multiple chunks, each approaching the 3000-character limit.

3.2.3 LLM-Based Mathematical Knowledge Extraction

Once converted into machine-readable form and segmented into chunks, the identification and extraction of mathematical knowledge is delegated to a large language model. This design decision was motivated by the need for scalability and domain generality. Rather than relying on manually crafted rules or domain-specific parsers, the system leverages the language model's capacity to recognize mathematically relevant content across heterogeneous domains.
Each generated chunk is enriched with metadata, including source document references and positional information. This metadata is subsequently stored in the knowledge graph, enabling traceability and provenance tracking of extracted formulas and entities. By preserving the association between extracted knowledge and its original textual context, the system supports transparency, traceability, and future graph maintenance. This structured and section-aware segmentation mechanism therefore allows scalable processing of large corpora while maintaining the semantic relationships necessary for reliable knowledge extraction.
Each chunk is provided to the model together with a structured prompt instructing it to extract formulas, variables, constants, relationships, and descriptive metadata. For each formula, the extracted information includes a textual description of the formula, a short phrase that indicates the scope of the formula (what it is used for), a set of keywords, an informative label, the parameters that the formula takes as inputs with descriptions for each of them, the unit that the numerical result of the formula requires, and finally the LaTeX representation of the formula and a Python-ready version that will be used downstream in the creation and evaluation of the deterministic tool.
When it comes to constants, the extracted information is again a textual description, the scope of application, a list of keywords, the symbol that the constant appears with in the different formulas, and finally, its numerical value and units.
The model also identifies semantic relationships between formulas and constants (e.g., derivation, specialization, and the use of constants).
The model output is constrained to a predefined JSON schema to ensure structural consistency. Validation mechanisms check syntactic correctness and schema compliance, enabling automatic detection of malformed outputs.

3.2.4 Computational Code Generation

To enable deterministic execution during downstream reasoning, extracted formulas must be transformed into executable code. The pipeline adopts a structured fallback strategy for LaTeX-to-Python conversion.
In the first stage, the transformation is attempted using the `SymPy` parsing utilities [12], which provide native support for converting LaTeX expressions into symbolic Python representations. This step ensures high fidelity for standard mathematical notation and preserves semantic correctness. When this automatic parsing fails, a rule-based conversion layer applies deterministic mappings for known patterns and operator structures. As a final fallback, a language model is queried to generate the corresponding Python implementation for expressions that cannot be handled by the previous methods.
All generated code is subjected to syntactic validation before being incorporated into the knowledge base. In addition, a semantic consistency check verifies that the variables declared in the formula metadata are correctly reflected in the generated Python template, ensuring alignment between the symbolic definition and its executable form. If all conversion attempts fail, the formula is retained only in its original LaTeX representation. This design guarantees that symbolic knowledge is never discarded while enabling reliable computational use whenever a valid translation is available. This method also, together with the incremental nature of the rdf graph construction, allows the Python templates that couldn't be created in the processing of a document, to be completed in the future when another document is analyzed, the same formula is detected and if this time a Python template gets correctly compiled it will go to enrich the entity that was already present in the graph.

3.2.5 Deduplication and Knowledge Consolidation

Since formulas may appear multiple times within or across documents, a deduplication phase merges semantically equivalent entities. In order to do so, the system computes dense embedding vectors for each formula and constant. These embeddings capture semantic information about the mathematical concept beyond exact syntactic representation. Cosine similarity between embedding vectors is used as a quantitative measure of conceptual proximity, with a configurable threshold (e.g., 0.9) determining high-confidence matches.
The deduplication process follows a multi-stage strategy. Exact or near-exact matches identified through the aforementioned similarity are merged immediately. Conversely, clearly dissimilar entities are discarded early to avoid unnecessary computation. For cases that fall within an intermediate "gray zone", where heuristic similarity suggests potential equivalence but does not provide sufficient certainty, an additional verification step is performed. In these edge cases, a large language model is explicitly queried with both formulas (including labels, LaTeX expressions, and parameter lists) using a constrained prompt of the form: "Are these two formulas/constants the same? Answer only SAME or DIFFERENT." Only if the model confirms semantic equivalence are the entities merged.
When duplicates are detected, metadata is consolidated, relationships are remapped to a canonical entity, and provenance information from all occurrences is preserved. This consolidation step prevents fragmentation of identical or equivalent formulas across the graph, reduces redundancy, and improves retrieval precision. Moreover, it supports incremental expansion of the knowledge base: as new documents are processed, previously known formulas can be enriched with additional descriptions, alternative notations, and extended relational links rather than being stored as disconnected duplicates.

3.2.6 RDF Knowledge Graph Construction

The structured JSON output of the extraction phase is used to populate an RDF knowledge graph, serialized as a Turtle (.ttl) file. A custom ontology defines entity classes such as formulas, constants, and document chunks, together with object and data properties encoding relationships and metadata.
Each formula is represented as a node with associated symbolic, descriptive, and computational attributes. Relationships between formulas and links to constants are encoded as RDF triples. Deterministic URI generation ensures stable identifiers across reprocessing cycles. The resulting graph supports symbolic querying via SPARQL and provides a formal semantic backbone for the reasoning component.

3.2.7 Vector Indexing and Semantic Retrieval Support

In addition to the symbolic RDF representation, the pipeline constructs dense vector embeddings for formulas, relationships, and constants. Embeddings are generated from composite textual representations combining labels, descriptions, LaTeX expressions, parameters, and keywords. These vectors are stored in a dedicated database to support semantic similarity search.
This dual representation, symbolic graph plus vector index, enables both precise logical queries and flexible semantic retrieval, forming the basis for the GraphRAG mechanisms employed at runtime.

3.2.8 Architectural Role and Development Prioritization

The knowledge extraction pipeline has been implemented entirely using open-source components. This design choice enables full reproducibility of the end-to-end workflow and allows systematic experimentation and evaluation under controlled conditions. However, it was clear from the outset that relying on open-source OCR systems, PDF parsers, and language models would introduce a higher level of inaccuracy, particularly for mathematically dense documents and for the conversion stages that require both visual recognition and semantic interpretation.
In contrast, several commercial solutions currently achieve very high precision in document understanding and in the generation of directly executable mathematical code. The adoption of an open, fully inspectable stack in this work is therefore motivated not by performance considerations alone, but by the need to obtain a transparent pipeline that can be repeatedly executed, rigorously analyzed, and used to produce scientifically meaningful test results.
From an architectural perspective, the extraction pipeline is designed as a modular ingestion layer whose role is to populate the knowledge graph that is subsequently queried by the reasoning agent. In a future deployed version of the framework, this component can be complemented by additional specialized pipelines targeting different document modalities, such as HTML sources, OCR-centric workflows, or domain-specific structured repositories.
All these heterogeneous ingestion processes can converge toward the same unified graph representation, ensuring that improvements in extraction capabilities do not require modifications to the knowledge model or to the

reasoning architecture. This separation of concerns allows the core contributions of the system, namely the knowledge representation and the agent-based reasoning mechanisms, to remain stable while the data acquisition layer evolves independently.

3.3. Knowledge Representation and Storage

With respect to structured knowledge representation, a central design requirement of the system was generality. The knowledge graph was therefore designed to be as domain-agnostic and versatile as possible.
Although some existing approaches account for rich ontologies capable of modeling theorems, lemmas, definitions, proofs, formulas, and even individual problem instances [2, 23], such expressive power was deemed unnecessary for the intended use case.
The primary goal of MathRAG is to support practical formula-application problems rather than formal proof construction. Consequently, modeling theorems, proofs, or individual problem statements would have added unnecessary complexity without providing a proportional benefit. Instead, the knowledge graph was designed around a single core entity type: the *Formula*, as show in Figure3.
This design choice enables formulas originating from diverse application domains to coexist within a unified schema. Whether describing the energy cost of a domestic appliance, the volume of a geometric solid, or the relationship between voltage, current, and resistance in an electrical circuit, these use cases can all be reduced to explicit input-output relationships expressed as mathematical formulas with associated variables and constants.
Each Formula entity includes several key attributes. First, it contains a natural-language description that clearly explains the scope and meaning of the formula. This description serves a dual purpose: it is used both for semantic similarity calculations during retrieval and as part of the tool creation process. In tool-calling mechanisms, the clarity and expressiveness of a tool's description are crucial, as the language model relies on this textual information to decide whether a particular tool is appropriate for a given task.
Second, the Formula entity stores a formula template expressed in a ready-to-use arithmetic form. This representation is designed to be directly executable, allowing the agent to evaluate the formula without additional rearrangement or symbolic manipulation.
Since formulas typically involve multiple variables and constants, a correct interpretation of variable names is essential in order to bind user-provided values to the appropriate parameters.
For example, in a query such as *"Calculate the current for 5V passing through a 4 Ohm resistance"*, the agent may retrieve the formula $i = V/R$. However, successful execution requires explicit knowledge of what the symbols V, i, and R represent. For this reason, each Formula entity includes a structured list of parameters, accompanied by natural language descriptions, such as: V: voltage of the circuit; i: current of the circuit; R: resistance of the circuit.
In addition, each formula is associated with a set of keywords generated during the extraction phase. These keywords act as an auxiliary semantic index, improving retrieval accuracy during graph search.
Finally, the schema includes an explicit field specifying the physical dimension or unit of the returned value.
During preliminary experiments, the need for an additional type of entity became evident. In many mathematical domains, formulas depend on fixed constants in addition to user-provided variables. A common example is the constant π in geometric formulas. Such constants do not vary across queries and should not be treated as free parameters. To address this, constants are modeled as separate *Definition* entities, each containing a description and a numerical value. These entities are linked to the formulas that use them through explicit relations, enabling the agent to correctly incorporate fixed constants during computation.

3.4. Graph-Augmented Retrieval and Reasoning Agent

The third and final conceptual block of the proposed framework is the Graph-Augmented Retrieval and Reasoning Agent, which represents the core runtime component of the architecture. While the knowledge extraction pipeline and the knowledge representation layer operate primarily in an offline or pre-processing phase, before the user even stated their first question, this pipeline is executed entirely at runtime and is activated only when a new question is submitted into the system.
The Agent intercepts each incoming request coming from the web interface (implemented using Streamlit) and processes the raw user message. Although the complete Web UI has the ability to provide auxiliary contextual metadata about the user which is stating the question, the reasoning logic implemented in this thesis is intentionally designed to be general-purpose and therefore relies exclusively on the textual information contained in the raw user query.
The Agent implements an iterative CoT-reasoning strategy inspired by the ReAct paradigm, combining structured retrieval, dynamic tool synthesis, and explicit workflow control to ensure mathematically grounded and computationally reliable answers.

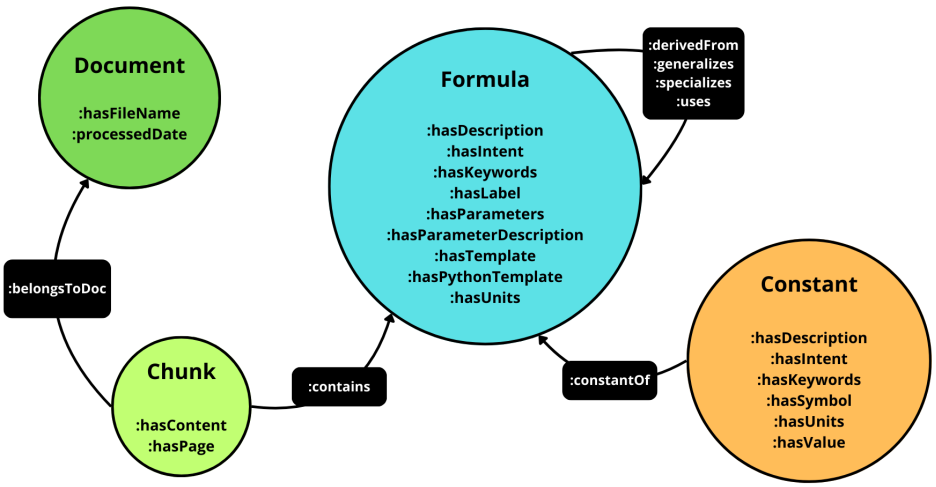

Figure 3: The graph ontology with the four major entity classes and the relationships that link them together.

3.4.1 Episodic State

Each new user query entering the Agent component triggers the creation of an explicit episodic state object. This state encapsulates all information relevant to the current reasoning session and persists throughout the entire Think–Act–Observe loop. The state includes the original user query, the extracted intents, numeric literals and other structured elements parsed from the input, the entities and formulas retrieved from the mathematical knowledge graph, and the set of dynamically generated computational tools created during the interaction. In addition, the state maintains a collection of workflow enforcement flags that record which stages of the reasoning process have already been completed.

The introduction of an explicit state representation serves multiple purposes. First, it ensures consistency across multiple LLM invocations in the same reasoning loop by preventing the loss of intermediate information between reasoning steps. Second, it enables principled enforcement of causal dependencies between actions, such as preventing tool invocation before tool creation. Third, it improves transparency and debuggability by making intermediate reasoning artifacts externally observable rather than implicitly embedded in prompt history.

The episodic state is isolated to the current query session and is discarded upon termination of the reasoning loop, ensuring that subsequent queries start from a clean configuration.

Upon initialization of the episodic state, the Agent enters an iterative reasoning process structured according to a Think–Act–Observe cycle. In each iteration, the Agent first produces a reasoning trace that articulates its current understanding of the problem and determines the next action to perform. It then executes the selected action among those at its disposal, and processes the resulting observation, which is fed back into the subsequent reasoning step. This cyclical structure allows the Agent to decompose complex queries into smaller sub-problems and to dynamically adjust its strategy based on intermediate feedback.

To ensure stability and reduce unpredictability, the Agent operates over a well-defined and explicitly enumerated action space. Constraining the model to a finite set of semantically described actions limits the possible evolution of the reasoning process and promotes consistent behavior across executions. At the same time, each action is explicitly documented and described to the language model, providing clear guidance regarding its intended effect and usage.

The available actions fall into three main categories. *Retrieval actions* query the knowledge graph for either Formula entities or Constant entities, depending on the kind of question the user has posed to the system. In this sense, Formula-retrieval actions prompt an automatic graph-traversal query that retrieves, together with the desired formula, all constants linked to it since they will always be needed for the numeric evaluation later on. Here we will show the main retrieval action that initiates the graph queries:

```
RETRIEVE_MATH_KNOWLEDGE
- Required: You MUST provide either a "query" string OR a "search_terms" list.
    * Use "query" when you have a natural language phrase
      (e.g., {"query": "final angular velocity formula"})
    * Use "search_terms" when you have a list of keywords
      (e.g., {"search_terms": ["angular", "velocity", "kinematics"]})
- Optional: You can include a "literals" dictionary
  if the user provided numbers or categories.
    * Format: {"literals": {"numeric": [5, 2, 6], "categorical": ["zone A"]}}
```

In addition to formula retrieval, the Agent can directly retrieve constants when the user query refers explicitly to a known constant rather than a full formula. This action queries the Constant entities in the knowledge graph.

```
RETRIEVE_CONSTANT - Input: {"query": "constant name or description"}
```

Tool lifecycle actions are responsible for constructing executable computational tools from retrieved mathematical knowledge and invoking them with user-provided or inferred parameters. *Dialogue management* actions handle situations in which the query requires language translation, or direct response generation when no retrieval or computation is necessary.
Here are the two *Tool lifecycle* actions, the one that builds tools and the one that selects them to be called:

```
   CREATE_DYNAMIC_TOOL - Input: {"formula_id": "exact_formula_label_or_uri"}
CALL_TOOL - Input: {"tool_name": "tool_name", "parameters": {"param1": value1}}
```

Finally, the Agent includes dialogue management actions for handling non-computational queries, clarification requests, or direct natural language responses that do not require retrieval or tool execution.

```
RESPOND - Input: {"answer": "final natural language answer"}
```

Intent classification plays a central role in determining which action category should be activated. The Agent analyzes the user query for numeric literals, symbolic expressions, and computational language patterns such as "calculate", "compute", or "how much". Queries that exhibit explicit mathematical intent trigger the retrieval of Formula or Constant entities from the knowledge graph and initiate the structured mathematical reasoning cycle. In contrast, purely conceptual or explanatory questions bypass retrieval and activate direct response generation, while ambiguous queries may trigger a clarification action before proceeding.
Conversely, queries that do not present clear mathematical characteristics bypass the retrieval mechanism altogether and are answered directly by the underlying language model without augmentation. This design, visualized in Figure 4 avoids unnecessary graph access, reduces latency, and preserves the standard generative behavior of the model for purely descriptive or conversational requests.

3.4.2 Dynamic Tool Factory

A central capability of the Agent is the dynamic synthesis of computational tools from declarative knowledge stored in the knowledge graph. When mathematical intent is detected, the Agent retrieves, using the URIs associated to the most semantically similar embedding vectors found in the database, the Formula entities from the RDF graph together with all their fields and attributes: descriptions, parameters and parameter descriptions, and most importantly the template that will be used to carry out the computation.
These retrieved specifications are dynamically transformed into executable Python functions. The transformation process instantiates a numerical Python function using the retrieved arithmetic expression, injects relevant constants obtained from visiting the knowledge graph into the function scope when required, constructs parameter validation logic according to RDF-encoded type specifications, and wraps the resulting function within a standardized interface compatible with structured tool calling. This interface enables the language model to invoke the tool through structured parameter passing rather than free-form text generation. For instance, the numerical result is computed through a restricted evaluation call of the form:

```
result = eval(template, {"__builtins__": {}}, namespace)
```

In this call, `template` denotes the arithmetic expression retrieved from the knowledge graph (already in valid Python syntax); `__builtins__` is explicitly set to an empty dictionary to prevent access to Python's default built-in functions and thereby restrict execution for security reasons; and `namespace` represents the controlled

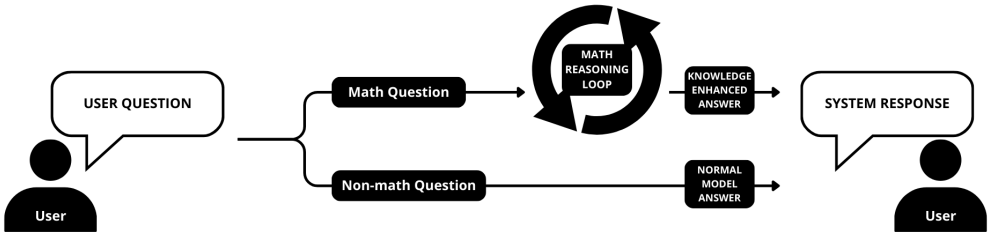

Figure 4: The two branches that can be followed by the Reasoning Agent after the first iteration, where it decides whether or not the user question needs specific mathematical knowledge to be answered or the LM can handle it on its own.

execution environment containing user-provided parameters, injected physical constants, and explicitly allowed mathematical functions (e.g., `math`, `numpy`).
By converting static, declarative graph knowledge into executable procedures, the Agent ensures deterministic computation of formulas without requiring manual implementation of individual calculations, thus solving one of the main failure modes of previous similar works. Importantly, only a small subset of semantically relevant tools is generated for each query. Since all available tools must be included in the LLM prompt during tool calling, restricting tool creation to highly relevant candidates reduces prompt size and improves efficiency and selection accuracy during subsequent reasoning steps.

3.4.3 Workflow Enforcement

To prevent incoherent execution patterns, the Agent incorporates an explicit gating mechanism that validates action preconditions against the current episodic state. For example, tool creation is permitted only after the relevant formula has been successfully retrieved from the knowledge graph, and tool invocation is allowed only if the corresponding tool has already been created and registered in the state.
If the language model selects an action whose preconditions are not satisfied, the Agent does not execute the action. Instead, it returns a structured observation explaining the violated constraint. This observation is incorporated into the next iteration of the reasoning loop and typically leads the model to self-correct. This mechanism prevents degenerate sequences such as invoking non-existent tools, generating tools without prior retrieval, or skipping necessary reasoning stages. By externalizing workflow constraints into explicit state flags and validation checks, the Agent maintains logical coherence throughout the execution process.
For example, if the Agent selects action `CALL_TOOL` before any tool has been created, the workflow validator intercepts the action and returns a structured error such as `"ERROR: You must create a tool first using CREATE_DYNAMIC_TOOL."`. Similarly, if `CALL_TOOL` is invoked with a `tool_name` that does not exist in `state.available_tools` (e.g., because the list is empty), the execution layer returns a message of the form `"Error: Tool 'X' not found. Available tools: [...]"`. In both cases, the error is appended as an `Observation` in the reasoning history, allowing the language model to inspect the explicit constraint violation and adjust its subsequent action accordingly.

3.4.4 Termination of the Loop

The CoT-reasoning loop terminates under two conditions. The first occurs when the Agent selects the `ANSWER_DIRECTLY` action and produces a final response. This action is chosen both in trivial cases, such as greetings (e.g., "Hi"), where no retrieval or computation is required, and in cases where all necessary retrieval, tool creation, and computation steps have been successfully completed, and the workflow flags indicate that the reasoning process is logically consistent.

The second termination condition occurs when a predefined maximum number of iterations is reached. This safeguard prevents infinite reasoning loops in the presence of unexpected behaviors or unresolved ambiguities. Upon termination, the episodic state object is discarded, and all dynamically generated tools are disposed of. This volatile tool lifecycle guarantees that computational capabilities remain contextually relevant to the current query and prevents interference between unrelated query sessions.

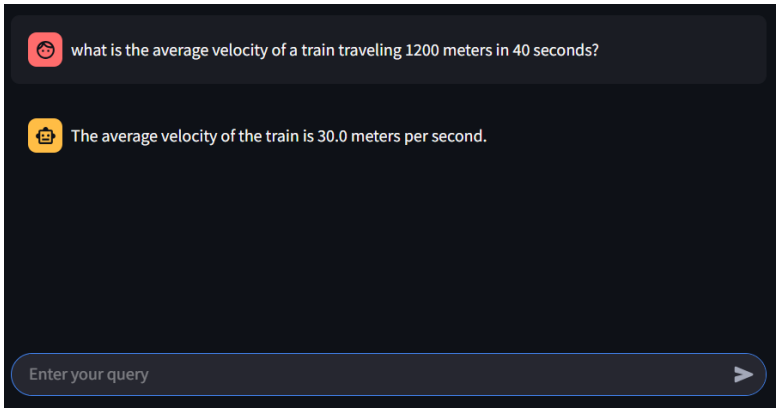

Figure 5: An example of a mathematical question posed trough the UI provided by Streamlit.

3.5. End-to-End Workflow and System Interaction

To illustrate the interaction between the ingestion pipeline and the runtime reasoning component, this section presents a complete end-to-end execution scenario, from document acquisition to final answer generation.
Consider a generic PDF document containing multiple mathematical formulas, descriptive explanations, as well as references to physical constants. The document is first processed by the Marker-based conversion pipeline, which transforms it into a structurally coherent markdown representation while preserving headings, paragraphs, and mathematical expressions in valid LaTeX form. The section-aware chunking mechanism then segments the structured markdown into semantically consistent blocks aligned with document headers and paragraph boundaries. Each chunk is subsequently submitted to the language model for structured knowledge extraction. For every detected formula, the model produces a JSON object containing its description, parameters with semantic annotations, keywords, output unit, LaTeX representation, and a Python-ready template. Constants appearing in the formulas are extracted as separate entities with numerical values and units.
After syntactic and schema validation, the extracted entities are converted into RDF triples and incorporated into the knowledge graph. Deduplication mechanisms merge semantically equivalent formulas using embedding similarity and, when necessary, language-model-based equivalence checks. In parallel, dense vector embeddings are generated from composite textual representations and stored in a vector index to enable semantic retrieval. At this stage, the document has been fully ingested, and its mathematical knowledge is accessible through both symbolic and vector-based querying.
Assume now that a user submits the following query: *"Calculate the electrical current when a voltage of 12V is applied across a 4 Ohm resistance."* Upon receiving the query, the Agent initializes a new episodic state object and performs intent classification. The presence of numeric literals (12, 4), symbolic quantities (V, Ohm), and computational language ("calculate") triggers the mathematical reasoning branch.
In the first reasoning iteration, the Agent determines that relevant mathematical knowledge must be retrieved and selects the action `RETRIEVE_MATH_KNOWLEDGE`, providing an appropriate search query such as "electric current voltage resistance formula". The retrieval layer performs a hybrid search over the vector index and the RDF graph, returning the formula corresponding to Ohm's law together with its associated parameters and linked constants.
During the next iteration, the Agent attempts to proceed directly to computation and selects `CALL_TOOL`. However, since no computational tool has yet been instantiated, the workflow enforcement mechanism detects a violated precondition. Instead of executing the action, the system returns a structured observation such as `"ERROR: You must create a tool first using CREATE_DYNAMIC_TOOL."`. This observation is appended to the reasoning history and incorporated into the subsequent reasoning step. The language model, now informed of the violated constraint, self-corrects and selects the appropriate action `CREATE_DYNAMIC_TOOL`, specifying the URI of the retrieved formula.
The Dynamic Tool Factory then transforms the Python-ready formula template (e.g., `i = V / R`) into an executable Python function. The resulting tool is registered in the episodic state under a structured schema describing its parameters. In the subsequent iteration, the Agent selects `CALL_TOOL`, providing the parameter

bindings `{"V": 12, "R": 4}`. The tool executes deterministically via the restricted evaluation mechanism and returns the numerical result (3.0). The Agent then selects the dialogue action `RESPOND`, producing a natural-language explanation stating that, according to Ohm's law, the current equals the voltage divided by the resistance, which in this case results in 3 amperes.
Once the final response is generated, the reasoning loop terminates. The episodic state object is discarded, and the dynamically generated tool is deallocated, ensuring isolation between independent query sessions. This example highlights the coordinated interaction between the offline ingestion layer and the online reasoning Agent: unstructured PDF content is transformed into structured, executable knowledge, which is then synthesized into a computational tool at runtime within a controlled Think–Act–Observe loop.

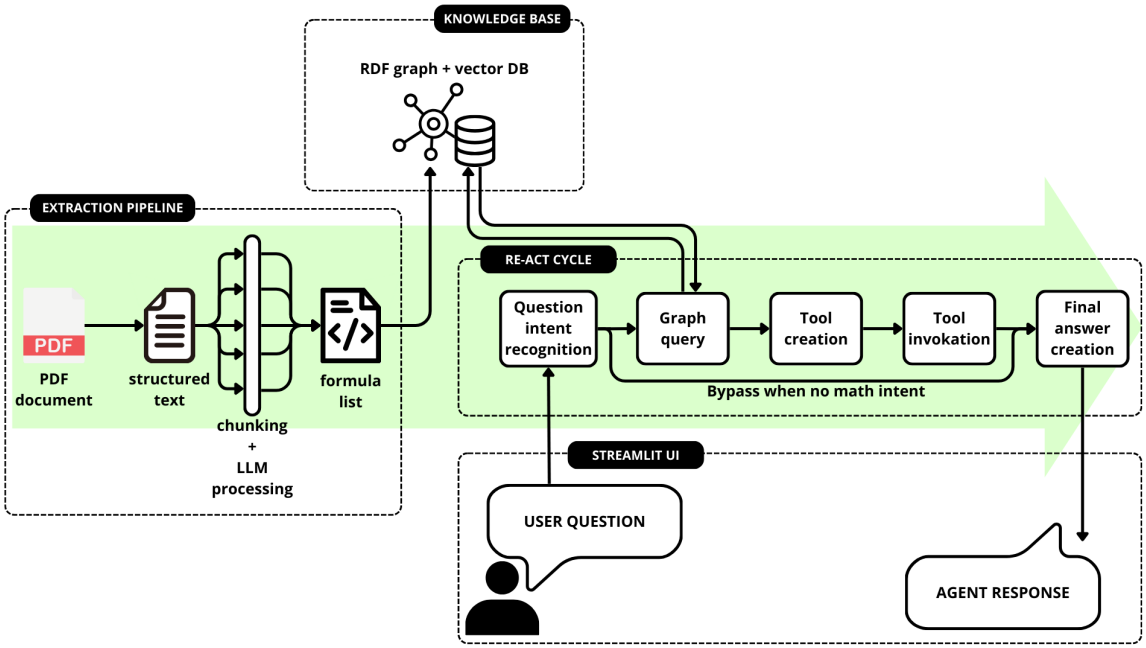

Figure 6: The end-to-end pipeline that goes from the upload of the file and its subsequent processing, to the user's question to the Agent through the Streamlit UI that sparks the ReAct cycle and the creation of the final answer based on the information saved in the knowledge base.

4. Implementation Experience

This section presents the technical architecture and implementation details of the proposed mathematical knowledge extraction and question-answering system. We will go over the specific interactions between the classes and components of the two main computational subsystems: (i) the compile-time document-driven knowledge extraction and graph construction pipeline, and (ii) the run-time interactive reasoning and computational-tool-executing agent. This design separates offline knowledge construction from online query processing while ensuring consistency between symbolic and semantic representations.

4.1. Document-Driven Knowledge Extraction

4.1.1 System Entry Point and Initialization

The knowledge extraction process begins with a document ingestion stage, which represents the entry point of the system. The system accepts scientific content either in PDF format or as an already structured textual representation. At this stage, the necessary components for handling the input source are initialized, ensuring that the subsequent processing pipeline can operate in a consistent and domain-independent manner.

- A large language model (LLM) provider (Ollama's LLama3.2:7B),
- An embedding model for semantic indexing (mxbai-embed-large),
- A persistent storage directory for graph and vector artifacts.

The configuration described above corresponds to the specific setup adopted for the experimental evaluation. However, the system architecture is model-agnostic and supports alternative language models and embedding models through modular parameterization. Different backbone models can be integrated by modifying high-

level configuration settings, without requiring structural changes to the pipeline. This design ensures flexibility, reproducibility of experiments, and extensibility to future model variants.

4.1.2 Document Preprocessing and Canonical Representation

For PDF inputs, the system employs the Marker model to convert PDF content into structured Markdown while preserving both LaTeX expressions and overall document layout. Preserving the intact LaTeX syntax is critical, as subsequent symbolic conversion depends on syntactically valid mathematical expressions.
The choice of this OCR framework was motivated by its ability in handling mathematically dense scientific documents, ensuring reliable reconstruction of formulas and structured elements such as tables, forms and code blocks.
The output of this stage is a single markdown document that serves as the input for formula extraction. Documents already provided in Markdown or plain text bypass this conversion stage and are fed directly into the next stage.

4.1.3 LLM-Based Formula and Constant Extraction

As described above, the markdown text is segmented into chunks using a paragraph-aware mechanism that preserves semantic coherence and ensures that logically connected information is not split across segments. This strategy maintains contextual integrity while remaining compatible with LLM context window constraints. Each chunk is then processed independently by the LLM using a structured prompt that enforces JSON-formatted output.
For each identified formula, the system extracts a human-readable label, the LaTeX template, Python-ready template (if present and valid) which is the actual formula that will be used in the computation, parameter lists and descriptions, units, a semantic intent and textual description, keywords, and inter-formula relationships.
The units, Python template, and parameter definitions with their descriptions are primarily used for downstream implementation, as they enable executable tool generation and structured parameter validation during runtime. The textual description, semantic intent, and LaTeX representation are provided to the LLM-based reasoner to supply contextual information about the mathematical meaning and applicability of the formula. Together with the associated keywords, these elements also contribute to the construction of semantically rich embeddings, improving the expressiveness and effectiveness of similarity-based retrieval within the vector database.
Mathematical constants are systematically extracted with their symbol, numerical value, associated units, descriptive keywords, and explicit references to the formulas in which they appear. Additionally, an *intent* field provides a natural language explanation of the context, purpose, and typical usage of each constant within relevant formulas. The extraction methodology is detailed in the Formula Extraction Prompt presented in Appendix A.1.
The symbol, numerical value, units, and the linkage to related formulas are, again, essential for the downstream tool construction, as they enable automatic parameter substitution and consistent numerical computation during execution. In contrast, the descriptive text, semantic intent, and associated keywords serve a dual purpose: they provide explanatory context to the LLM-based reasoner during query processing, and they contribute to the construction of semantically expressive embeddings that improve similarity-based retrieval within the vector database.
Finally, after the Entity extraction has finished, the resulting structured JSON output is validated and sanitized before being converted into internal data structures to ensure consistency, completeness, and computational reliability.

4.1.4 LaTeX-to-Python Conversion

To enable executable computation, LaTeX expressions must be converted into Python templates. In our approach, a multi-stage fallback strategy ensures that every generated template is valid Python code. The Python template stored for each formula node in the knowledge graph is a strict operational requirement: it is the numerical expression that the Agent will use at response-time to build the numerical Python function that will be used to calculate the numerical answer to the user's question. Templates must therefore be syntactically valid, self-contained Python expressions, with symbols matching parameter names that the model can bind to values extracted from the user query. Invalid templates render the associated formula node unusable, regardless of retrieval accuracy.
The extraction pipeline applies three successive conversion strategies. The first is (i) LLM-generated code: during the structured extraction pass (with prompt A.1), the language model is explicitly prompted to output a Python expression alongside formula metadata. This leverages the model's semantic understanding of dependent and free variables. The output is immediately validated using Python's Abstract Syntax Tree (AST) parser, if validation fails, the pipeline proceeds to the next strategy.

The second strategy (ii) uses a grammar-based parser via the `latex2sympy2` package, translating LaTeX expressions into SymPy symbolic representations, then serialising them into Python code. Prior to parsing, the left-hand side of equations is stripped using regular expressions to isolate the right-hand-side expression. This approach ensures mathematically consistent conversions for classical and commonly used formulas, though it may struggle with domain-specific notation, such as in physical chemistry or thermodynamics.
As a final fallback, (iii) a rule-based converter applies deterministic regular expression substitutions to the raw LaTeX string. Fractions, powers, roots, Greek letters, and transcendental functions are rewritten into syntactically correct Python forms, and residual formatting commands are removed. The resulting expression undergoes AST validation to ensure executability.
All templates are further validated against the formula's parameter list to verify that each symbol appears explicitly. This step mitigates issues with implicit multiplication, which can otherwise merge adjacent symbols (e.g., converting $\frac{1}{2}kx^2$ into `(1/2)*kx**2` rather than the correct `(1/2)*k*x**2`). Formulas that fail all conversion stages are retained with their symbolic and semantic metadata but without an executable template.

4.1.5 Entity Deduplication

Duplicate detection is performed prior to graph construction. Formula deduplication relies on normalized LaTeX representations, while constant deduplication considers symbol, numerical value, and units to ensure consistency. Metadata fields are merged conservatively to preserve all relevant information, avoiding overwriting or loss during consolidation.

4.1.6 RDF Knowledge Graph Construction

The consolidated entities are materialized into a Resource Description Framework (RDF) graph using the `rdflib` library. The domain-specific ontology adopted for the graph structure has already been illustrated in Figure3 and defines the classes and properties that comprise the representation.
Following standard RDF construction principles, knowledge is encoded as triples of the form *subject–predicate–object*, where each formula, constant, document, and chunk is represented as a subject or object node, and their attributes and relationships are represented through predicates. Both datatype properties (e.g., LaTeX template, Python template, numerical value, units, textual descriptions) and object properties (e.g., derivation links or usage dependencies) are instantiated as triples and stored within the graph, collectively forming the structured knowledge base.
Within the current system design, `math:Formula` and `math:Constant` entities constitute the core of the knowledge-based answering process, as they are directly involved in retrieval, reasoning, and dynamic tool generation. In contrast, `doc:Document` and `doc:Chunk` entities primarily serve a provenance and contextualization role, linking each extracted mathematical entity to its original source and location within the document.

4.1.7 Vector Embedding and Semantic Indexing

To support semantic retrieval, textual representations of formulas and constants are embedded using dense vector representations generated by the *mxbai-embed-large model*. For each formula entity, a composite textual description is constructed by concatenating the formula label, natural language description, parameter names and their semantic meanings, intended use case, output units, and semantic keywords, omitting numerical values and LaTeX templates to avoid embedding noise. These concatenated descriptions produce embeddings of dimensionality 384.
Constant entities follow a similar encoding strategy, incorporating their symbolic notation, descriptive text, physical units, and domain-specific keywords, while excluding raw numerical values. This approach maximizes semantic coverage by capturing conceptual meaning and relational context, enabling the embedding model to map relationships between mathematical concepts and natural language queries such as "calculate energy from mass and velocity."
The resulting vector embeddings are indexed in a *ChromaDB* collection using cosine similarity metrics, creating a semantic search layer that complements the structural precision of SPARQL queries over the RDF graph. This dual-retrieval architecture enables both exact graph-traversal queries and fuzzy semantic matching.

4.2. Reasoning Agent and Tool Execution

The second major subsystem is the one that provides interactive question answering capabilities over the extracted knowledge base.

4.2.1 System Initialization

Upon startup, the knowledge manager module:

1. Loads the RDF graph into memory,
2. Connects to the ChromaDB vector store,
3. Initializes the LLM provider,
4. Loads the embedding model.

In-memory graph loading ensures low-latency SPARQL querying and graph visiting.

4.2.2 Query Processing Framework

User queries are processed using a ReAct (Reasoning and Acting) framework [21]. The full system prompt goverining this reasoning process is available in Appendix A.2. The system maintains an internal state structure tracking retrieved knowledge, generated tools, execution status, and the final answer.
At the beginning of the computation (when the query comes from the UI interface) a `ReactState` object gets created. This object works as a container that holds all the informations that will be used by the Agent during a single response. First of all the text of the user question gets analyzed and it's intent gets extracted together with any keyword that will be used to direct the graph search later on.
For example a question like

Hello, My name is Marco.

Will be instantly recognized as a normal greeting and no mathematical intent will be extracted from it, thus signaling the Agent to not engage in the cyclical Think-Act-Observe pattern and instead go directly to the final answer generation.
Instead a question like

What is the linear velocity of an object turning at omega = 5 rad/s on the border of a circle of radius r = 2 cm?

Will be instantly recognized as a mathematical question. The LLM will thus first of all propagate forward in the thinking process the information that the thinking mechanism will be needed and on top of that a series of simple variables, values and literals will be extracted as well to be later on used to retrieve the correct entities in the graph and plug the correct values in the correct places.
Another information contained in the ReActState are the values for the different flags that act as workflow-enforcement gates. These boolean values are needed in order to enforce the correct thinking flow and to prevent incomplete reasoning or direct answer generation without computation.

4.2.3 Semantic Retrieval

When retrieval is triggered, the system embeds the user query using the same embedding model employed during indexing, ensuring vector space consistency. A cosine similarity search is performed across the ChromaDB collection, initially retrieving the top 10 candidate formula entities.
These 10 candidates are then deduplicated by URI and filtered to retain the 5 closest entities by cosine distance. Each of these 5 formulas is hydrated from the RDF graph via SPARQL queries, fetching complete metadata including descriptions, parameters, units, Python and LaTeX templates, keywords, parameter descriptions, and intended usage.
For each retrieved formula, a secondary SPARQL query enumerates all associated constants, extracting their symbols, numerical values, and units. The system then performs symbolic substitution: if a formula's Python template contains variables matching constant symbols (e.g., c for speed of light), their numeric values are directly injected into the expression, and these constants are removed from the required parameter list. This automatic enrichment transforms a formula like

E = m * c**2

with parameters [m, c]. Into

E = m * 299792458**2

requiring only [m], making tool invocation actively more effective by eliminating the need for users to provide fundamental physical constants they may not readily know.
Although 5 formulas are fully retrieved and stored in the agent's state, only the top 3 are presented in the observation text used by the LLM for tool selection. The remaining formulas are acknowledged in summary (e.g., ... (+2 more available)) but are not explicitly displayed, avoiding unnecessary inflation of the context size.

4.2.4 Dynamic Tool Generation

Retrieved formulas are converted into executable Python functions at runtime through a dynamic tool generation pipeline implemented via the `DynamicToolFactory` class. This class accepts formula metadata retrieved from the RDF graph and programmatically constructs a complete computational tool consisting of three integrated components: (i) a Python function synthesized directly from the stored PythonTemplate property (that has been validated trough the AST parsing to ensure syntactic correctness), (ii) a structured parameter schema documenting each input variable's name, type, semantic description, and requirement status, and (iii) a `ToolWrapper` object that encapsulates the function with standardized invocation and validation interfaces.
The `ToolWrapper` provides a uniform invoke(parameters) method that validates parameter types and completeness against the schema, executes the underlying function, and returns type-checked numerical results, abstracting away from formula-specific implementation details.
The workflow enforcing the agent's structured operation is present here as well, since tools are instantiated exclusively after successful knowledge retrieval. This ensures that all computational artifacts originate from the verified knowledge graph rather than from statically defined, hardcoded functions that would behave identically across all queries. In other words, every tool the agent uses is dynamically constructed from the graph content, reflecting the current knowledge base, rather than relying on fixed implementations that are pre-defined and invariant. This constraint guarantees provenance traceability.

4.2.5 Tool Invocation and Validation

As explained before, the tool execution is mediated through a dedicated method that performs multi-stage validation before computation. The invocation pipeline first verifies parameter completeness by cross-referencing provided inputs against the tool's schema, identifying missing required parameters and rejecting calls with incomplete argument sets.
Type validation follows, ensuring all parameters conform to expected numeric types (integer or floating-point) and converting string representations where necessary. Upon successful validation, the wrapper executes the encapsulated Python function in a controlled environment, captures the numerical result, and propagates it back to the reasoning loop as an observation. This observation, formatted as `"Tool result: <value>"`, becomes part of the conversation history and informs subsequent reasoning steps about the results that have been obtained.
Error handling is comprehensive: missing parameters generate descriptive error messages specifying which inputs are required, type mismatches report expected versus received types, and execution exceptions (division by zero, domain errors) are caught and returned as diagnostic observations rather than terminating the workflow, enabling the agent to request corrected parameters or alternative formulas.

4.2.6 Workflow Enforcement and Error Handling

The core of the gating mechanism that enforces the agent's workflow relies on three boolean flags: `has_retrieved_knowledge`, `has_created_tool`, and `has_called_tool`. These flags track mandatory state transitions, ensuring that knowledge must be retrieved before tools are instantiated, and tools must be created before they are executed. By enforcing this order, the system guarantees that all computational steps derive from verified graph content and prevents the agent from performing operations out of sequence.
Knowledge retrieval actions (`RETRIEVE_MATH_KNOWLEDGE`) are permitted only when `has_retrieved_knowledge` is false; subsequent retrieval attempts are blocked with validation errors that explicitly guide the agent toward tool creation, preventing infinite search cycles and at the same time preventing the Agent to try to answer directly if the question has been flagged as a mathematical one. Once the knowledge has been successfully retrieved the next step is the tool creation. `CREATE_DYNAMIC_TOOL` requires prior successful retrieval (`has_retrieved_knowledge == true`), ensuring all tools derive from verified graph content.
Tool invocation (`CALL_TOOL`) mandates prior tool creation (`has_created_tool == true`), and answer generation for mathematical queries (`ANSWER_DIRECTLY`) requires successful tool execution (`has_called_tool == true`), guaranteeing that numerical responses originate from actual computation rather than language model hallucination.
This mechanism is further enriched by the use of informative error messages generated any time the agent tries to force the normal workflow and that specify the required next action, guiding the agent toward compliant behavior. The reasoning loop terminates either upon successful answer generation (a full computation that has gone through all the steps in order) or when a maximum iteration threshold (default: 10) is exceeded.
In the latter case, a fallback mechanism invokes `_build_final_answer_prompt()`, which synthesizes a response from the accumulated conversation history, ensuring graceful degradation rather than silent failure when the agent encounters confusing queries or graph retrieval anomalies.

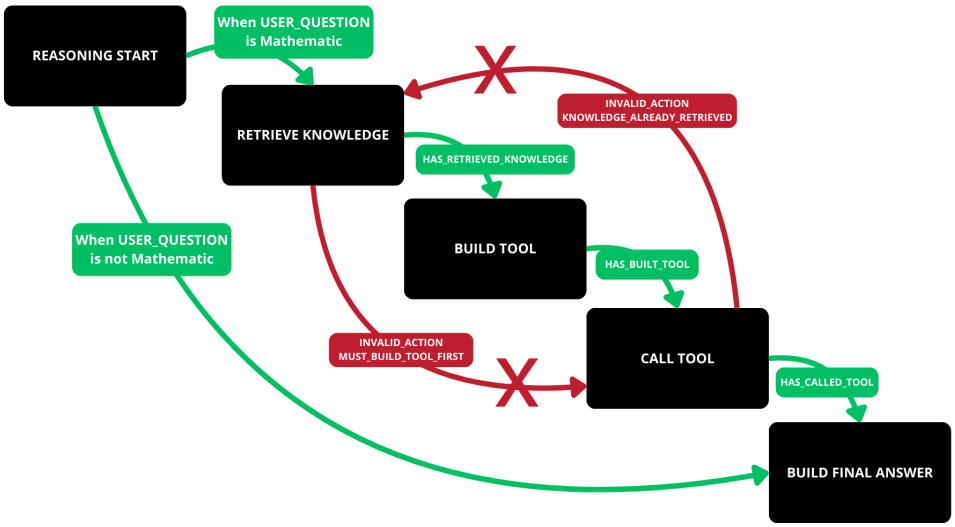

Figure 7: The different stages of the Workflow that the Agent has to go through in order to produce a valid final answer, with some examples of invalid actions and the error message that the system generates.

4.3. Code Availability

The source code implementing the methods described in this thesis is publicly available on Github in the MathRAG repository [3].

5. Experiments and Discussion

This section presents the experimental evaluation of the proposed MathRAG framework under a full end-to-end setting. The testing setup is designed to test the from document ingestion and formula extraction to knowledge graph construction, retrieval, tool synthesis, and final response generation.
The objective of this evaluation is to assess the robustness of the entire system when operating on realistically ingested documents, rather than on manually curated and noise-free graph entities.

5.1. Experimental Setup

5.1.1 Document Ingestion and Knowledge Graph Construction

To evaluate the full pipeline, two publicly available documents containing mathematical formulas were selected from the internet:

- A physics cheat sheet containing physical formulas and theorems.
- A chemistry formula sheet containing physical chemistry formulas and laws

These documents were provided as input to the document parsing and knowledge extraction pipeline described in previous sections. The parser processed the documents to identify mathematical expressions, extract parameters and constants, and construct corresponding RDF entities and relationships.
After processing the first document, the resulting RDF graph contained:

- 110 formula entities,
- 19 constant entities,
- 97 relationships.

The ingestion of the second document extended the graph with:

- 66 formula entities,
- 20 constant entities,
- 90 additional relationships.

The final knowledge graph therefore consisted of:

[3] `https://github.com/relli-d/MathRAG`

- 176 total formula entities,
- 39 total constant entities,
- 187 total relationships.

This graph was used without manual correction, ensuring that the evaluation reflects the real behavior of the complete ingestion and extraction pipeline.

5.1.2 Question Dataset Construction

Since no predefined benchmark dataset existed for the specific content of the selected documents, a dedicated evaluation corpus was manually constructed.
First, a set of formulas was manually selected from the two ingested documents: 30 formulas selected from the first document and 30 formulas selected from the second one.
For each selected formula, a corresponding natural language question was manually formulated. Each question required the agent to retrieve the appropriate formula and compute a numerical result. For example, from a formula such as:

$$\text{Average Velocity: } v_{\text{avg}} = \frac{\Delta x}{\Delta t}.$$

The following question was created

What is the average velocity of an object that travels 200 meters in 5 seconds?

To increase the size and variability of the test corpus, each manually written question was further expanded using a large language model. Specifically, a few-shot prompting strategy was employed to generate 10 paraphrased variants for each original question. For example, the base question:

> What is the volume of a sphere with radius 4 cm?

could be paraphrased as:

> Volume of a sphere that has a radius of $r = 10$ cm.
> Use the formula for the volume of a sphere to compute V with $r = 1$ cm.
> Volume of sphere with $r = 8$ cm.

This process served to produce an expanded and linguistically diverse question corpus that amounted to a total of 600 total differen questions.

5.1.3 Ground Truth Construction

For each generated question, the corresponding numerical result was computed manually to establish ground truth values. The final evaluation dataset was stored in a JSON file, where each entry contained the following fields: (i) `id`, a unique numerical identifier for the question; (ii) `question`, the natural language query posed to the agent; (iii) `formula`, a textual representation of the target formula to be retrieved and applied; (iv) `answer`, the manually computed numerical result; and (v) `unit`, the unit associated with the expected numerical output.

5.2. Evaluation Metrics

The system was evaluated using a set of metrics designed to assess retrieval performance, tool selection accuracy, and numerical correctness. The structured design of the ReAct reasoning process, implemented through distinct stages and explicit control flags tracking the progression of reasoning, enabled detailed logging of each agent response. This allowed us to determine whether a question was incorrectly flagged as "non-mathematical", to inspect which formulas were retrieved from the knowledge graph, to identify which formula was selected and instantiated as a computational tool, and to analyze how numerical values from the user query were mapped to the parameters of the selected formula template.
By collecting this fine-grained information for each response, we were able to construct a significantly more informative evaluation framework than a simple aggregate measure of correct answers over total attempts. The final performance metrics adopted are the following:

Formula Recognition Accuracy measures the proportion of mathematical questions for which the agent correctly identified and instantiated the appropriate computational tool. This metric captures the effectiveness of semantic retrieval and agent-level tool selection. Tool recognition outcomes are categorized as: (i) correct tool created, (ii) wrong tool created, or (iii) no tool created.

Tool Failure Statistics quantifies the proportion of mathematical questions for which the system failed to instantiate any computational tool. To enable diagnostic analysis, no-tool cases are further classified into: (i) coverage gaps, where no corresponding tool exists in the system; (ii) agent-level selection failures, where the correct tool was retrieved but not selected; and (iii) retrieval failures, where the correct tool was not retrieved among the top candidates.

Overall Response Accuracy measures the proportion of questions for which the final numerical output matches the expected ground truth value. This metric reflects the end-to-end correctness of the system, including all intermediate steps required to produce a correct response: formula extraction, retrieval, parameter extraction, parameter matching, and deterministic computation.

Conditional Response Accuracy evaluates numerical correctness conditioned on successful tool selection. It is computed as the proportion of correct responses among cases where the correct tool was instantiated. This metric isolates parameter extraction and execution reliability from retrieval performance. It remains indirectly influenced by the initial extraction pipeline, as an incorrect formula extraction may hinder proper execution even when the correct tool has been selected.
Together, these metrics provide a structured and layered evaluation framework that aims to isolate retrieval quality, agent decision-making, and computational correctness, enabling precise identification of failure modes and bottlenecks within the overall pipeline.

5.3. Hardware Configuration

All experiments were conducted on a workstation equipped with:

- *12th Gen Intel(R) Core(TM) i5-12500H (2.50 GHz)* CPU,
- *NVIDIA GeForce RTX 3050 Ti Laptop (4 GB VRAM)* GPU,
- *16 GB* RAM,
- Python version and key library versions used for the pipeline.

The language model used for mathematical knowledge extraction was Ollama's `qwen2-math:7B`. The language model responsible for orchestrating the ReAct reasoning process was Ollama's `Llama3.2:7B`. Both models were executed locally on the workstation.

5.4. Context Window Size and Execution Times

A central design constraint of the system was controlling context-window growth during the multi-step ReAct loop. Since each iteration accumulates system instructions, retrieved metadata, tool schemas, intermediate reasoning traces, and tool outputs, unbounded context expansion would both risk exceeding the model's maximum window and increase inference latency. This is particularly relevant for local inference with small models, where generation time scales with prompt length.
To mitigate this, several constraints were introduced, like heavily limiting the list of formulas retrieved that are presented to the Agent after the graph query stage. Intermediate reasoning traces are kept whole only under a certain threshold and are truncated if they exceed it. These choices keep the context size bounded while preserving the information strictly necessary for decision making.
Empirical instrumentation over a 30-query subsample shows that context size remains stable across steps, growing moderately during tool injection and then stabilizing. Most queries complete in six ReAct steps, with a median end-to-end latency of approximately 50 seconds under local GPU inference. The average per-call model latency is around 5 seconds, and non-LLM overhead (retrieval and tool execution) accounts for only a minor fraction of total runtime.
Overall, these results indicate that the adopted context-control strategy successfully balances informational sufficiency and computational efficiency, preventing prompt bloat while maintaining manageable inference times.

5.5. Results

The evaluation, as described previously, was conducted on a dataset of 600 samples, divided into two domains: Physics (IDs 1–300) and Chemistry (IDs 301–600). Among the total samples, only 7 were incorrectly identified as non-mathematical questions and therefore caused the Agent to skip the knowledge-based response pipeline entirely. Some measures are thus calculated both on the total 600 sample corpus and just on the 593 correctly recognized ones.

Formula Recognition Accuracy Table 1 reports the proportion of cases in which the Agent successfully retrieved and instantiated knowledge from the graph into a tool. A distinction is made between: (i) cases where the correct formula was retrieved and subsequently recognized and synthesized as the correct computational tool; (ii) cases where the correct formula was retrieved but the Agent selected an incorrect alternative among the retrieved candidates; and (iii) cases where the correct formula was not retrieved and consequently no appropriate tool was created.

Formula Recognition Accuracy

	Total (600)	Math-only (593)
Correct Tool	47.50%	48.06%
Wrong Tool	35.17%	35.58%
No tool	16.17%	16.36%

Table 1: Performance of the system in the retrieval from the graph and the correct selection of the formula to synthesize tools

From these results, we observe that the largest portion of the dataset corresponds to correctly retrieved and selected tools (285 out of 600 samples). The second largest portion consists of cases in which the correct formula was successfully retrieved but not selected by the Agent, resulting in the creation of an incorrect tool (211 out of 600 samples).
It is important to emphasize that this selection step is delegated entirely to the language model. The Agent is presented with the top retrieved formulas and prompted to select the most appropriate one for solving the problem. Consequently, errors in the selection process may stem from incomplete or ambiguous metadata associated with the stored formulas, or from limitations in the reasoning capabilities of the language model when disambiguating between similar candidates.
Finally, 97 out of 600 samples resulted in no tool being created during the entire ReAct reasoning process. These cases may originate from multiple causes: absence of the required formula in the knowledge graph (due to upstream extraction errors), failure of the retrieval mechanism, or truncation of the reasoning process after reaching the maximum allowed number of steps. These failure modes are analyzed in greater detail in the subsequent sections. Figure 8 illustrates these three distinct categories of results.

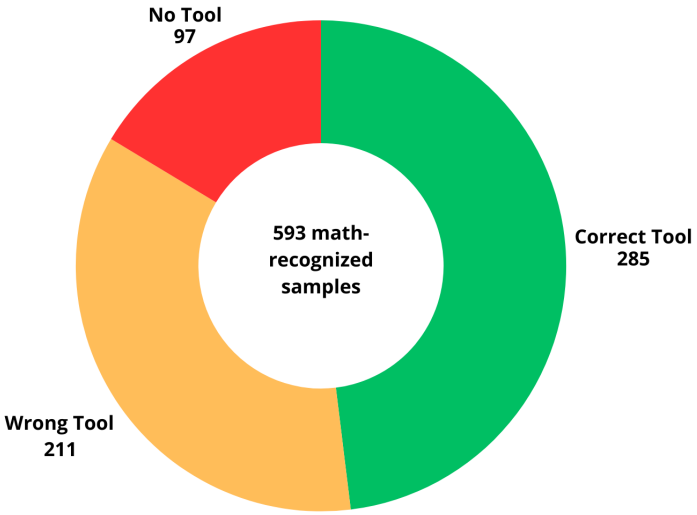

Figure 8: The three distinct Agent's results groups: (red) No Tool created, (yellow) Wrong Tool created, (green) Correct Tool created

5.5.1 Tool Failure Statistics

Table 2 reports the statistics of failed tool creation instances. Since tool creation failures can arise from fundamentally different causes, we distinguish three failure modes: (i) cases where the formula was not present in the knowledge graph, indicating a limitation of the extraction process rather than of the Agent; (ii) cases where the correct formula was retrieved but not selected by the Agent; and (iii) cases where the correct formula was present in the knowledge base but was not retrieved, pointing to a semantic matching failure between the user query and the stored knowledge.

Tool Failure Statistics

	Total (600)	Over the No Tool cases (97)
Not present in the KB	0.83%	5.15%
Retrieved but not used	11.00%	68.04%
Not retrieved	4.33%	26.80%

Table 2: Focus on the Tool failures

Focusing on the no-tool cases, we observe that absence from the knowledge graph represents the least frequent cause (5 out of 97 cases). More than two-thirds of these failures (66 out of 97) occurred in situations where the correct formula was successfully retrieved but not selected by the Agent.
This suggests that the principal bottleneck lies in the decision-making stage rather than in retrieval itself. Potential explanations include repeated retrieval attempts that exhausted the maximum reasoning steps, forcing premature termination of the reasoning process, or inconsistencies in the stored metadata (e.g., corrupted PythonTemplate fields) that prevented successful tool instantiation despite correct formula identification.
Additionally, 26 out of 97 cases correspond to failures in which the correct formula was never retrieved despite being present in the knowledge base. These cases likely result from suboptimal semantic matching during retrieval, possibly due to poorly extracted or inconsistently labeled metadata that increases semantic distance between the query and the stored entity.

5.5.2 Overall and Conditional Response Accuracy

Table 3 presents both the overall response accuracy computed over the full corpus of 600 samples and the conditional accuracy computed only over cases in which the correct tool was created.
The overall metric provides a global assessment of the entire pipeline. In contrast, the conditional metric isolates the execution stage by filtering out failures related to formula extraction and retrieval. It therefore focuses on parameter identification within the user query, parameter-to-template matching, and correct substitution of constants — all of which jointly determine the correctness of the final numerical result.

Overall and Conditional Response Accuracy

	Total (600)	Math-only (593)	Over the Correct Tool cases (285)
Correct Answer	28.00%	28.33%	58.95%

Table 3: Accuracy of the responses given by the system calculated on the total corpus of questions and only on the samples that generated a correct tool creation.

While the overall accuracy across the full dataset is relatively low (28%), the conditional accuracy reveals a substantially different picture. Among the 285 cases in which the correct tool was created, 168 produced a numerically correct result, whereas 117 did not.
This discrepancy highlights that a significant portion of failures occurs prior to tool instantiation. However, even when the correct tool is available, incorrect parameter mapping remains a major source of error. Since the assignment of numerical values from the user message to the corresponding parameters in the formula template is handled by the language model, ambiguously phrased questions or semantically similar parameters may lead to incorrect substitutions.

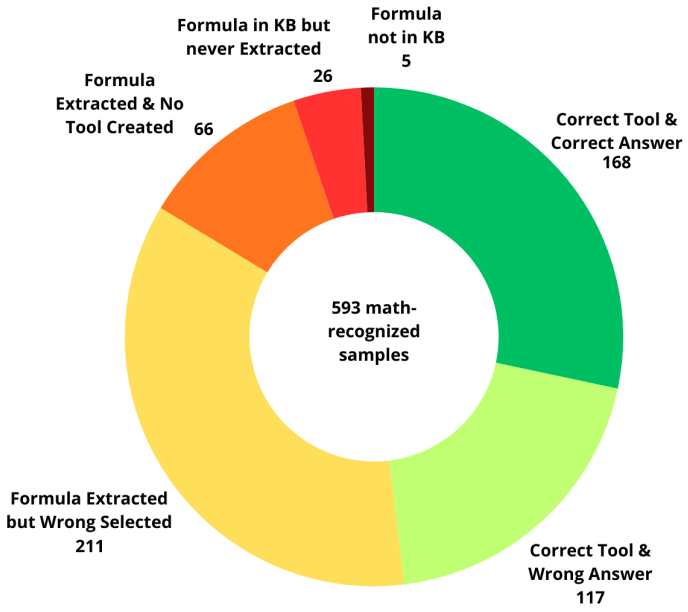

Figure 9: This more fine-grained graph helps to understand the true composition of the whole response dataset. The division in the three main sections is still visible but now are more evident the correct answers and all other sub-categories.

Another possible source of error lies in the upstream extraction pipeline. Misinterpreted operators, misplaced parentheses, or incorrectly reconstructed LaTeX expressions may result in corrupted Python templates, which subsequently propagate errors into the numerical computation stage. Figure 9 visualizes in detail the complete composition of the whole response dataset, in the three main categories and each individual sub-category.

5.5.3 Differences between the Questions' Domains

Finally, we investigated whether performance differences emerged between the two domains. All previously defined metrics were therefore recomputed separately for Physics (IDs 1–300) and Chemistry (IDs 301–600). This domain-specific analysis aims to determine whether structural differences in formula characteristics or document composition influence system performance.
Among the 7 samples that were recognized as non-mathematical, 4 came from the physics subset and 3 from the chemistry one. The results are presented in four tables: a general one illustrating the split among the three categories previously discussed: correct tool cases, wrong tool cases, and no-tool cases, and then one going in detail in each one of the three.

Overall Tool Outcome Distribution by Domain

	Physics (296)	**Chemistry (297)**
Correct Tool Calls	63.33%	31.67%
Wrong Tool Calls	28.67%	41.67%
No Tool Calls	6.67%	25.67%

Table 4: Distribution of tool creation outcomes across Physics and Chemistry domains. Percentages are computed over the total number of samples in each domain.

Correct Tool Cases

	Physics (190)	Chemistry (95)
Correct Tool and Correct Answer	66.84%	43.16%
Correct Tool but Wrong Answer	33.16%	56.84%

Table 5: Performance conditional on correct tool selection.

Wrong Tool Created

	Physics (86)	Chemistry (125)
Wrong Tool Created	28.67%	41.67%

Table 6: Cases where a tool was instantiated but did not correspond to the correct formula.

No Tool Cases

	Physics (20)	Chemistry (77)
Formula Retrieved but No Tool Created	95.00%	61.04%
Formula not Retrieved	0.00%	33.77%
No Knowledge in KB	5.00%	5.19%

Table 7: Breakdown of cases where no computational tool was instantiated.

From these domain-specific results, a substantial performance gap between Physics and Chemistry clearly emerges. Physics consistently exhibits higher tool recognition accuracy and higher conditional response accuracy. In contrast, Chemistry shows a significantly higher rate of wrong tool creation and retrieval failures.
This disparity likely reflects structural differences between the two domains. The Chemistry formulas extracted from the reference document contain more embedded constants, more complex symbolic expressions, and denser parameter schemas, increasing both retrieval ambiguity and parameter-mapping difficulty. Longer and more symbolically dense formulas may place additional stress on the extraction pipeline, making errors in OCR-based template recognition and subsequent conversion into Python templates more likely.
At the same time, formulas requiring a larger number of parameters increase the cognitive load on the downstream language model operating at the core of the Agent. A higher number of parameters implies more variables to identify within the user query and correctly align with the template structure, thereby increasing the probability of incorrect substitutions.
Finally, the characteristics of the source documents themselves should not be overlooked. The Physics cheat sheet contained clearer formulas, fewer embedded constants, and explicit definitions provided immediately alongside the expressions. In contrast, the Chemistry document included many structurally similar formulas (e.g., multiple entropy formulations involving changes in pressure, temperature, or volume). Even assuming correct mapping into the knowledge base, the high semantic similarity among these formulas increases the likelihood of confusion during retrieval and selection, as the language model must discriminate between closely related entities with overlapping contextual descriptions. Figure 10 helps visualize the difference across the two individual domain's results

5.6. Result's Discussion

The experimental results indicate that the main limitation of the system lies in the tool selection stage rather than in the numerical computation itself. A substantial portion of the total errors originates from failures in correctly identifying and instantiating the appropriate formula, while the execution phase proves comparatively

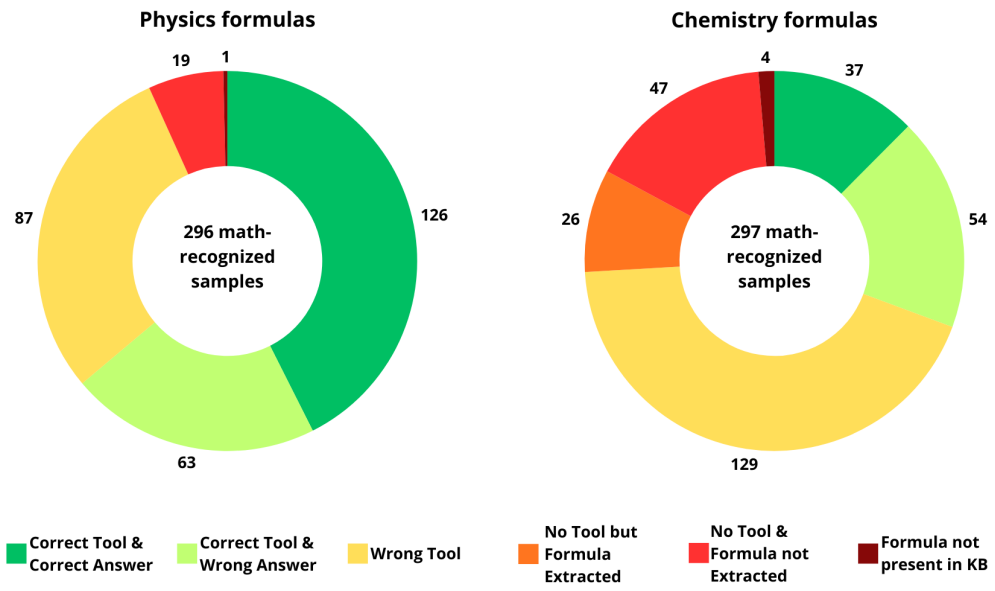

Figure 10: Here is visualized the difference in the results across the two domains: Physics questions and Chemistry questions

reliable once the correct tool has been created. This is confirmed by the sharp increase in accuracy when conditioning on correct tool generation.
The breakdown of failure modes further clarifies this behavior. Errors in the selection of the correct tool do not stem from a single identifiable cause. In several cases, they may originate from the limited amount of information returned to the Agent during retrieval, a design choice intended to prevent excessive context-window usage. This is an essential tradeoff to improve efficiency, but it may also reduce the semantic evidence available to the model when ranking candidate formulas. In other cases, failures may be attributed to corrupted, incomplete, or unclear metadata generated during the upstream extraction process. Imperfect OCR recognition, inconsistencies in parameter descriptions, or inaccuracies in the generated Python templates can all degrade the quality of the retrieved entities and consequently impair correct tool selection.
The comparison between Physics and Chemistry questions further highlights the importance of high-quality document ingestion. Documents that present formulas with clear explanations, explicit parameter definitions, and well-structured contextual descriptions facilitate the construction of richer and more discriminative metadata in the knowledge graph. When multiple formulas are structurally similar or semantically related, the presence of detailed annotations becomes even more critical. The more contextual information and descriptive attributes accompany each formula, the more effectively the graph-building process can disambiguate closely related entities, thereby improving retrieval precision and overall system reliability.
Overall, the experiments demonstrate that the structured tool-based reasoning framework is effective when the correct formula is successfully identified, but system-level performance remains strongly dependent on retrieval precision and robust tool recognition.

6. Conclusions

This work introduced and evaluated a structured Retrieval-Augmented Generation framework for mathematical question answering, in which formulas extracted from documents are stored in a knowledge graph and instantiated as deterministic computational tools. The objective was to combine the interpretability and controllability of symbolic computation with the flexibility of large language models.
The experimental evaluation on a 600-question dataset shows that the proposed architecture expresses significant reliability in the numerical computation phase when the correct formula is selected. Conditional accuracy results demonstrate that deterministic tool execution provides stable and reproducible outcomes, reducing the uncertainty typically associated with purely generative reasoning.
However, the overall system performance is constrained primarily by retrieval and tool selection quality. Most errors originate upstream of execution, either due to incorrect formula ranking, failure to select the correct retrieved entity, or semantic mismatches between question phrasing and stored metadata. These findings high-

light that, in hybrid neuro-symbolic architectures, retrieval precision and structured knowledge consistency are at least as critical as the reasoning capabilities of the language model itself.
In conclusion, the MathRAG framework represents a promising direction for trustworthy mathematical reasoning systems. By separating retrieval, tool synthesis, and deterministic execution, it increases transparency and modularity. Future work should focus on improving semantic matching, refining metadata standardization, and strengthening the interaction between language models and structured knowledge representations in order to further enhance robustness and domain generalization.
Future improvements could aim to further enrich the knowledge graph by incorporating additional entity types and more complex relationships, extending beyond formulas and constants to include theorems, proofs, lemmas, and other mathematical constructs. Enhancing the interaction between graph entities and the retrieval processes would allow the system to access and reason over more complex mathematical domains, supporting a more comprehensive and flexible framework for manipulating mathematical knowledge. Such extensions would move toward a truly all-encompassing neuro-symbolic platform capable of robust, context-aware reasoning across a wide range of mathematical domains.

References

[1] Dang Hoang Anh, Vu Tran, and Le Minh Nguyen. Analyzing logical fallacies in large language models: A study on hallucination in mathematical reasoning. In Yukiko Nakano and Toyotaro Suzumura, editors, *New Frontiers in Artificial Intelligence*, pages 179–195, Singapore, 2025. Springer Nature Singapore.

[2] Rong Bian, Yu Geng, Zijian Yang, and Bing Cheng. Automathkg: The automated mathematical knowledge graph based on llm and vector database. *Computational Intelligence*, 41(4):e70096, 2025.

[3] Wenhu Chen, Xueguang Ma, Xinyi Wang, and William W Cohen. Program of thoughts prompting: Disentangling computation from reasoning for numerical reasoning tasks. *arXiv preprint arXiv:2211.12588*, 2022.

[4] Kenny Davila, Ritvik Joshi, Srirangaraj Setlur, Venu Govindaraju, and Richard Zanibbi. Tangent-v: Math formula image search using line-of-sight graphs. In *European conference on information retrieval*, pages 681–695. Springer, 2019.

[5] Darren Edge, Ha Trinh, Newman Cheng, Joshua Bradley, Alex Chao, Apurva Mody, Steven Truitt, Dasha Metropolitansky, Robert Osazuwa Ness, and Jonathan Larson. From local to global: A graph rag approach to query-focused summarization, 2025.

[6] Guhao Feng, Kai Yang, Yuntian Gu, Xinyue Ai, Shengjie Luo, Jiacheng Sun, Di He, Zhenguo Li, and Liwei Wang. How numerical precision affects arithmetical reasoning capabilities of llms, 2025.

[7] Luyu Gao, Aman Madaan, Shuyan Zhou, Uri Alon, Pengfei Liu, Yiming Yang, Jamie Callan, and Graham Neubig. Pal: Program-aided language models. In *International Conference on Machine Learning*, pages 10764–10799. PMLR, 2023.

[8] Yunfan Gao, Yun Xiong, Xinyu Gao, Kangxiang Jia, Jinliu Pan, Yuxi Bi, Yixin Dai, Jiawei Sun, Haofen Wang, and Haofen Wang. Retrieval-augmented generation for large language models: A survey. *arXiv preprint arXiv:2312.10997*, 2(1), 2023.

[9] Patrick Lewis, Ethan Perez, Aleksandra Piktus, Fabio Petroni, Vladimir Karpukhin, Naman Goyal, Heinrich Küttler, Mike Lewis, Wen-tau Yih, Tim Rocktäschel, Sebastian Riedel, and Douwe Kiela. Retrieval-augmented generation for knowledge-intensive nlp tasks. In H. Larochelle, M. Ranzato, R. Hadsell, M.F. Balcan, and H. Lin, editors, *Advances in Neural Information Processing Systems*, volume 33, pages 9459–9474. Curran Associates, Inc., 2020.

[10] Hongwei Liu, Zilong Zheng, Yuxuan Qiao, Haodong Duan, Zhiwei Fei, Fengzhe Zhou, Wenwei Zhang, Songyang Zhang, Dahua Lin, and Kai Chen. Mathbench: Evaluating the theory and application proficiency of llms with a hierarchical mathematics benchmark, 2024.

[11] Jin Lu and Ji Li. A novel framework for educational qa: Leveraging rag and code interpreters for knowledge retrieval and logical computation. *PLOS ONE*, 20(12):1–23, 12 2025.

[12] Aaron Meurer, Christopher P. Smith, Mateusz Paprocki, Ondřej Čertík, Sergey B. Kirpichev, Matthew Rocklin, AMiT Kumar, Sergiu Ivanov, Jason K. Moore, Sartaj Singh, Thilina Rathnayake, Sean Vig, Brian E. Granger, Richard P. Muller, Francesco Bonazzi, Harsh Gupta, Shivam Vats, Fredrik Johansson,

Fabian Pedregosa, Matthew J. Curry, Andy R. Terrel, Štěpán Roučka, Ashutosh Saboo, Isuru Fernando, Sumith Kulal, Robert Cimrman, and Anthony Scopatz. Sympy: symbolic computing in python. *PeerJ Computer Science*, 3:e103, January 2017.

[13] Jing Qian, Hong Wang, Zekun Li, Shiyang Li, and Xifeng Yan. Limitations of language models in arithmetic and symbolic induction. In Anna Rogers, Jordan Boyd-Graber, and Naoaki Okazaki, editors, *Proceedings of the 61st Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 9285–9298, Toronto, Canada, July 2023. Association for Computational Linguistics.

[14] Vipula Rawte, Amit Sheth, and Amitava Das. A survey of hallucination in large foundation models, 2023.

[15] Timo Schick, Jane Dwivedi-Yu, Roberto Dessi, Roberta Raileanu, Maria Lomeli, Eric Hambro, Luke Zettlemoyer, Nicola Cancedda, and Thomas Scialom. Toolformer: Language models can teach themselves to use tools. In A. Oh, T. Naumann, A. Globerson, K. Saenko, M. Hardt, and S. Levine, editors, *Advances in Neural Information Processing Systems*, volume 36, pages 68539–68551. Curran Associates, Inc., 2023.

[16] Ayush Kumar Shah, Abhisek Dey, and Richard Zanibbi. A math formula extraction and evaluation framework for pdf documents. In Josep Lladós, Daniel Lopresti, and Seiichi Uchida, editors, *Document Analysis and Recognition – ICDAR 2021*, pages 19–34, Cham, 2021. Springer International Publishing.

[17] Kurt Shuster, Spencer Poff, Moya Chen, Douwe Kiela, and Jason Weston. Retrieval augmentation reduces hallucination in conversation. In Marie-Francine Moens, Xuanjing Huang, Lucia Specia, and Scott Wen-tau Yih, editors, *Findings of the Association for Computational Linguistics: EMNLP 2021*, pages 3784–3803, Punta Cana, Dominican Republic, November 2021. Association for Computational Linguistics.

[18] Yuhong Sun, Zhangyue Yin, Qipeng Guo, Jiawen Wu, Xipeng Qiu, and Hui Zhao. Benchmarking hallucination in large language models based on unanswerable math word problem, 2024.

[19] Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, brian ichter, Fei Xia, Ed Chi, Quoc V Le, and Denny Zhou. Chain-of-thought prompting elicits reasoning in large language models. In S. Koyejo, S. Mohamed, A. Agarwal, D. Belgrave, K. Cho, and A. Oh, editors, *Advances in Neural Information Processing Systems*, volume 35, pages 24824–24837. Curran Associates, Inc., 2022.

[20] Xingcheng Xu, Zibo Zhao, Haipeng Zhang, and Yanqing Yang. Principled understanding of generalization for generative transformer models in arithmetic reasoning tasks. In *Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, page 4721–4747. Association for Computational Linguistics, 2025.

[21] Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik R Narasimhan, and Yuan Cao. React: Synergizing reasoning and acting in language models. In *The Eleventh International Conference on Learning Representations*, 2023.

[22] Zheng Zhang. Comprehension without competence: Architectural limits of llms in symbolic computation and reasoning, 2025.

[23] Tianyu Zhao, Yan Huang, Songfan Yang, Yuyu Luo, Jianhua Feng, Yong Wang, Haitao Yuan, Kang Pan, Kaiyu Li, Haoda Li, et al. Mathgraph: A knowledge graph for automatically solving mathematical exercises. In *International conference on database systems for advanced applications*, pages 760–776. Springer, 2019.

A. Appendix A

A.1. Formula Extraction Prompt

```
You are an expert in mathematical knowledge extraction and Python translation.

TASK 1: Extract ALL mathematical formulas from the text below.
For each formula, provide these fields (use empty string if not found):
- latex: complete LaTeX expression
  JSON ESCAPING: Each backslash needs to be a double backslash
  Correct: "\\\\frac{{a}}{{b}}", "\\\\Delta x", "\\\\omega"
  Wrong: "\\frac{{a}}{{b}}", "\\Delta x", "\\omega"
```

```
- label: short descriptive name
- description: brief explanation from text
- intent: one sentence on when/why to use this
- parameters: comma-separated list of ALL variables
  Include: greek letters, subscripts (v_i, v_f)
- parameter_description: what each parameter means
- units: measurement units (if any)
- keywords: comma-separated tags
- python_code: Python expression (Without assignments)
  Correct: "m * c**2", "v_i + a * t", "np.sum(x)"
  Wrong: "E = m * c**2", "result = ..."

TASK 2: Extract constants.
- symbol: variable name (keep exact casing)
- value: numerical value (include all significant digits)
- units
-description
-intent
-keywords

TASK 3: Relationships between two formulas and between formula and constant.
Use ONLY these types: "uses", "derived_from", "generalizes", "specializes", "uses_constant"

For formula-to-formula relationships:
- source_label: first formula label
- target_label: second formula label
- relation_type: "uses", "derived_from", "generalizes", or "specializes"
- confidence: "high", "medium", or "low"
- evidence: brief quote

For formula-to-constant relationships:
- source_label: formula label (from TASK 1)
- target_label: constant symbol as written in TASK 2
- relation_type: "uses_constant"
- confidence: "high" when the constant symbol explicitly appears in the formula's LaTeX,
"medium" when usage is implied by context
- evidence: brief quote or "symbol appears in LaTeX"

Distinguish constants by meaning, not just symbol: if two formulas both contain "c"
but one means speed of light and the other means concentration, output two separate
"uses_constant" entries pointing to the correct constant from TASK 2.

CRITICAL JSON RULES:
1. Double all backslashes: \\\\ not \\
2. Use double quotes: "text" not 'text'
3. No trailing commas
4. Return ONLY: {{"formulas": [...], "relations": [...], "constants": [...]}}

Text:
{chunk}

JSON output:
```

A.2. ReAct Starting Prompt

```
You are an intelligent reasoning agent helping users with mathematical and computational
questions.

USER QUERY:
{state.user_query}
```

```
{workflow_status}

INSTRUCTIONS:
1. First, decide if the query is a math/computation question (requires
formulas/calculations) or not.
- If NOT, use ANSWER_DIRECTLY immediately with a helpful response.
- If YES, follow the workflow: RETRIEVE -> CREATE -> CALL -> ANSWER.

WORKFLOW (for math queries):
1. RETRIEVE_MATH_KNOWLEDGE: Get formulas/constants
2. CREATE_DYNAMIC_TOOL: Build computational tool
3. CALL_TOOL: Execute calculation
4. ANSWER_DIRECTLY: Provide final answer

RESPONSE FORMAT (always use this):
Thought: [Your reasoning]
Action: [Action name]
Action Input: [Valid JSON only, nothing else]

AVAILABLE ACTIONS:
1. RETRIEVE_MATH_KNOWLEDGE
- Required: You MUST provide either a "query" string OR a "search_terms" list.
    * Use "query" when you have a natural language phrase
    * Use "search_terms" when you have a list of keywords
- Optional: You can include a "literals" dictionary if the user provided numbers or
categories.
    * Format: {"literals": {"numeric": [5, 2, 6], "categorical": ["zone A"]}}
- Examples:
    * {"query": "energy consumption formula"}
    * {"search_terms": ["kinematic equations"], "literals": {"numeric": [5, 2, 6]}}
    * {"search_terms": ["angular velocity"]}

2. CREATE_DYNAMIC_TOOL
- Input: {"formula_id": "exact_formula_label_or_uri"}
- CRITICAL: The formula_id must be one of the strings shown in the observation under
"Formulas:".
- Use the exact label as it appears. Do not invent a name.
- Examples:
    * {"formula_id": "Final Angular Displacement"}

3. CALL_TOOL
- Input: {"tool_name": "tool_name", "parameters": {"param1": value1}}
- Use the exact parameter names shown in the tool description.

4. TRANSLATE_INPUT
- Input: {"text": "text", "target_lang": "English"}

5. ANSWER_DIRECTLY
- Input: {"answer": "your answer"}

Available tools:
{tools_description}

RULES:
- For math queries, you MUST follow the workflow in order. Do not skip steps.
- For non-math queries, answer directly.
- Do NOT add extra text or sections. Do NOT write "Observation:".
- When using RETRIEVE_MATH_KNOWLEDGE, your Action Input MUST contain either "query" or
"search_terms". Never leave the input empty.

Start your reasoning now.
```

Abstract in lingua italiana

I Large Language Models (LLM) hanno dimostrato notevoli capacità nella comprensione e nel ragionamento in linguaggio naturale; tuttavia, rimangono inaffidabili nella risoluzione di problemi matematici a causa di fenomeni come allucinazioni, incoerenze aritmetiche e scarsa trasparenza nei passaggi intermedi del ragionamento. Questa tesi propone un framework strutturato di Retrieval-Augmented Generation, denominato *MathRAG*, progettato per migliorare affidabilità e precisione nel rispondere a domande di carattere matematico attraverso l'integrazione tra recupero di conoscenza da un knowledge graph ed esecuzione deterministica di strumenti computazionali.
Il sistema proposto estrae formule da documenti contenenti formule e/o leggi matematiche, le converte in entità strutturate arricchite con metadati semantici e le memorizza in un knowledge graph. Alla ricezione di una query, il framework esegue una ricerca semantica sulle rappresentazioni vettoriali delle formule, seleziona quella più pertinente e sintetizza dinamicamente uno strumento computazionale a partire da un template Python validato. Il calcolo numerico viene quindi eseguito in modo deterministico, separando il ragionamento simbolico dalla generazione linguistica e riducendo il rischio di allucinazioni. L'intero processo di risposta è incapsulato in un ciclo di ragionamento in stile ReAct, così da sfruttare al massimo le capacità di Chain-of-Thought dell'LLM.
Il sistema è stato valutato su un dataset di 600 domande matematiche appartenenti ai domini della Fisica e della Chimica. I risultati mostrano che il principale collo di bottiglia risiede nella fase di riconoscimento e selezione dello strumento, piuttosto che nell'esecuzione numerica. Sebbene l'accuratezza complessiva sull'intero dataset sia pari al 28%, l'accuratezza condizionata supera il 58% nei casi in cui viene istanziato lo strumento corretto, dimostrando la robustezza dell'esecuzione deterministica se la giusta formula é stata selezionata. L'analisi a livello di dominio evidenzia inoltre un significativo divario prestazionale tra Fisica e Chimica, sottolineando l'impatto della struttura delle formule, della coerenza dei metadati e dell'allineamento semantico sull'efficacia del recupero.
I risultati confermano che la combinazione tra rappresentazione strutturata della conoscenza ed esecuzione di strumenti computazionali migliora significativamente l'affidabilità e l'interpretabilità rispetto ad approcci puramente generativi. Tuttavia, le prestazioni del sistema rimangono fortemente dipendenti dalla precisione del recupero, dalla qualità dei metadati e dalla robustezza del processo di ingestione. La tesi conclude che architetture ibride knowledge-grounded come MathRAG rappresentano una direzione promettente per lo sviluppo di sistemi di ragionamento matematico affidabili, a condizione che lavori futuri si concentrino sul miglioramento dell'allineamento semantico, della standardizzazione dei documenti e della coerenza del knowledge graph.

Parole chiave: RAG, GraphRAG, formule matematiche, ReAct, invocazione di strumenti, ragionamento Chain-of-Thought, estrazione di conoscenza

Acknowledgements

I want to thank Professor Marco Brambilla for having allowed me to expand my knowledge in this field. Riccardo Campi and Mathyas Giudici for having provided me useful guidance througout these months in the laboratory and for having built together with the other people in the laboratory a truly engaging and friendly environment where I felt at home from the start and always welcome.