Dynamic Agentic RAG with Pathway

Team 67

Abstract

In this study, we introduce PathLex, an advanced agentic Retrieval-Augmented Generation (RAG) system specifically tailored for the legal domain. Built on Pathway's real-time data processing capabilities and leveraging LLMCompiler's dynamic task planning, PathLex addresses critical limitations in existing legal RAG systems, such as hallucinations, retrieval inaccuracies, and long-context handling. With innovations in chunking, multi-tier replanning, and robust fallback mechanisms, including a human-in-the-loop framework, PathLex ensures precise, context-aware answers with verifiable citations. This work lays the foundation for intelligent automation in high-stakes domains, demonstrating the potential for transformative improvements in legal information systems.

1. Introduction

The legal domain demands precision, context awareness, and traceability, making it one of the most challenging arenas for deploying Retrieval-Augmented Generation (RAG) systems. Despite advances in RAG pipelines, current solutions often fail to address issues such as hallucinations, retrieval inaccuracies, and the ability to handle long-context documents. These limitations undermine the reliability and usability of automated systems in critical legal workflows.

To tackle these challenges, we propose PathLex, an advanced agentic RAG system designed to revolutionize legal question-answering and retrieval processes. Built on the Pathway framework for real-time data processing, PathLex integrates state-of-the-art tools to ensure robust and efficient performance:

1. LLMCompiler (Kim et al. (2024)) for dynamic task decomposition and execution, excelling at multi-hop query resolution and token optimization.

2. VoyageAI's voyage-law-2, a fine-tuned embedding model that captures the semantic richness of legal texts.

3. Pathway's Vector Store and Retriever, ensuring precise retrieval during planning and execution, complemented by the Beam Retriever for fallback redundancy.

PathLex employs a sophisticated multi-stage pipeline to optimize both accuracy and robustness when processing legal texts.

We effectively used recursive chunking along with a powerful embedding model to transform the lengthy texts present in legal documents into manageable segments while preserving the context in a way LLMs can easily take advantage of.

To further enhance query and output refinement, PathLex integrates a Dynamic Replanning mechanism that operates on three tiers. This system includes a human-in-the-loop (HIL) component, which is essential for addressing edge cases; it allows for the rewriting of queries or even the direct generation of answers.

By tackling the limitations found in existing retrieval-augmented generation (RAG) methodologies, PathLex marks a significant advancement in legal information systems.

2. Uniqueness of Solution

In integrating LLMCompiler into a Retrieval Augmented Generation (RAG) workflow, our solution distinguishes itself by addressing several key challenges inherent in traditional RAG setups while leveraging the capabilities of LLMCompiler.

1. Parallel Task Execution Traditional RAG systems often sequentially execute retrieval queries, analyze outputs, and generate responses. This sequential nature introduces latency, especially when dealing with multiple retrievals. LLMCompiler's planner-executor architecture allows us to identify independent retrieval tasks and execute them in parallel.

2. Dynamic Replanning for Retrieval Dynamic replanning is crucial in RAG, as intermediate retrieval results can change the course of subsequent queries or generations, especially in multi-hop queries. LLMCompiler's dynamic execution graph adapts to these changes by recomputing task dependencies based on the results of prior retrievals, making sure that the agent's actions remain contextually relevant.

3. Enhanced Retrieval Precision with Task-Specific Tools

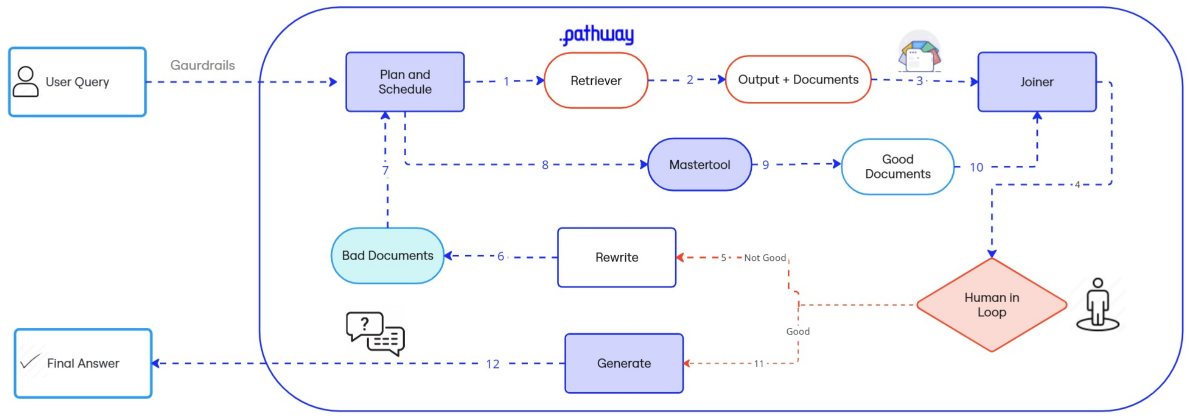


Figure 1. Main Pipeline

LLMCompiler's executor framework integrates specialized retrieval tools. Each retrieval task is dynamically mapped to the most relevant tool based on its requirements.

4. Scalability to Complex Queries The RAG framework often faces challenges with multi-step queries that involve intricate reasoning and retrieval dependencies. LLMCompiler's ability to form directed acyclic graphs (DAGs) for task execution enables efficient management of these dependencies.

5. Plan-and-Solve Alignment RAG benefits significantly from the plan-and-solve strategy inherent in LLMCompiler. Instead of treating retrieval and generation as a monolithic process, tasks are broken down into manageable sub-steps (e.g., retrieval $\rightarrow$ analysis $\rightarrow$ generation), each optimized independently for accuracy and efficiency.

6. Reduced Token Usage and Cost By decoupling the reasoning process from execution, as inspired by ReWOO (Xu et al. (2023)), LLMCompiler minimizes invocations of the LLM which can lead to excessive token consumption

3. Use Case Selection Novelty

The application of agentic Retrieval-Augmented Generation (RAG) in the legal domain is both challenging and impactful. Legal systems require outputs that are highly accurate, contextually relevant, and traceable to reliable sources.

Our approach introduces a novel solution to address the inherent problems in traditional legal RAG systems, including **hallucinations, retrieval inaccuracies, long-context handling, and accurate citation generation**.

Key innovations of our use case include:

1. Re-planning for Hallucination Mitigation: Our system implements a re-planning mechanism to reduce hallucinations. If the output remains unreliable after three re-planning cycles, we invoke a human-in-the-loop (HIL) step for oversight, ensuring correctness and reliability.

2. Error Handling and Fallback Mechanisms: We integrate robust error-handling processes for API and tool calls, preventing execution failures from disrupting the workflow. These mechanisms collectively address hallucination issues while maintaining efficiency and accuracy.

3. Mitigating Retrieval Inaccuracies: During the initial planning phase, we filter out unreliable documents, ensuring only high-quality sources are included. In the event of a re-planning step, previously flagged documents are excluded from subsequent retrievals, reducing the risk of propagating inaccuracies.

4. Efficient Long-Context Management: By using recursive chunking, we address limitations in processing extensive legal documents while preserving their logical structure and coherence.

5. Traceable Citations for Reliable Output: Our document prompts incorporate metadata such as page numbers, paragraphs, sources, and page content. This ensures that generated answers can be accurately traced back to their origin, addressing the challenge of faithfulness to cited sources.

4. Solution Overview

Our proposed solution, PathLex, is an advanced agentic architecture built specifically for legal retrieval-augmented generation (RAG). In its current state, it deals primarily with data in the PDF format, although it can be easily adapted to handle other types of data as well.

Our pipeline begins with text parsing, utilizing the Unstructured library to accomplish this task efficiently. After parsing, we chunk our data using LangChain's built-in **RecursiveCharacterTextSplitter**, which allows us to break down the text into manageable segments. This step is crucial for ensuring that the subsequent steps capture the context and nuance in the documents.

Next, we embed the text using VoyageAI's **voyage-law-2**, a powerful embedding model that has been fine-tuned specifically for legal use cases. This model is designed to capture the semantic meaning of the texts effectively and has a deep understanding of legal jargon, making it the perfect fit for our system.

Our solution has been built with complex, multi-hop queries in mind. To achieve high accuracy on these queries, we chose **LLMCompiler** due to its advanced question decomposition and planning capabilities, which provide an advantage in handling such tasks by creating well-defined plans for each generated sub-query, minimizing the ambiguity present in the query. The joiner then combines these plans, passing on the final outputs to a generate head, which provides the user with the desired answer.

Considering the importance of well-cited sources in the legal domain, we have also integrated citation generation into our system. This allows the user to find the exact source of the information provided to them, down to the page number on which it was found.

5. Dataset

5.1. Contract Law

One of the two domains we focused on for our datasets was contract law, specifically using the Contract Understanding Atticus Dataset (CUAD). This dataset includes over 13,000 labels across 510 commercial legal contracts, which have been manually annotated under the supervision of experienced lawyers. The annotations identify 41 types of legal clauses deemed important in contract reviews related to corporate transactions, including mergers and acquisitions.

5.2. Case Law

We also concentrated on case law as another focus area. We utilized the **AILA Case Docs**, which consist of a collection of Indian criminal and civil cases, along with queries curated for testing retrievals. Additionally, we incorporated a dataset of **Indian law** to evaluate our pipeline on case laws.

6. LLMCompiler

LLMCompiler is an agentic architecture designed to speed up the execution of agentic tasks by utilising eagerly-executed tasks ordered using a DAG. It also saves costs on redundant token usage by reducing the number of calls to the LLM. This is achieved by exploiting multi-threading to execute functions in a parallel manner. This leads to considerable latency reduction, along with improving cost effectiveness. It has 3 main components:

1. Planner: Creates and streams a DAG of tasks. It does this by decomposing questions into simpler sub-queries, which are then dealt with on separate threads with parallel tool calls.

2. Task Fetching Unit: Schedules and executes the tasks as soon as all dependencies are resolved.

3. Joiner: Merges - or *joins* - the outputs from the parallelly executed plans. Based on these merged outputs, it decides whether to generate the final answer or try to find better information via replanning.

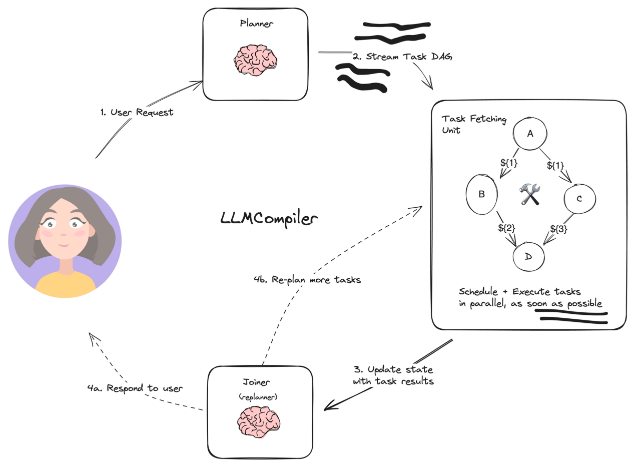


Figure 2. LLM Compiler

7. System Architecture

Our system architecture consists of Pathway's Vector Store and Retriever, LLMCompiler for planning and executing tasks, and Beam Retriever. We use Pathway's Vector Store to store embeddings after parsing and chunking documents in our dataset. The Pathway Retriever is employed for primary retrieval, ensuring relevant documents are fetched during the initial planning and execution phases. LLMCompiler serves as the planner and executor, orchestrating tasks dynamically based on query requirements. It implements token optimization to ensure cost-effectiveness and scalability during retrieval and generation. When Pathway's retriever falls short for a query, Beam Retriever acts as a backup to ensure high-quality results and redundancy in document retrieval.

To address edge cases where the system fails to generate

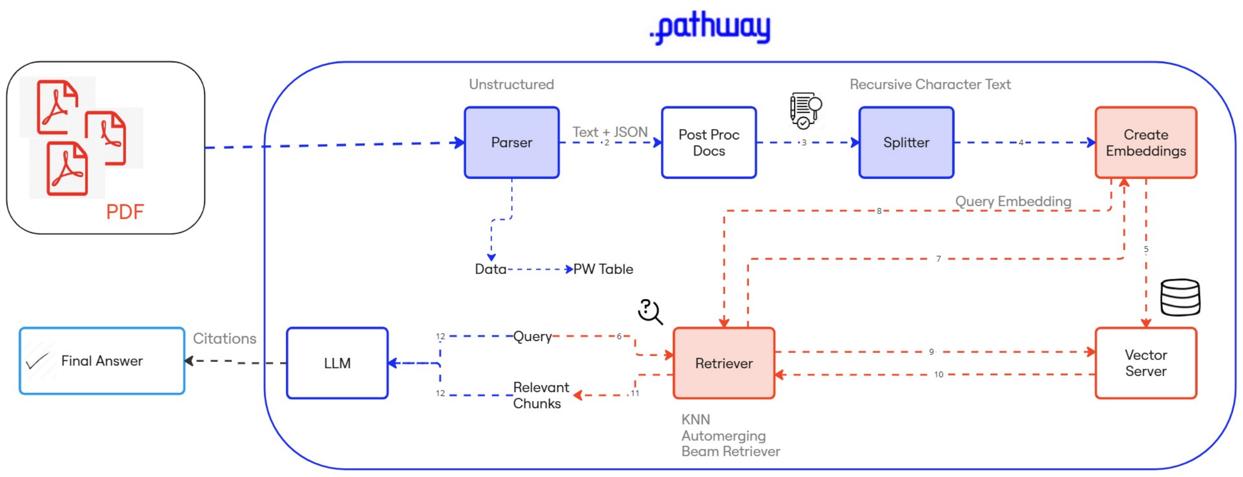


Figure 3. Pathway Architecture

satisfactory results after three replanning cycles, we introduce a human-in-the-loop mechanism. The human operator can instruct the agent to rewrite the query and restart the planning sequence from scratch. If automation continues to fall short, the human operator can directly generate the final answer based on their expertise and findings.

7.1. Parsing

We have developed a custom parser based on the unstructured parser available in Pathway. This custom parser was created to include page numbers and paragraph numbers for citations. We achieve this by carefully inspecting the elements; when an element reaches a certain length, we increment the paragraph number and assign it accordingly. Surprisingly, this custom parser performs faster than both the standard unstructured parser and the OpenParse parser, while also parsing documents more effectively. As a result, it has improved our latency and accuracy.

7.2. Splitting

We have used the RecursiveTextSplitter with a chunk size of 500 with an overlap of 50. We updated the metadata of the chunks after they were split to include their source. Additionally, we used a unique key (UUID) to distinguish between the bad documents and the good ones during retrieval (more on this will be explained later). This key has also been added to the page content after the splitting process. We experimented on various chunk sizes like 1000, 500, 2000 with varying overlap sizes and concluded that 500 performed the best.

7.3. Embeddings

As in any RAG system, embeddings play a crucial role in our system architecture. We required an embedding model that could retain the complexity of legal documents and properly represent the complicated legal terms often present in legal texts. To this end, we used VoyageAI's voyage-law-2 embedding model due to its high performance on various legal embedding benchmarks. While we also considered other legal-specific embedding models like Legal-BERT (Mamakas et al. (2022)), we found them to be too computationally expensive. Additionally, more standard embedding models like OpenAI's embedding models often couldn't represent the semantics of legal texts, leading to poor retrieval.

7.4. Planner and Scheduler

This is the first component of the LLMCompiler used in our pipeline, which we have modified to work well with the RAG use case. The prompt for the planner instructs it to make a plan with tasks with utmost parallelization and analyze the query well to ensure the tasks created are sufficient to answer it. The tasks created are tool uses, all of which end with the calling of **join**, which combines the tools' outputs for further processing. The tasks generated specifically for our use case of RAG decompose the query automatically based on the plan generated by the planner to effectively get the answer to the subparts of the query, which are to be combined into the final answer after join is called.

Replan schedules and invokes the tasks using **ThreadPoolExecuter** to ensure the tool execution occurs in parallel. When all tasks have been executed, it returns their output.

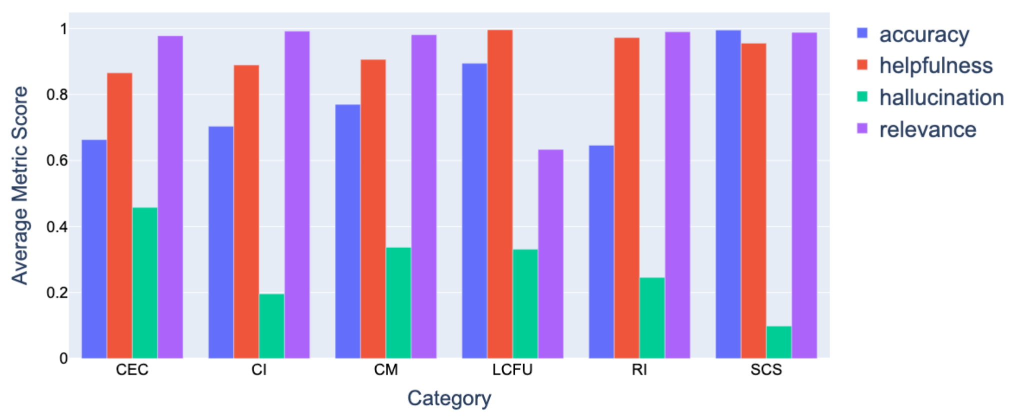


Figure 4. Category Wise Evaluation on CUAD (Voyage)

The Scheduler also ensures that, in case of a re-plan, if there are tasks that are dependent on previously executed tasks, it ensures they are not repeated in the new plan.

7.5. The Joiner Agent

The joiner agent decides whether to proceed with **Generation** or to **Replan** based on the outputs of the tool calls executed by the scheduler. It decides this by using an LLM to analyze the results of the tool calls and create a **Thought** and an **Action** along with feedback in case it decides to replan. This is facilitated well by the use of Pydantic, which ensures the outputs conform to our requirements.

7.6. Human In The Loop

This system includes a human-in-the-loop (HIL) component, which is essential for addressing edge cases. it allows for the rewriting of queries or even the direct generation of answers according to the human feedback.

7.7. Rewrite Agent

If it decides to replan, the joiner then cedes program control to the Rewrite Agent. Thus, the Rewrite Agent receives the **Thought** and **Feedback** from the joiner, based on which it decides to rewrite the query. It has a grade documents function that generates a score for each chunk retrieved by the tool. This allows it to identify "good" documents among the retrieved documents that are relevant to the user query. We do this because a retrieved chunk that doesn't contain enough content to answer the query on its own might answer a part of the query, making it important to retrieve it. Thus, in the output of the Rewrite Agent, we give the good documents back to the planner to generate a replan.

The replanner prompt instructs the replanner to generate a plan that answers the original query based on the feedback it received from its plan. Additionally, it asks the planner to identify what part of the user query is answered by the already retrieved good docs and thus decompose the queries in the plan according to what was not answered.

7.8. Generator Agent

This agent is invoked when the joiner agent decides to go ahead with generating the final answer. This agent reviews the outputs from previous tool calls and the query to generate the final answer that is to be shown to the user.

7.9. Master Tool

This tool serves as a wrapper around our retrieval mechanisms. It helps process all retrieved documents in 3 ways. We initialised top_k at 10, a value we deemed to be optimal through experimentation.

- It filters out all irrelevant documents from the set of retrieved documents. Document irrelevancy is judged using an LLM-as-a-judge.
- It dynamically increases the top_k parameter of our retriever as per the following formula: $top\_k = n + 3$, where n refers to the number of irrelevant documents.
- It implements retriever fallbacks, switching to a backup retriever if the primary retriever is either unable to retriever the relevant documents or returns an error

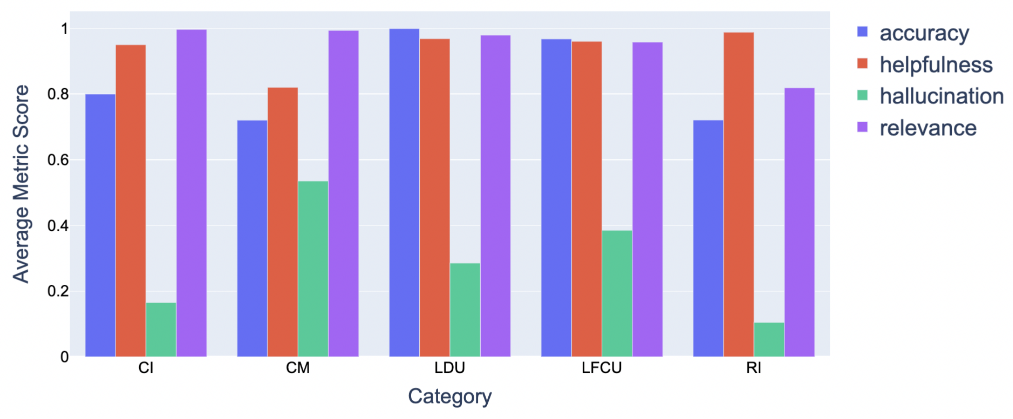


Figure 5. Category Wise Evaluation on AILA (Voyage)

7.10. Automerging Retriever

Automerging retriever addresses the limitations of traditional retrievers, which return disjoint snippets of the documents, by recursively merging leaf nodes that reference a parent node beyond a specified threshold to improve response synthesis. The system parses documents hierarchically, linking smaller chunks to larger parent chunks. However, this approach can be computationally expensive, particularly with models like GPT-4 and integration with Pathway proved to be a big challenge.

7.11. Beam Retriever

Beam Retriever(Zhang et al. (2024)) is a cutting-edge retrieval mechanism that specializes in **multi-hop queries**. It is the highest-rated retriever at the **HotpotQA** website. It models the entire retrieval process as an optimization problem involving an encoder and two classification heads. Its novelty lies in its utilization of the beam search paradigm, which has been employed to retrieve relevant documents at each hop.

We implemented our own version of Beam Retriever for multi-hop queries. We further fine-tuned the **LegalBERT transformer** on queries made on CUAD and Indian Acts, using it as a scoring head. At each hop, the retriever maintains a set of **B most** relevant hypotheses, where each hypothesis represents a chain of documents. For every hypothesis, we concatenate the current chain of documents with each document not yet included in the chain. Using our fine-tuned transformer, we then score the resulting combinations to evaluate their relevance to the query. This iterative process is repeated over K hops, progressively refining the retrieval output.

The beam retriever proved to be highly effective in consistently retrieving the correct chunks. However, it was quite **computationally expensive**. Since the retrieval logic in the main pipeline was already performing well, we decided to use the beam retriever as a fallback mechanism instead.

7.12. Cohere Reranker

Most of the post-retrieval processing includes the usage of rerankers. While experimenting with Pathway's Retriever, we discovered that integrating Cohere's Reranker produced better results. **Cohere**'s reranking tool Cohere (2024) uses advanced language models to reorder lists based on contextual relevance, enhancing search accuracy and content discovery.

7.13. Citation Generation

When delivering text chunks to the LLM, we incorporate the page number, source, and page content into the document prompt. This detailed information allows us to trace the origin of the answer accurately. The citations (Qian et al. (2024)) are correspondingly extracted from the retrieved chunks and the retrieved chunks along with the query and answer are presented to the LLM to generate in-context citations along with the relevant main citations.

8. Results and Metrics

8.1. Query Generation

We have meticulously curated over 100 queries from a variety of documents within the CUAD datasets, ensuring that each query is thoughtfully designed to evaluate distinct components of our pipeline.

Additionally, more than 40 queries have been manually developed based on Indian case law to evaluate our pipeline on case law docs.

Furthermore, we utilized GiskardAI to generate additional queries. Giskard is an open-source Python library that efficiently identifies performance, bias, and security issues in AI applications. This versatile library supports applications based on large language models (LLMs), including retrieval-augmented generation (RAG) agents, as well as traditional machine learning models for tabular data.

8.2. Evaluation Metric

We used LangSmith to evaluate our responses. LangSmith employs a language model-based evaluation to assess various metrics. This is done by using various prompts and sending the answer along with retrieved documents to the language model. It assigns a score of 0 or 1 based on the evaluation of the answer. The metrics are:

1. Hallucination: How grounded is the answer with respect to the reference context?

2. Accuracy: How similar is the generated answer to the ground truth?

3. Helpfulness: How helpful is the answer to the question?

4. Relevance: How relevant are the documents to the question?

9. Experimentation

9.1. Latency and Cost Analysis

We have analyzed our pipeline in terms of parallelization and serialization. The original LLMCompiler framework, which utilizes parallelization, reduces token usage by minimizing the number of calls made to the LLM. This is accomplished through multi-threading, allowing functions to execute simultaneously. As a result, we experience a significant reduction in latency and improved cost-effectiveness. When parallelization is disabled in LLMCompiler, costs nearly double, and processing time increases substantially since tasks are executed sequentially.

9.2. Hierarchical Chunking

Hierarchical chunking is a highly effective text segmentation technique that captures the intricate structure within written content. Unlike conventional methods that simply divide text into basic units, this approach analyzes the relationships between various elements. It segments the text into meaningful components, reflecting different levels of hierarchy such as sections, subsections, paragraphs, and sentences. This method provides a powerful means of splitting text so that inherent relationships between chunks are retained. We didn't use this chunking method because some chunks were too small, resulting in worse retrieval performance.

9.3. Meta Chunking

Meta-chunking(Zhao et al. (2024)) functions at both the sentence and paragraph levels. It identifies groups of sentences within a paragraph, known as meta-chunks, which share deep linguistic and logical connections, surpassing mere semantic similarity. The technique primarily uses Perplexity (PPL) Chunking, which evaluates the perplexity of each sentence based on its context to determine the boundaries of these chunks. A dynamic merging strategy is applied to balance fine-grained and coarse-grained chunking. However, we did not include this method in our final pipeline because our experiments revealed that the sizes of the meta-chunks were relatively small (with an average chunk length of 30), resulting in poor performance for our retrieval tasks.

9.4. Lumber Chunking

Lumber Chunking(Duarte et al. (2024)) method employs an LLM to dynamically segment documents into semantically independent chunks. The chunker first divides the document into paragraphs. Each paragraph is sequentially concatenated into a group until the collective length of the paragraphs exceeds a predefined value. This group is sent to an LLM, which marks the first paragraph where the semantic meaning changes the most. This detection marks the end of the chunk, and the remaining document keeps being sequentially partitionWe found this method to be very costly as we had to remake the server repeatedly, so we decided against using this chunking method in our pipeline.

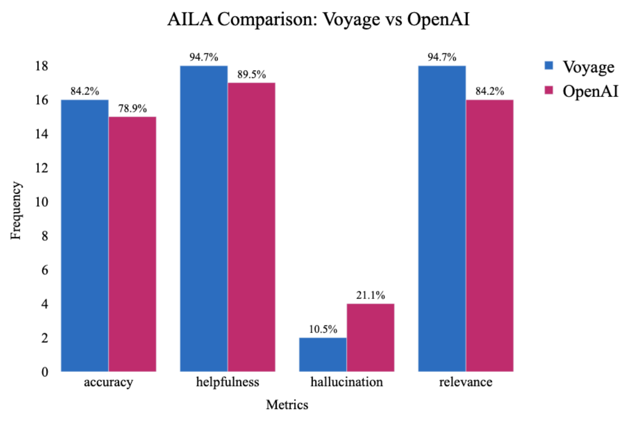


Figure 6. Evaluation on AILA

9.5. Contextual Embedding

Contextual Embedding by Anthropic (Anthropic (2024)) enhances text chunks by adding summaries to each one, which simplifies the retrieval process. This method can be effective with a retriever like BM-25 that relies on semantic similarity. However, we experimented with this approach

Queries Generated	Query Type	Query Example
36	Clause Identification	Who was the petitioner in the case 'Masud Khan v. State of Uttar Pradesh'?
27	Reasoning and Inference	Why did A. K. Srinivasan, Advocate, file a petition before the Kerala High Court?
15	Long-Form Context Understanding	How did the Supreme Court modify the compensation awarded to Joginder?
18	Clause Matching	What did the Resolution of the Sanctioning Authority reveal about Mohd. Iqbal Ahmed?
12	Specific Conditions and Query	What was the nature of the injury caused to Rangnath?
42	Legal Dispute Understanding	What was the dispute between the appellant's family and the deceased brothers?

Table 1. Simulated Case Law Queries Dataset for Evaluation

Queries Generated	Query Type	Query Example
38	Clause Identification	What licensing rights does GlobalWeb have regarding AlphaTech's trademarks?
42	Reasoning and Inference	How would DAE Group's bankruptcy affect its endorsement deal with Kathy Ireland Inc.?
25	Long-Form Context Understanding	Where are notices to the American Diabetes Association be sent under Newegg agreement?
34	Clause Matching	What are the exclusivity restrictions on the type of content displayed in the theatres?
24	Specific Conditions and Scenarios	What happens if CONSULTANT finds MANDATORY PRODUCTS unsuitable for use?
30	Clause Extraction with Context	State the responsibilities of each party in managing campaigns for Co-Branded Applications?

Table 2. Simulated Conract Law Queries Dataset for Evaluation

and found it to be quite costly, so we decided not to include it in the main pipeline.

9.6. Web Scraper

We experimented with using a web scraping tool designed to retrieve information from the internet when relevant documents are unavailable in the vector store or database. It first performed a search query using DuckDuckGo, which identified a suitable website URL based on the user's input. Once the URL was inferred, the tool sent a request to the target website and extracted text content, specifically paragraphs, using the BeautifulSoup library for HTML parsing. These paragraphs were then compiled into a structured format, including metadata such as the inferred URL, the number of paragraphs retrieved, and any errors encountered during the process.

While we saw potential in using this tool as an effective fallback mechanism in the event we weren't able to find the necessary information in the provided documents, the highly specific nature of most cases and the queries about them made it highly computationally expensive to crawl the web and then extract the necessary answer from the large amount of data scraped.

10. Challenged Faced and Solutions

10.1. Open Source Models

Integrating the pipeline with open-source models through OpenRouter proved to be challenging. We encountered issues such as rate limits and model changes, which sometimes necessitated adjustments to prompts and resulted in unsatisfactory outcomes. To bypass the rate limits, we also experimented with using open-source models via Ollama. However, due to limited computing resources, we were unable to run models larger than 8 billion parameters, which significantly hindered our ability to compete with OpenAI's GPT-4o and produce acceptable results. This situation raised questions about whether the shortcomings were due to the model's performance or the pipeline itself. While using GPT-4o was straightforward, it was also very costly. We struggled to find a balance between managing expenses and addressing ongoing errors. Additionally, there were occasions when the tool usage and Langgraph were not compatible with an open-source model.

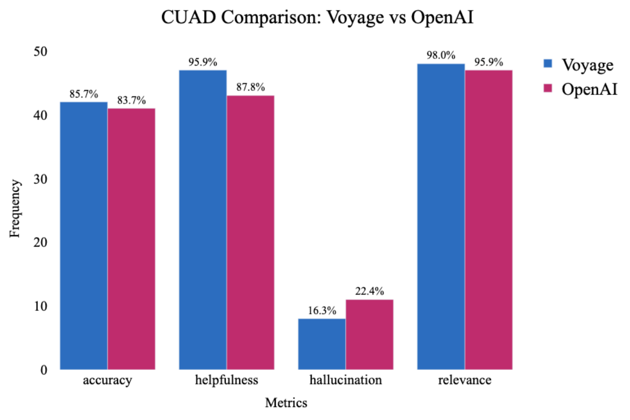


Figure 7. Evaluation on CUAD

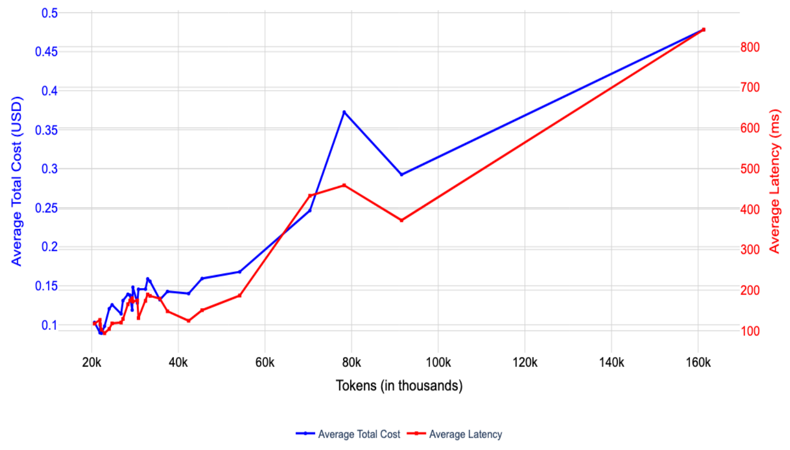


Figure 8. Average Total Cost and Latency v/s Tokens

10.2. Metadata Filters

We faced significant challenges with the metadata filter in our pipeline. A key aspect of our process was to eliminate irrelevant chunks after replanning, and we aimed to achieve this using metadata filters. However, the metadata filter required a JMES path, which failed to resolve to a valid path. As a result, the server kept crashing whenever an incorrect metadata filter was applied. This not only caused major disruptions, but also incurred high costs, as we had to rebuild the server from scratch.

10.3. Output Structuring of LLMs

Ensuring that large language models (LLMs) conform to a specific output format remains a significant challenge. This issue is particularly evident with open-source models, which often struggle to produce the outputs we expect. We utilize Pydantic to validate these outputs, but open-source models don't perform as well as OpenAI's GPT-4o. We also experimented with the Outlines library and SGLang, but we encountered implementation challenges since we used Ollama to run the open-source models, and support for that is limited in these libraries.

10.4. Adding Documents: Missing implementations

We aimed to implement a feature that would allow users to add documents directly from the frontend. However, we discovered that Pathway's client does not provide a method for adding documents to the vector store. Additionally, the LangChain community vector store library is incomplete and does not include an implementation for the `add_doc` function.

10.5. Problems with Streamlit

One major limitation of Streamlit is the lack of modularization, as it triggers a full-page rerun on each interaction. While workarounds for this are on Streamlit's development roadmap, this limitation posed significant challenges for us.

We wanted to dynamically display the thought process of the LLM compiler to the user, but Streamlit's full-page reruns made this difficult to achieve seamlessly. As a result, we opted to use the MERN stack instead. Integrating MERN with Python wasn't straightforward either. We had to implement caching mechanisms and use WebSockets to ensure efficient communication between the frontend and backend.

11. Resilience to Error Handling

While building our vector store using Voyage and OpenAI embeddings, we encountered a challenge: whenever the internet connection was lost, the API connection error would cause the server to shut down. This required us to restart the server each time, which was highly inconvenient during testing. To address this, we modified the source code of the Voyage and OpenAI libraries. We added error-handling logic with try and except blocks, ensuring that instead of immediately throwing an error, the system would gracefully wait for the internet connection to be restored. This change allowed us to continue testing without needing to recreate the server repeatedly.

Additionally, we have implemented fallback mechanisms to handle errors associated with LLM calls effectively. These errors may include, but are not limited to, incorrect API keys and connection issues. As a contingency, we use Anthropic's Sonnet model, which activates whenever OpenAI's

GPT-4o encounters any errors. Although these are the models currently in use, our pipeline is highly flexible, allowing for the integration of other models, such as LLaMa, into our system.

In the process, we discovered that LLMCompiler by itself is designed for OpenAI models, and adding support for other models, including Anthropic's Claude 3.5 Sonnet, proved to be difficult. Considering that our pipeline depends on Langchain components, ensuring alternative models conform to Pydantic requirements, and valid input message formats are sent to the the model proved to be challenging but ultimately resolved. We ended up adding support for Anthropic's Claude models in LLMCompiler as a fallback incase OpenAI's API fails for any reason.

12. User Interface

12.1. MERN Stack

The User Interface (UI) was developed using the MERN stack, which includes MongoDB, Express.js, React.js and Node.js. We chose this stack for its flexibility, scalability, and ability to support seamless full-stack JavaScript development.

The back-end was built with **Node.js** and **Express.js**, using RESTful APIs for smooth front-end integration. We used **MongoDB** for efficient data storage and real-time updates, with collections for users, chats, and script responses.

12.2. Inter-process Communication

We use text files as a cache to enable communication between the Python backend and the JavaScript frontend. The Python pipeline writes intermediate results, including the LLM's thought process, into these files. On the frontend, React uses **WebSockets** to keep an eye on these files. Whenever a file is updated, the displayed text is refreshed in real time. This creates an interactive and transparent experience, allowing users to follow the LLM's reasoning as it happens.

12.3. Session Management

Our user interface includes a login page that prompts users for their email address and password. This functionality allows users to seamlessly access their previous chats with PathLex, effectively managing each user's session. The backend utilizes MongoDB to facilitate this process. Additionally, users have the option to create new chats and view their past conversations.

12.4. State Management

We utilized the state management features available in the React framework. State management in React involves handling the state of components in a predictable and efficient way. This process includes tracking, updating, and sharing data (or state) across the application as it changes over time. Effective state management ensures that the application behaves consistently and responds to user interactions and other events as intended.

13. Responsible AI Practices

Our RAG pipeline incorporates a two-pronged approach to content moderation. We primarily use OpenAI's content moderation model to evaluate user queries for potentially harmful content across multiple categories based on an optimal threshold set by us determined by stress testing on different types of queries and queries exceeding this threshold are blocked by the system. Additionally, we implement a fallback mechanism utilizing Anthropic's Claude model that assesses risk based on predefined unsafe categories and assigns a risk level to user queries, and high-risk queries are blocked by the system. The flagged queries are denied with a standardized response, maintaining compliance with the terms of service and ethical guidelines.

14. Lessons Learned

We discovered that integrating existing codebase, such as Langgraph, for a new purpose like Retrieval Augmented Generation (RAG) is not only challenging but also highly rewarding. Throughout this process, we deepened our understanding of llamaindex, langchain, and the Pathway frameworks. We realized that retrieval heavily depends on its subcomponents, including chunking, parsing, and embeddings. The Pathway framework gave us a unique opportunity to experiment with a real-time data pipeline. Additionally, while developing the user interface for this project, we applied important computer science concepts, such as inter-process communication, which was truly incredible.

15. Conclusion

We introduce PathLex, an innovative end-to-end retrieval-augmented generation (RAG) system designed for legal applications. PathLex marks a significant advancement in both legal AI systems and open-domain RAG. It employs a powerful conglomeration of cutting-edge AI tools to achieve high accuracies in addressing complex queries. Experimental results on datasets covering contract and case law demonstrate its effectiveness in answering challenging, multi-step legal questions. In general, PathLex serves as a sophisticated solution for the difficult task of legal RAG.

References

Anthropic (2024). Introducing contextual retrieval. Accessed: 2024-11-12.

Cohere (2024). Introducing rerank 3: A new foundation model for efficient enterprise search & retrieval. Accessed: 2024-11-12.

Duarte, A. V., Marques, J., Graça, M., Freire, M., Li, L., and Oliveira, A. L. (2024). Lumberchunker: Long-form narrative document segmentation.

Jiang, Z., Sun, M., Liang, L., and Zhang, Z. (2024). Retrieve, summarize, plan: Advancing multi-hop question answering with an iterative approach.

Kim, S., Moon, S., Tabrizi, R., Lee, N., Mahoney, M. W., Keutzer, K., and Gholami, A. (2024). An llm compiler for parallel function calling.

Mamakas, D., Tsotsi, P., Androutsopoulos, I., and Chalkidis, I. (2022). Processing long legal documents with pre-trained transformers: Modding legalbert and longformer.

Qian, H., Fan, Y., Zhang, R., and Guo, J. (2024). On the capacity of citation generation by large language models.

Xu, B., Peng, Z., Lei, B., Mukherjee, S., Liu, Y., and Xu, D. (2023). Rewoo: Decoupling reasoning from observations for efficient augmented language models.

Zhang, J., Zhang, H., Zhang, D., Liu, Y., and Huang, S. (2024). End-to-end beam retrieval for multi-hop question answering.

Zhao, J., Ji, Z., Feng, Y., Qi, P., Niu, S., Tang, B., Xiong, F., and Li, Z. (2024). Meta-chunking: Learning efficient text segmentation via logical perception.

16. Appendix

The datasets we have used for Contract Law: CUAD and the datasets we have used for Case Law: AILA Dataset, Indian Acts

We carried out experimentation on CUAD and AILA datasets to draw out comparisons between OpenAI and Voyage Embeddings. Here we've provided the results for OpenAI embeddings, which we did not end up using, as voyage-3 produced better results. We studied the MTEB leaderboard to analyse various embedding models used specifically for legal retrieval. We couldn't use vstackai-law because of it's closed source nature. So, we opted for voyage-law-2 and voyage-3 which clearly produced better results when compared to OpenAI's text-embedding-3-large

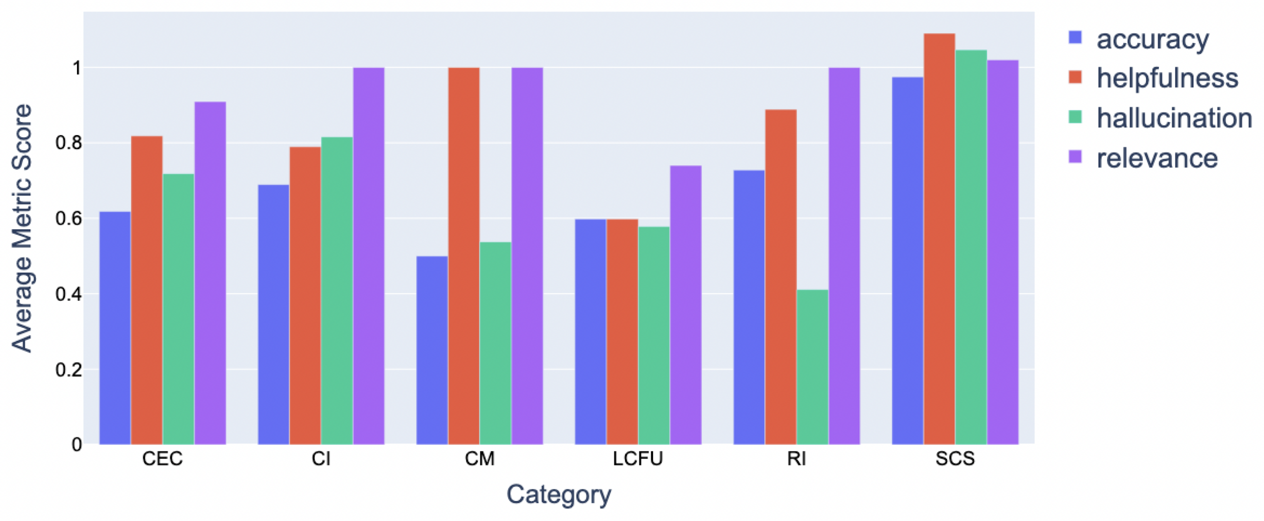


Figure 9. Category Wise Evaluation on CUAD (OpenAI)

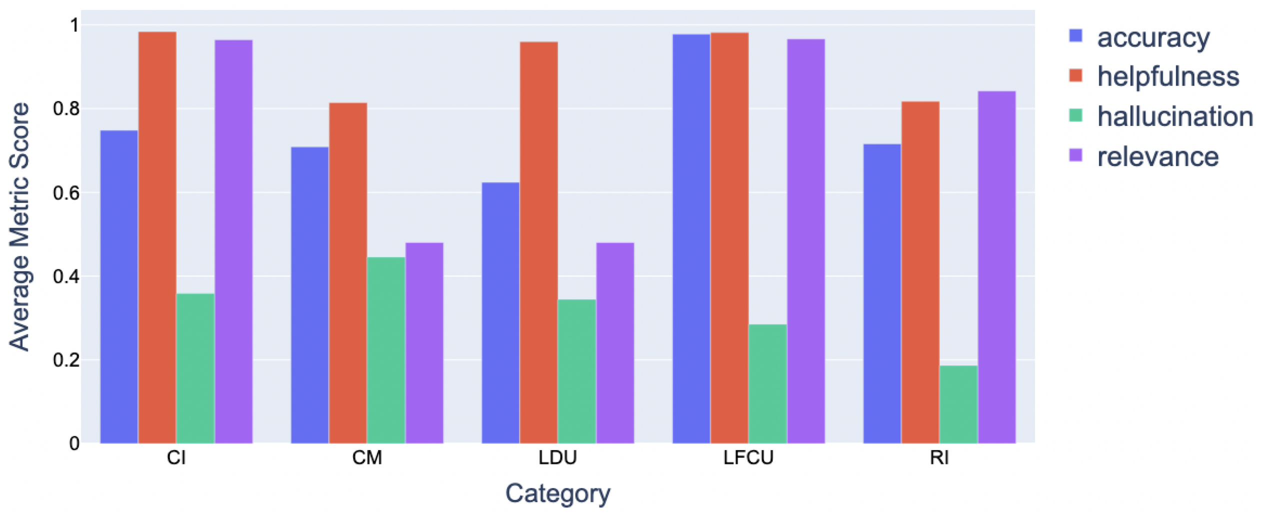
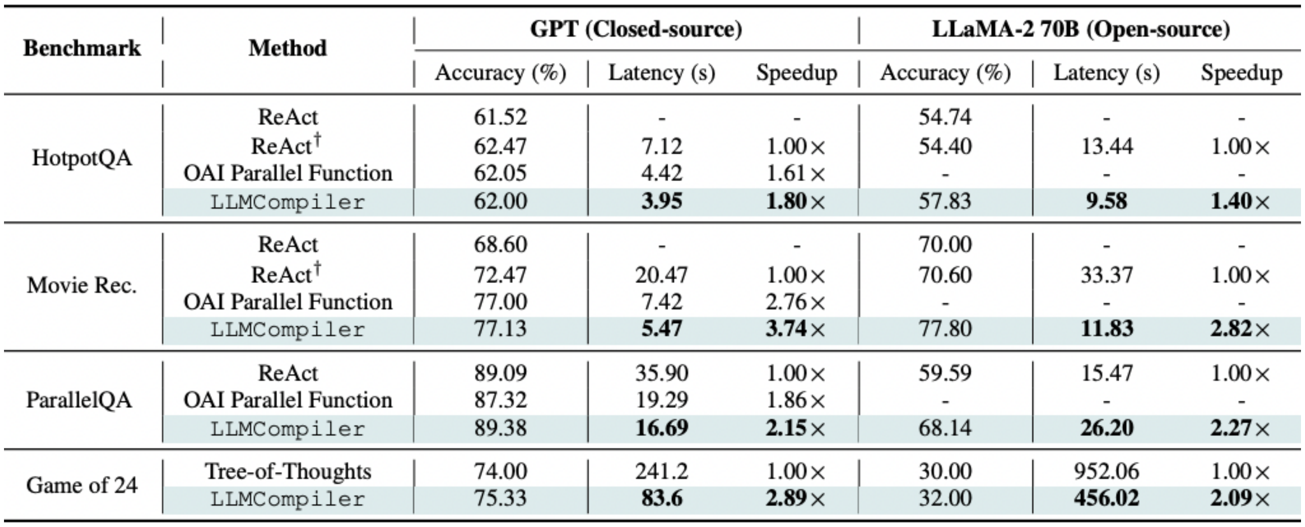


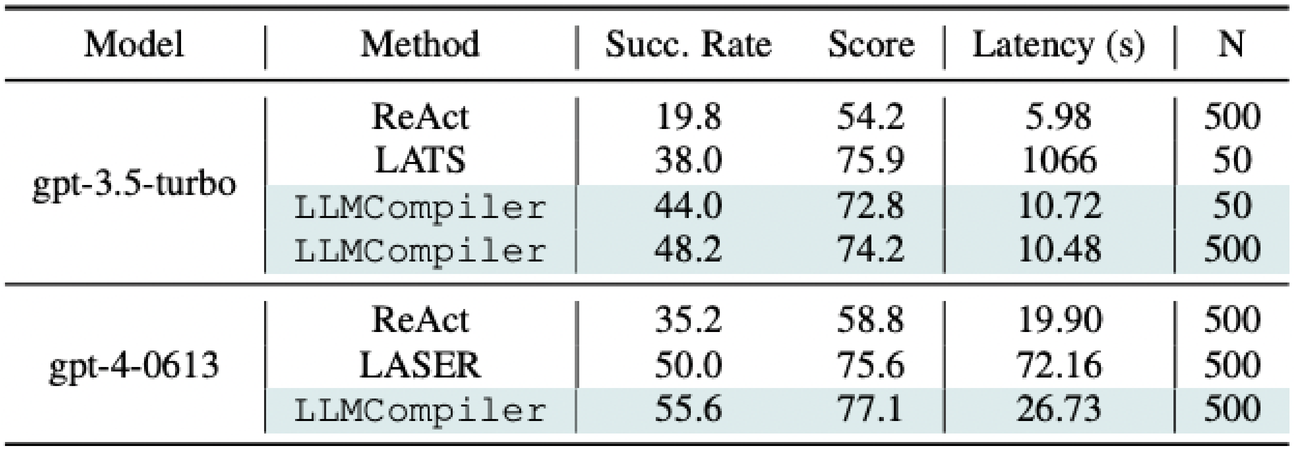
Figure 10. Category Wise Evaluation on Case Law (OpenAI)



Benchmark	Method	GPT (Closed-source)			LLaMA-2 70B (Open-source)		
		Accuracy (%)	Latency (s)	Speedup	Accuracy (%)	Latency (s)	Speedup
HotpotQA	ReAct	61.52	-	-	54.74	-	-
	ReAct[†]	62.47	7.12	1.00×	54.40	13.44	1.00×
	OAI Parallel Function	62.05	4.42	1.61×	-	-	-
	`LLMCompiler`	62.00	**3.95**	**1.80**×	57.83	**9.58**	**1.40**×
Movie Rec.	ReAct	68.60	-	-	70.00	-	-
	ReAct[†]	72.47	20.47	1.00×	70.60	33.37	1.00×
	OAI Parallel Function	77.00	7.42	2.76×	-	-	-
	`LLMCompiler`	77.13	**5.47**	**3.74**×	77.80	**11.83**	**2.82**×
ParallelQA	ReAct	89.09	35.90	1.00×	59.59	15.47	1.00×
	OAI Parallel Function	87.32	19.29	1.86×	-	-	-
	`LLMCompiler`	89.38	**16.69**	**2.15**×	68.14	**26.20**	**2.27**×
Game of 24	Tree-of-Thoughts	74.00	241.2	1.00×	30.00	952.06	1.00×
	`LLMCompiler`	75.33	**83.6**	**2.89**×	32.00	**456.02**	**2.09**×

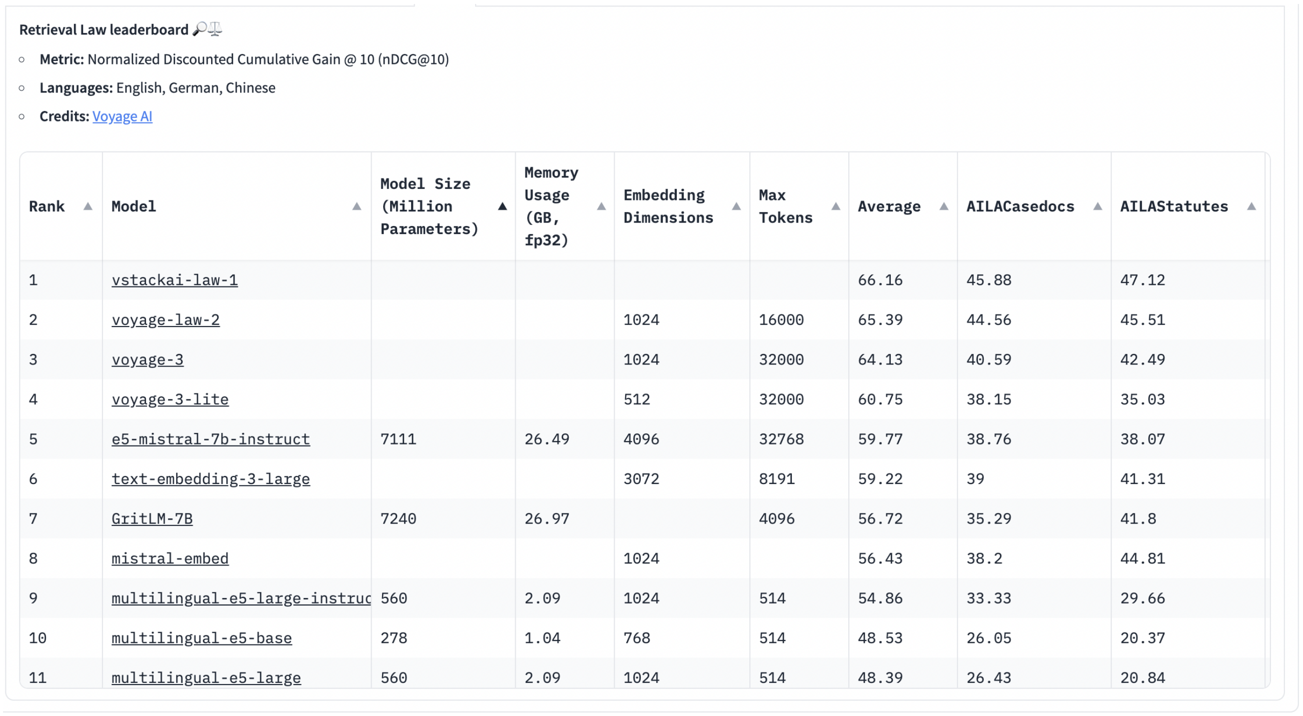
Figure 11. Evaluation of GPT v/s LLaMA-2 70B on LLMCompiler

From the evaluation carried out in Kim et al. (2024) and further experiments carried out by us, we came to a conclusion that GPT performs significantly better than most open-source models on the LLMCompiler framework. We carried out experiments using Ollama, OpenRouter, Groq, etc., and couldn't find an open-source model that performed better than GPT. Moreover, we found that *gpt-4-0613* produced more accurate results compared to *gpt-3.5-turbo*, with a little latency overhead. As our main focus was on achieving the highest accuracy possible, we sticked to *gpt-4-0613*



Model	Method	Succ. Rate	Score	Latency (s)	N
gpt-3.5-turbo	ReAct	19.8	54.2	5.98	500
	LATS	38.0	75.9	1066	50
	`LLMCompiler`	44.0	72.8	10.72	50
	`LLMCompiler`	48.2	74.2	10.48	500
gpt-4-0613	ReAct	35.2	58.8	19.90	500
	LASER	50.0	75.6	72.16	500
	`LLMCompiler`	55.6	77.1	26.73	500

Figure 12. Comparison of gpt-3.5-turbo and gpt-4-0613



Retrieval Law leaderboard 🔎⚖️

- **Metric:** Normalized Discounted Cumulative Gain @ 10 (nDCG@10)
- **Languages:** English, German, Chinese
- **Credits:** Voyage AI

Rank ▲	Model ▲	Model Size (Million Parameters) ▲	Memory Usage (GB, fp32) ▲	Embedding Dimensions ▲	Max Tokens ▲	Average ▲	AILACasedocs ▲	AILAStatutes ▲
1	vstackai-law-1					66.16	45.88	47.12
2	voyage-law-2			1024	16000	65.39	44.56	45.51
3	voyage-3			1024	32000	64.13	40.59	42.49
4	voyage-3-lite			512	32000	60.75	38.15	35.03
5	e5-mistral-7b-instruct	7111	26.49	4096	32768	59.77	38.76	38.07
6	text-embedding-3-large			3072	8191	59.22	39	41.31
7	GritLM-7B	7240	26.97		4096	56.72	35.29	41.8
8	mistral-embed			1024		56.43	38.2	44.81
9	multilingual-e5-large-instruc	560	2.09	1024	514	54.86	33.33	29.66
10	multilingual-e5-base	278	1.04	768	514	48.53	26.05	20.37
11	multilingual-e5-large	560	2.09	1024	514	48.39	26.43	20.84

Figure 13. MTEB leaderboard for law embeddings

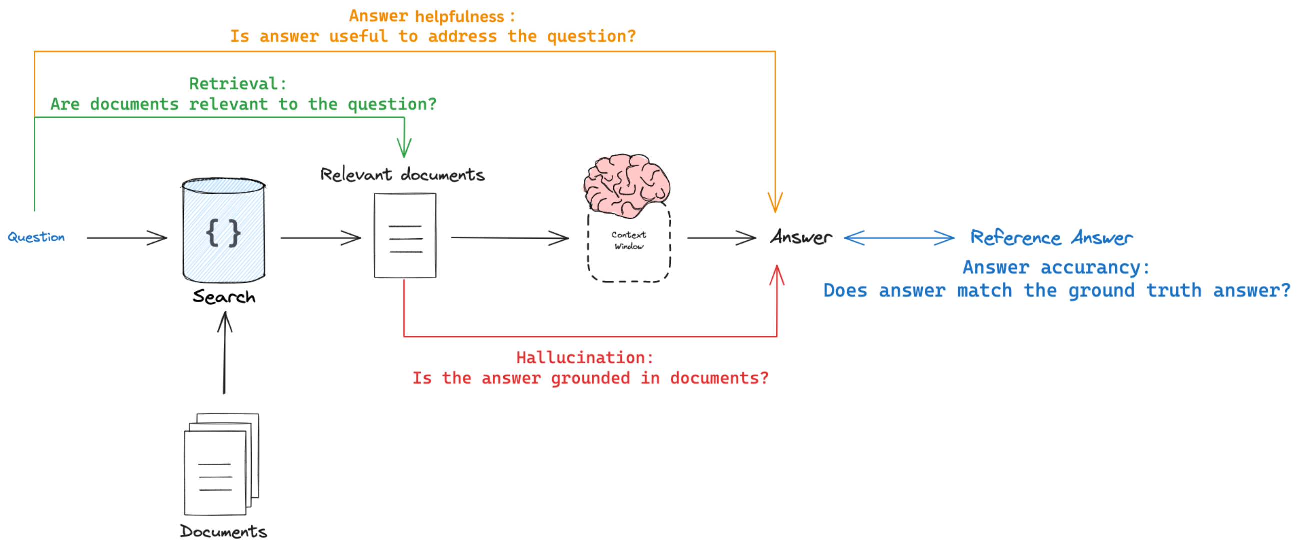
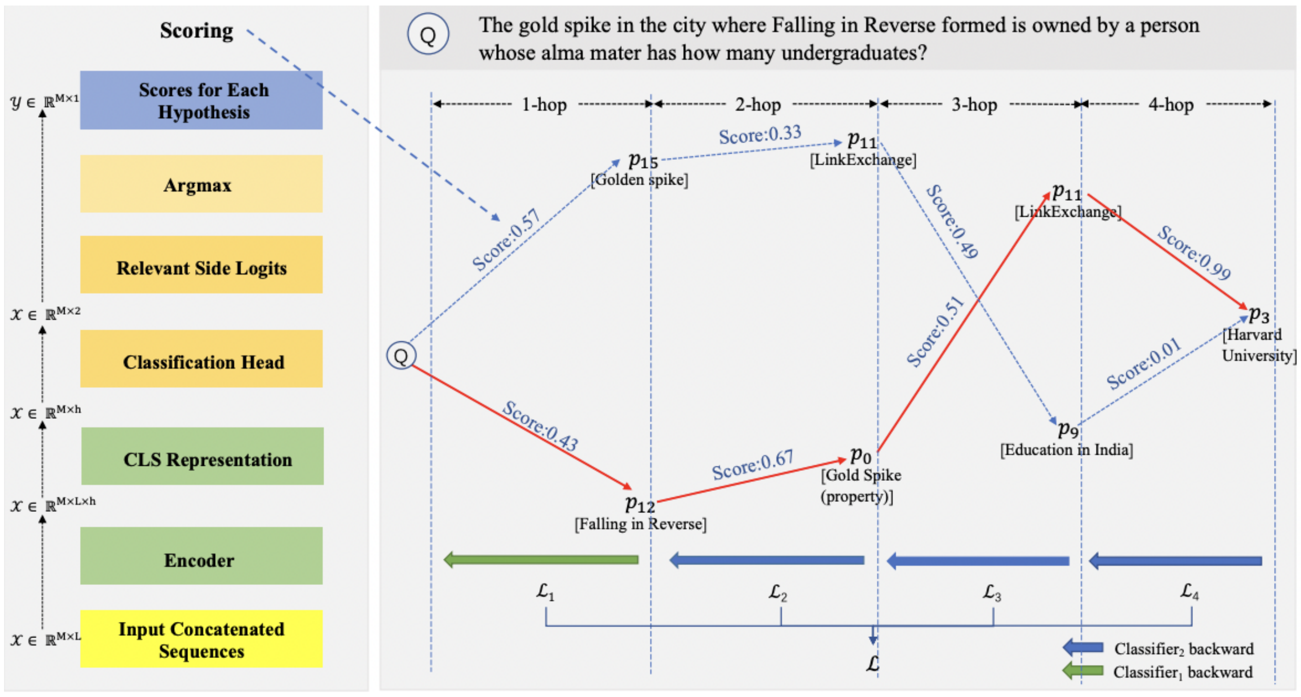

Answer helpfulness :
Is answer useful to address the question?
Retrieval:
Are documents relevant to the question?
Relevant documents
Question
{ }
Search
Context Window
Answer
Reference Answer
Answer accurancy:
Does answer match the ground truth answer?
Hallucination:
Is the answer grounded in documents?
Documents

Figure 14. Langsmith Evaluation



Scoring
Scores for Each Hypothesis
Argmax
Relevant Side Logits
Classification Head
CLS Representation
Encoder
Input Concatenated Sequences
$y \in \mathbb{R}^{M\times1}$
$x \in \mathbb{R}^{M\times2}$
$x \in \mathbb{R}^{M\times h}$
$x \in \mathbb{R}^{M\times L\times h}$
$x \in \mathbb{R}^{M\times L}$
Q The gold spike in the city where Falling in Reverse formed is owned by a person whose alma mater has how many undergraduates?
1-hop
2-hop
3-hop
4-hop
Score:0.57
Score:0.33
Score:0.49
Score:0.99
Score:0.43
Score:0.67
Score:0.51
Score:0.01
p_{15} [Golden spike]
p_{11} [LinkExchange]
p_{11} [LinkExchange]
p_3 [Harvard University]
p_{12} [Falling in Reverse]
p_0 [Gold Spike (property)]
p_9 [Education in India]
$\mathcal{L}_1$
$\mathcal{L}_2$
$\mathcal{L}_3$
$\mathcal{L}_4$
$\mathcal{L}$
Classifier$_2$ backward
Classifier$_1$ backward

Figure 15. Beam Retriever

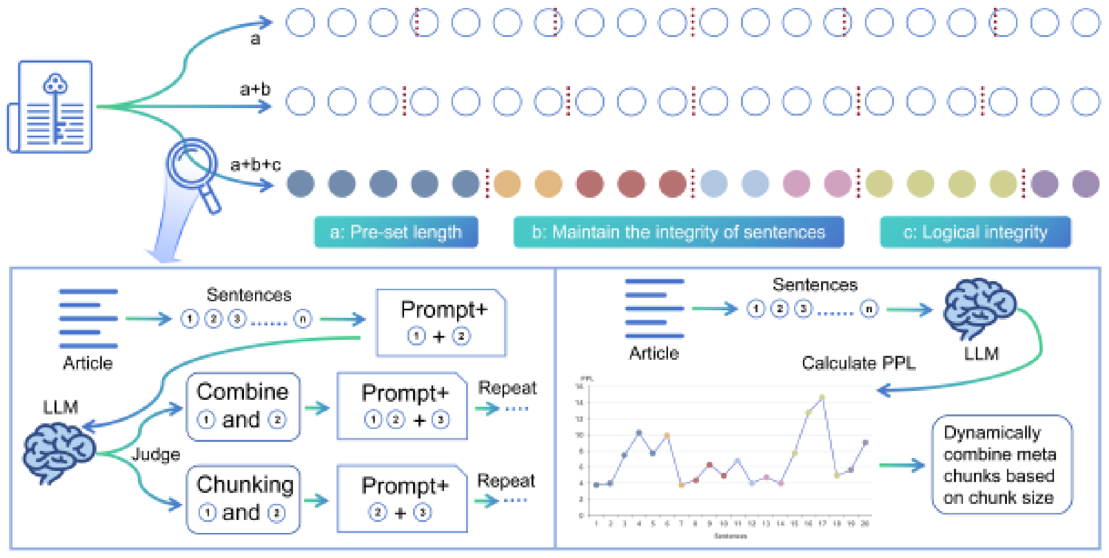


Figure 2: Overview of the entire process of Meta-Chunking. Each circle represents a complete sentence, and the sentence lengths are not consistent. The vertical lines indicate where to segment. The two sides at the bottom of the figure reveal Margin Sampling Chunking and Perplexity Chunking. Circles with the same background color represent a meta-chunk, which is dynamically combined to make the final chunk length meet user needs.

Figure 16. Meta Chunking