RAG Design Doc

Tianming Du Qirui Wu Yanci Zhang

Feedback

Type	Notes
General Comm…	It'll be better if we can incorporate flow charts in this documentation to help visualize the process
General Comm…	We should use one section to talk about how do we handle long-context scenario
General Comm…	Patent projects have higher priority than evaluating financial statement projects.
General Comm…	We may need a separate doc for planning.
General Comm…	It will be better if the doc can use cleaner titles and subtitles to show the hierarchical structure
General Comm…	
General Comm…	
Question	Is this doc supposed to be a design doc or a detailed breakdown of roadmap?
Question	~~balalala~~
Question	Do we have discussion on GraphRAG & existing benchmark?
Action Item	Add Routing, Scheduling, Fusion modules
Action Item	Add more content about GraphRAG
Action Item	Have a discussion with team members and select one or two members to let them own Rag tasks.
Action Item	Write a new doc to describe what dataset we can use and what task we will do.

1. Objectives

- Provide a clear framework for RAG that is easy to break down and implement.

- Ensure the applicability for evaluating on patent data set information retrieval
- Ensure the applicability for evaluating financial statement information retrieval.

2. Background

RAG Overview
Retrieval-Augmented Generation (RAG) enhances the performance of language models by incorporating relevant external information during the text generation process. We will develop and research RAG on domain-specific data, such as financial statements, earnings calls, and finance news.

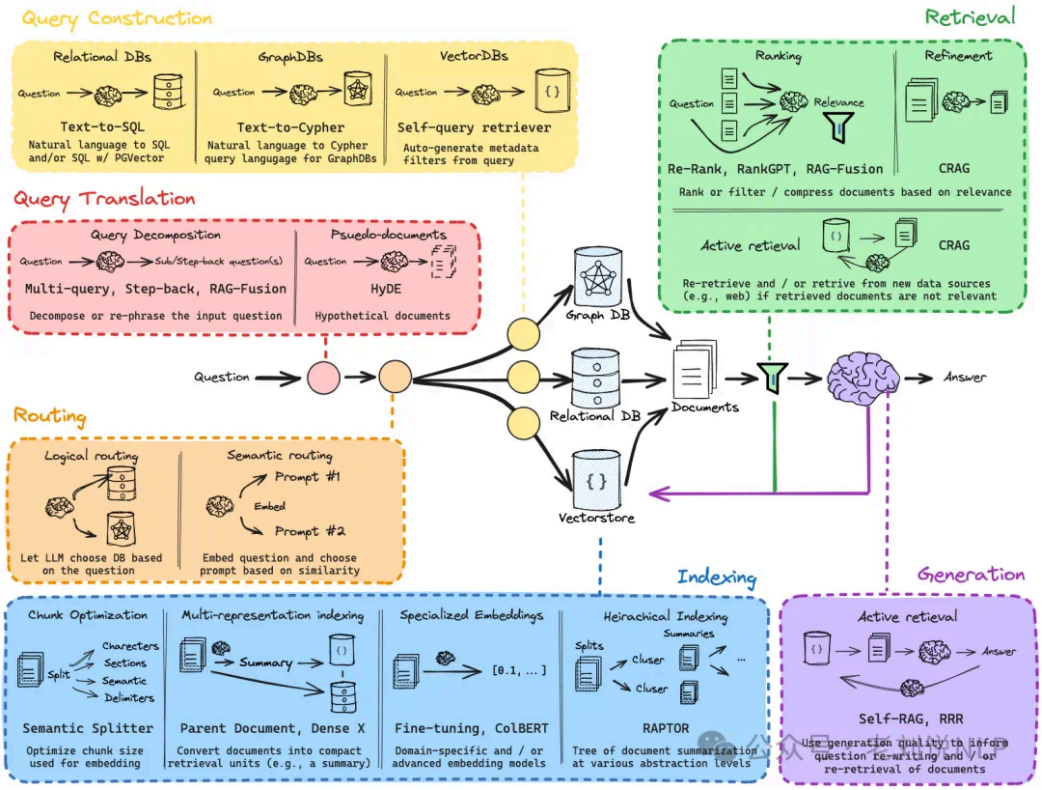
Key Components of RAG in Finance

- **Embedding model:** An embedding model converts text (such as queries, documents, or other textual data) into dense numerical vectors, often called embeddings. These embeddings capture the semantic meaning of the text in a way that makes it possible to compare the similarity between different pieces of text efficiently.
- **Indexing module:** Indexing refers to the process of creating a structured, searchable representation of a large corpus of documents or data. This indexed representation allows the system to quickly retrieve relevant pieces of information in response to a query, which are then used to generate or augment responses.
- **Retrieval module:** After indexing, RAG retrieves relevant documents or information. This step ensures that the generated content is grounded in factual and up-to-date information.
- **Augmented Generation:** After retrieving the relevant documents, the language model generates text by combining the retrieved information with its own internal knowledge. This results in outputs that are both informed by external data and enriched by the model's generative capabilities.

Importance of RAG in Financial Analysis
RAG plays a crucial role in financial analysis by:

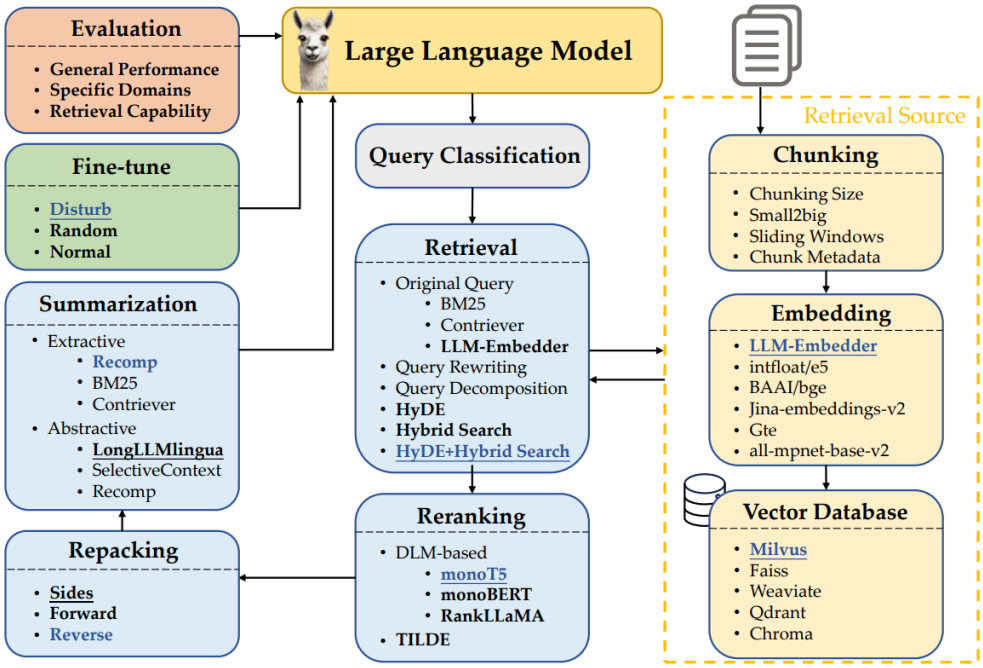
- **Enhancing Decision-Making:** By integrating real-time, context-specific data, RAG helps financial analysts and decision-makers generate insights that are more aligned with current market conditions and company performance.
- **Improving Accuracy:** The retrieval step ensures that the generated outputs are accurate and based on the latest available information, reducing the risk of outdated or irrelevant content.
- **Providing Contextual Understanding:** RAG allows for a deeper understanding of complex financial scenarios by incorporating information from multiple sources, offering a more comprehensive view of the subject matter.

3. Methods



3.1) General RAG pipeline

Assume we already had an eval dataset.

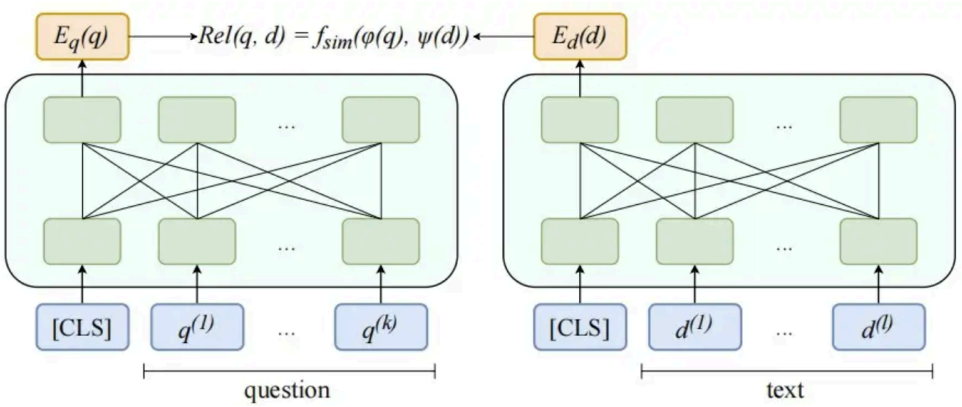


- Run a demo RAG system
  - Step 1: Baseline: Query translation -> Query Embedding -> Indexing -> Retrieval -> Generation
    - Run the whole pipeline.
      - Code: https://github.com/langchain-ai/rag-from-scratch/tree/main
      - Video: https://www.youtube.com/watch?v=wd7TZ4w1mSw&list=PLfaIDFEXuae2LXbO1_PKyVJiQ23ZztA0x&index=1
    - Prepare a mini evaluation dataset.
      - Collect some dataset of about 20 case
      - Use ChatGPT to generate answers
      - Eva
  - Step 2: More modules like repacking or summarization can be added.

- Embedding model

  Embedding model is usually a dual-encoder model to extract information from query and text.

  - Run Hugging Face bce embedding model locally. https://huggingface.co/maidalun1020/bce-embedding-base_v1
  - Deploy a Hugging Face embedding model.



Low *High*

Embedding Models	WithoutReranker [*hit_rate/mrr*]	bge-reranker-large [*hit_rate/mrr*]	bge-reranker-v2-m3 [*hit_rate/mrr*]	CohereRerankMultilingualV3 [*hit_rate/mrr*]	***bce-reranker-base_v1*** [*hit_rate/mrr*]
OpenAI-ada-2	81.04/57.35	88.89/69.64	88.39/70.10	89.16/72.49	90.71/75.46
OpenAI-embed-3-small	83.01/58.10	89.20/69.86	89.20/70.22	90.02/72.78	90.91/75.49
OpenAI-embed-3-large	83.78/59.65	89.59/70.20	90.05/70.96	90.60/73.34	91.37/75.82
bge-large-en-v1.5	52.67/34.69	64.71/52.05	63.97/51.89	64.51/54.12	65.36/55.50
bge-large-zh-v1.5	69.81/47.38	80.11/63.95	79.33/64.64	79.95/66.23	81.19/68.50
bge-m3-large	84.67/61.25	89.94/70.17	89.40/70.57	90.13/72.93	91.72/76.14
llm-embedder	50.85/33.26	63.54/51.32	68.11/54.89	68.81/56.78	64.47/54.98
CohereV3-en	53.10/35.39	66.29/53.31	65.02/52.96	66.02/55.07	66.91/56.93
CohereV3-multilingual	79.80/57.22	86.76/68.56	86.15/68.72	87.11/71.04	88.35/73.73
JinaAI-v2-Base-zh	71.63/49.62	81.77/64.89	80.96/65.16	81.39/67.03	83.13/69.64
gte-large-en	53.17/34.71	65.02/52.04	64.05/51.86	64.55/53.68	65.67/55.87
gte-large-zh	59.48/39.38	71.56/58.27	70.67/58.89	71.01/60.22	72.37/62.71
e5-large-v2-en	61.03/40.67	71.52/56.61	70.90/57.01	71.28/58.62	72.37/60.91
e5-large-multilingual	79.14/55.54	87.35/68.50	86.92/68.77	87.73/71.15	88.97/73.81
bce-embedding-base_v1	85.91/62.36	91.80/71.13	91.02/71.43	91.87/73.34	*93.46/77.02* *96.36/78.93*(★)

(★): Hybrid search for top10 recall of bce-embedding and top10 recall of bm25, which are reranked together by bce-reranker subsequently.

- Simply evaluate the embedding model.
- (Optional) Using our in-house data to train embedding model for better performance

An **embedding model** converts text (such as queries, documents, or other textual data) into dense numerical vectors, often called **embeddings**. These embeddings capture the semantic meaning of the text in a way that makes it possible to compare the similarity between different pieces of text efficiently.

Some wide-use embedding model, team members can test more and compare results of them:

Openai: https://platform.openai.com/docs/guides/embeddings
huggingface：https://huggingface.co/BAAI/bge-large-zh-v1.5

We can use the above huggingface embedding model.

Embedding model leaderboard:
https://huggingface.co/spaces/mteb/leaderboard

(Optional) Team members can train new embedding models using our in-house data for better performance.

Some papers:

C-Pack: Packed Resources For General Chinese Embeddings
https://arxiv.org/pdf/2309.07597
SimCSE: Simple Contrastive Learning of Sentence Embeddings
https://arxiv.org/pdf/2104.0882

kTrans: Knowledge-Aware Transformer for Binary Code Embedding
https://arxiv.org/pdf/2308.12659

- Indexing

  Indexing refers to the process of creating a structured, searchable representation of a large corpus of documents or data. This indexed representation allows the system to quickly retrieve relevant pieces of information in response to a query, which are then used to generate or augment responses.

    - Chunk creation
        - fixed chunk size
        - Other options:
            - semantic segmentation
            - sliding window
    - Build a Faiss database. Faiss is highly optimized for speed, particularly on GPU. It's well-suited for large-scale similarity search and can handle billions of vectors efficiently when combined with GPUs.
        - Other option
            - Milvus
            - RAPTOR
            - HNSW
    - Deploy a Faiss indexing service.
        - Index Creation
        - Index Tuning
        - Deployment

- Recall the passages according to the recall embedding.

- Some new tech can be explored:
  - Multi-representation Indexing:

(Utilize multiple representations or embeddings of the same document or piece of data. Allows the system to capture different aspects or views of the content)

ME-BERT: https://arxiv.org/pdf/2005.00181
Poly-Encoder: https://arxiv.org/pdf/1905.01969

- Contextual Indexing:
(Refers to the process of incorporating additional context into the indexing and retrieval stages of a RAG system.)

ULMFiT: https://arxiv.org/abs/1801.06146
ColBERT: https://arxiv.org/pdf/2004.12832

- Pre-retrieval
  - Baseline
    - Multi Query: multiple queries are generated or issued simultaneously to improve the relevance and diversity of the retrieved results.
      - Implementation of Multi-query Translation Mechanism
      - Performance Optimization and Testing
      - Comparative Analysis and Validation

    -
      - PROS:
        - Diversity: Capable of handling multiple queries simultaneously, which enhances the breadth and coverage of information retrieval.
        - Robustness: Better adapts and extracts relevant information from various sources.
        - Feasibility: Easy to implement
      - CONS:

        - Computational Cost: Processing multiple queries at once increases computational burden.
        - Complexity Management: Managing the results from multiple queries can add complexity to the system.

- More methods to be explored
  - RAG-Fusion
    - PROS:
      - Context Awareness: Better utilizes the context retrieved to generate more accurate answers.
    - CONS:
      - Complexity: Requires a highly optimized model architecture to effectively integrate retrieval and generation steps.
      - Feasibility: Hard to implement
      - Resource Consumption: May require more computational resources and storage.
  - Decomposition
    - PROS:
      - Modularity: By breaking down complex queries into simpler sub-questions, each part can be dealt with more focus and detail.
      - Explainability: The decomposition steps can provide more intermediate results, aiding in understanding the model's decision-making process.
    - CONS:
      - Loss of Information: The decomposition process might overlook the contextual relationships across sub-problems.
      - Efficiency: The decomposition and reassembly steps could increase processing time.
  - Step Back
    - PROS:
      - Iterative Optimization: Allows the model to reconsider its decisions after generating an answer, potentially improving the quality of the response.
      - Error Correction: Can correct mistakes or inaccuracies in preliminary answers.
    - CONS:
      - Delay: Increases processing time, which may not be suitable for applications requiring quick responses.
      - Resource Consumption: Requires additional computational steps to reassess and modify answers.
  - HyDE Advantages:
    - PROS:

        - Flexibility: Capable of dynamically adjusting based on real-time data and feedback.
        - Adaptability: Better adapts to complex and changing query environments.
    - CONS:
        - Implementation Difficulty: May require complex algorithms and models to support dynamic adjustments.
        - Predictability Issues: Dynamic adjustments can make model behavior unpredictable.

- Retrieval

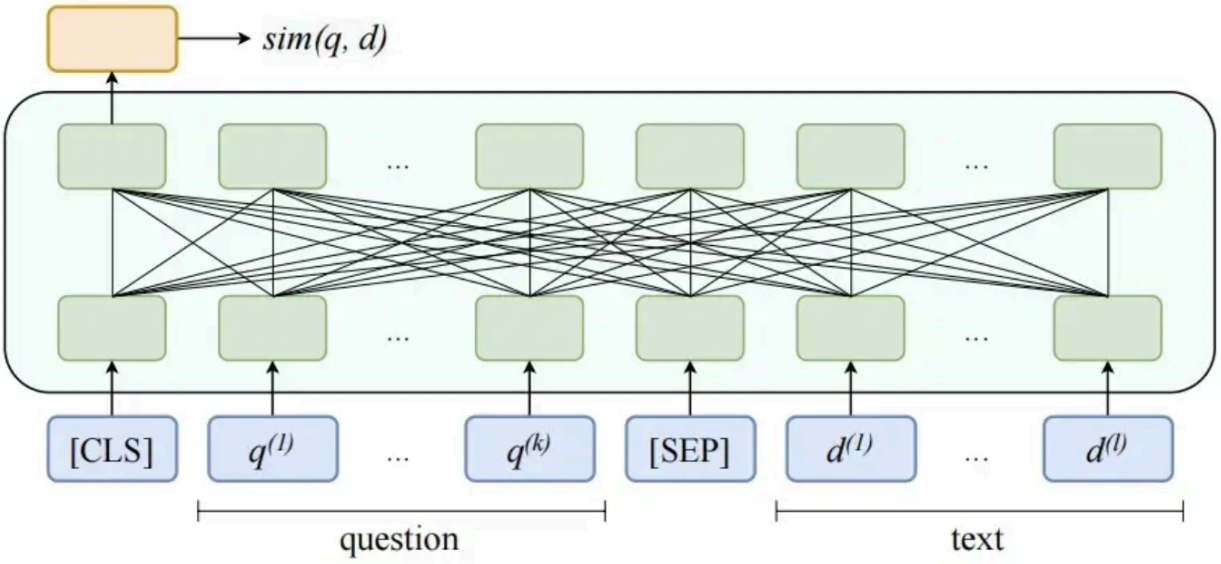
  Retrieval refers to the process of retrieving relevant information or documents from a large corpus or database to assist in generating responses.

    - Baseline:
        - Local deployment for Bce-reranker model, a cross-encoder architecture model, a sota reranker model. https://huggingface.co/maidalun1020/bce-reranker-base_v1
        - Deployment for Bce-reranker model
        - Evaluation for Bce-reranker model
        - Integrate this model in the RAG system to rerank the recall passages in the indexing processing

Low → *High*

Embedding Models	WithoutReranker [*hit_rate/mrr*]	bge-reranker-large [*hit_rate/mrr*]	bge-reranker-v2-m3 [*hit_rate/mrr*]	CohereRerankMultilingualV3 [*hit_rate/mrr*]	***bce-reranker-base_v1*** [*hit_rate/mrr*]
OpenAI-ada-2	81.04/57.35	88.89/69.64	88.39/70.10	89.16/72.49	**90.71/75.46**
OpenAI-embed-3-small	83.01/58.10	89.20/69.86	89.20/70.22	90.02/72.78	**90.91/75.49**
OpenAI-embed-3-large	83.78/59.65	89.59/70.20	90.05/70.96	90.60/73.34	**91.37/75.82**
bge-large-en-v1.5	52.67/34.69	64.71/52.05	63.97/51.89	64.51/54.12	**65.36/55.50**
bge-large-zh-v1.5	69.81/47.38	80.11/63.95	79.33/64.64	79.95/66.23	**81.19/68.50**
bge-m3-large	84.67/61.25	89.94/70.17	89.40/70.57	90.13/72.93	**91.72/76.14**
llm-embedder	50.85/33.26	63.54/51.32	68.11/54.89	68.81/56.78	**64.47/54.98**
CohereV3-en	53.10/35.39	66.29/53.31	65.02/52.96	66.02/55.07	**66.91/56.93**
CohereV3-multilingual	79.80/57.22	86.76/68.56	86.15/68.72	87.11/71.04	**88.35/73.73**
JinaAI-v2-Base-zh	71.63/49.62	81.77/64.89	80.96/65.16	81.39/67.03	**83.13/69.64**
gte-large-en	53.17/34.71	65.02/52.04	64.05/51.86	64.55/53.68	**65.67/55.87**
gte-large-zh	59.48/39.38	71.56/58.27	70.67/58.89	71.01/60.22	**72.37/62.71**
e5-large-v2-en	61.03/40.67	71.52/56.61	70.90/57.01	71.28/58.62	**72.37/60.91**
e5-large-multilingual	79.14/55.54	87.35/68.50	86.92/68.77	87.73/71.15	**88.97/73.81**
bce-embedding-base_v1	**85.91/62.36**	**91.80/71.13**	**91.02/71.43**	**91.87/73.34**	***93.46/77.02 96.36/78.93*(★)**

(★): Hybrid search for top10 recall of bce-embedding and top10 recall of bm25, which are reranked together by bce-reranker subsequently.



- Reranking : Train a bert-base re-ranking model to assign a score to document with query, more methods can be explored
  - Re-ranking Improve Retrieval Augmented Generation (RAG) with Re-ranking

- Compression:
  - Using LLMlingua for compression:
    - Initialize LLMlingua:
      - Load pre-trained LLMlingua model.
    - Prepare Input:
      - Tokenize retrieved document chunks.
    - Apply Compression:
      - For each chunk: compressed_chunk = LLMlingua.compress(chunk, target_length=desired_length)
    - Post-process:
      - Detokenize compressed chunks.
      - Ensure compressed chunks maintain coherence and key information.
    - Update Retrieved Set:
      - Replace original chunks with compressed versions in the retrieved set.

- Selection:
  - Direct Removal of Irrelevant Chunks:
    - Define Relevance Threshold:
      - Set a minimum relevance score (e.g., 0.5 on a 0-1 scale).

- Calculate Relevance Scores
    - Use a simple method like TF-IDF or BM25 to score chunks against the query.
- Filter Chunks:
    - Keep chunks where relevance_score(chunk, query) > threshold
    - Remove other chunks from the retrieved set.

- LLM-based Relevance Evaluation:
    - Prepare Prompt Template:
        - Create a template asking LLM to evaluate relevance, e.g.:"On a scale of 0-10, how relevant is the following text to the query: '{query}'?\n\nText: {chunk}\n\nRelevance score:"
    - Evaluate Chunks:
        - For each chunk: a. Fill in the prompt template with the query and chunk. b. Send to LLM and get the relevance score.
        - Filter Chunks:
            - Define a threshold (e.g., 7 out of 10).
            - Keep chunks with scores above the threshold.
            - Remove other chunks from the retrieved set.
        - Optional: Adaptive Thresholding
            - If too many/few chunks are kept, adjust the threshold dynamically.
        - Update Retrieved Set:
            - Return the filtered set of relevant chunks.

- **Implementation Notes:**
    - **These steps can be combined in a pipeline, e.g., first compress, then rerank, then select.**
    - **The order of operations may affect the final result, so experimentation is crucial.**
    - **Efficient implementation (e.g., batch processing for LLM calls) is important for real-time applications.**
    - **Always monitor and evaluate the impact of post-retrieval processing on final generation quality.**

- Generation

  The language model generates text by combining the retrieved information with its own internal knowledge.

    - Baseline:
        - OpenAI LLM API
        - In-house deployed LLM API.
    - More methods can be explored
        - Self-RAG
    - Generator Fine-tuning（Optional）:
        1. Instruction Fine-tuning:
            1) Data Preparation:
                a) Collect domain-specific Q&A pairs or instruction-response pairs.
                b) Convert data into model-acceptable format, typically JSON: {"instruction": "...", "input": "...", "output": "..."}.
            2) Data Preprocessing:
                a) Clean, deduplicate, and normalize the data.
                b) Encode data using a specific tokenizer.
            3) Set Fine-tuning Parameters:
                a) Choose appropriate learning rate, batch size, number of epochs, etc.
                b) Set optimizer (e.g., AdamW) and learning rate scheduler.
            4) Fine-tuning Process:
                a) Use Hugging Face's Trainer or custom training loop for fine-tuning.
                b) Evaluate model performance after each epoch, save best checkpoints.
            5) Evaluation and Deployment:
                a) Evaluate fine-tuned model performance on test set.
                b) Deploy best model to production environment.
        2. Distillation:
            1) Prepare Teacher Model:
                a) Select a powerful model (e.g., GPT-4) as the teacher model.
            2) Generate Training Data:
                a) Use teacher model to generate high-quality Q&A or instruction-response pairs.
                b) Ensure generated data covers various aspects of the target task.
            3) Prepare Student Model:

        a) Select a smaller model as the student model (e.g., open-source LLM).
    4) Set Distillation Objectives:
        a) Define loss function, typically including: a. Distillation loss: KL divergence between student and teacher outputs. b. Task-specific loss: Cross-entropy between student output and true labels.
    5) Training Process:
        a) Train student model using generated dataset.
        b) Calculate both distillation and task losses in each batch.
        c) Update student model parameters to minimize total loss.
    6) Evaluation and Fine-tuning:
        a) Evaluate student model performance on validation set.
        b) Perform further fine-tuning or adjustments as needed.
3. Reinforcement Learning: Steps **(Optional)**:
    1) Prepare Environment:
        a) Define state space (usually current dialogue history).
        b) Define action space (usually possible model outputs).
        c) Define reward function (based on human feedback or LLM scoring).
    2) Implement DPO or PPO Algorithm:.
    3) Collect Feedback:
        a) For human feedback, design interface for humans to evaluate model outputs.
        b) For LLM feedback, use a powerful LLM to assess output quality.
    4) Training Loop:
        a) Generate dialogue samples.
        b) Collect feedback and calculate rewards.
        c) Update model parameters using PPO algorithm.
        d) Periodically evaluate model performance.
    5) Fine-tuning and Deployment:
        a) Make necessary adjustments based on evaluation results.
        b) Deploy trained model to production environment.

- Orchestration
    - Routing:
        - Metadata Routing:
            - Keyword Extraction:
                - Use NLP techniques (e.g., TF-IDF, RAKE) to extract keywords from the query.
            - Metadata Matching:
                - Create a mapping of keywords to specific RAG flows.
                - For each extracted keyword, check if it matches any predefined routing rules.
            - Scoring:
                - Calculate a score for each potential flow based on keyword matches.
            - Route Selection:
                - Choose the flow with the highest score.
        - Semantic Routing:
            - Encode Query:
                - Use a pre-trained sentence encoder (e.g., SBERT) to convert the query into a vector.
            - Compute Similarities:
                - Calculate cosine similarity between the query vector and pre-defined flow vectors.
            - Route Selection:
                - Choose the flow with the highest similarity score.
        - Hybrid Routing:
            - Perform both Metadata and Semantic Routing.
            - Combine Scores:
                - Use a weighted sum of metadata and semantic scores.
                - score = α * metadata_score + (1 - α) * semantic_score
            - Route Selection:
                - Choose the flow with the highest combined score.

    - Scheduling:
        - Rule-based Scheduling:
            - Define Rule Set:
                - Create a set of if-then rules for different scenarios.
            - State Tracking:
                - Maintain a state object tracking current progress in the RAG process.
            - Rule Evaluation:
                - At each decision point, evaluate all applicable rules.
            - Action Selection:
                - Execute the action specified by the highest priority matching rule.

        - LLM-based Scheduling:
            - Prompt Design
                - Create a prompt template describing the current state and possible actions.
            - LLM Querying:
                - Send the filled prompt to the LLM.
            - Response Parsing:
                - Parse the LLM's response to extract the chosen action.
            - Action Execution:
                - Execute the action specified by the LLM.
                -
        - Knowledge-guided Scheduling:
            - Knowledge Graph Query:
                - Extract key entities from the current query/state.
                - Query the knowledge graph to get relevant information.
            - Reasoning Chain Construction:
                - Use the graph structure to construct a chain of reasoning steps.
            - Step Mapping:
                - Map each reasoning step to a corresponding RAG operation.
            - Sequential Execution:
                - Execute RAG operations in the order specified by the reasoning chain.

    - Fusion:
        - LLM Fusion:
            - Prepare Fusion Prompt:
                - Create a template summarizing results from different branches.
            - LLM Analysis:
                - Send the filled prompt to the LLM, asking it to analyze and integrate the information.
            - Response Generation:
                - Use the LLM's integrated response as the final output.

        - Weighted Ensemble:
            - Token-level Scoring:
                - For each generated token from different branches, assign a confidence score.
            - Weighted Probability Calculation:
                - Calculate the weighted probability for each possible next token
            - Token Selection:

        - Choose the token with the highest weighted probability.
    - Iterative Generation:
        - Repeat steps 2-3 until the full response is generated.
- Reciprocal Rank Fusion (RRF):
    - Rank Normalization:
        - For each retrieval result list, normalize ranks to a common scale.
    - RRF Score Calculation
    - Merged Ranking:
        - Sort items based on their RRF scores.
    - Final Selection:
        - Use the top-ranked items from the merged list.

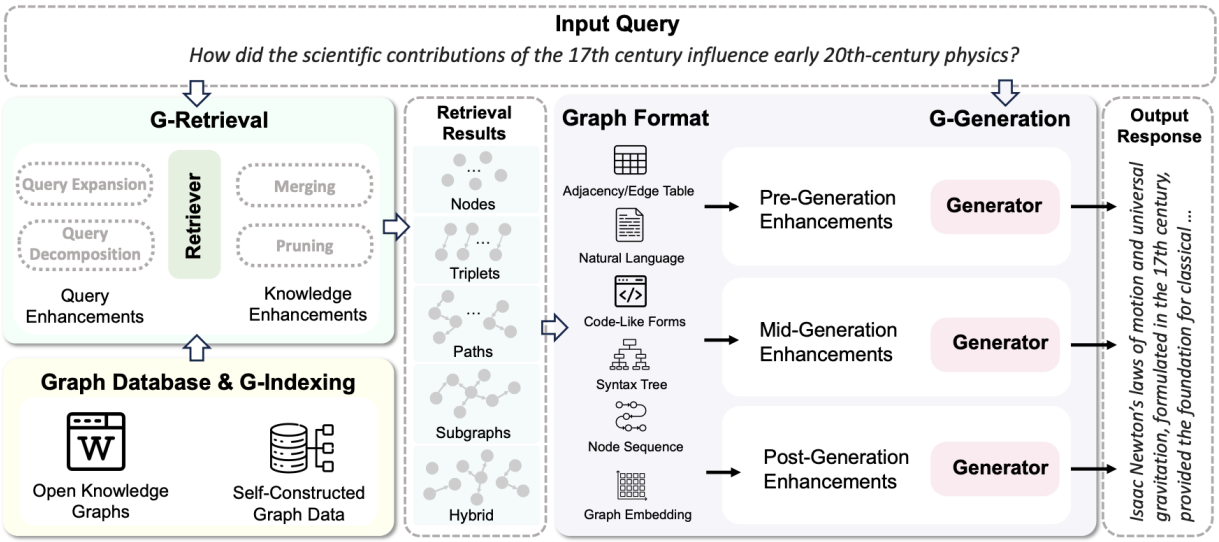
- Agentic RAG
    1. Hierarchical Multi-Agent Architecture:
        - Implement a top-level master agent analyzes user requests and delegates tasks to specialized sub-agents. LLaMA-8B can be used here.
        - Sub-agents are dedicated to specific time series tasks such as forecasting, anomaly detection, imputation, etc.
    2. Sub-Agent Implementation:
        - Utilizes small pre-trained language models (SLMs), LLaMA-8B.
        - Applies instruction tuning to these SLMs to adapt them to specific time series tasks.
    3. Dynamic Prompting Mechanism:
        - Maintains a prompt pool for each sub-agent, storing knowledge of historical patterns and trends.
        - Uses similarity matching to retrieve relevant prompts from the pool.
        - Combines retrieved prompts with input data to enhance the model's predictive capabilities.
    4. Data Processing:
        - Employs sliding window techniques to process time series data.
        - Applies data standardization.
    5. Training Process:
        - Uses Parameter-Efficient Fine-Tuning (PEFT) techniques like QLoRA for training.
        - Adopts the AdamW optimizer and cosine learning rate scheduling.
    6. Inference:
        - The master agent receives user requests and selects appropriate sub-agents.
        - Sub-agents retrieve relevant information from prompt pools and generate outputs.
        - The master agent integrates sub-agent outputs to produce the final answer.
    7. Evaluation:

- Conducts evaluations on multiple benchmark datasets, covering tasks such as forecasting, anomaly detection, imputation, and classification.
- Utilizes various evaluation metrics including MAE, RMSE, MAPE, precision, recall, F1 score, etc.

3.2) RAG pipeline for special data (copy from Algo/ML Roadmap)

- RAG with table data
  - Paper list: https://github.com/SpursGoZmy/Awesome-Tabular-LLMs
  - Table extraction and database creation
    - Table extraction with HTML
    - Key-value store based on table

- GraphRAG
  - Research for open-source graph dataset.
  - Run a graph pipeline.
    1. https://github.com/Cinnamon/kotaemon
    2. https://github.com/infiniflow/ragflow
  - Evaluate this GraphRAG pipeline using open-source dataset and our inhouse dataset.



- GraphRAG Component Details
  - Graph-Based Indexing
    - Data Source: Our previous graph data
    - Indexing Method:
      - Vector indexing, use Text2Vector and OpenAI Embedding
      - Graph indexing, employed extractors such as triple extraction and keyword extraction
    - Output: Indexed graph database
  - Graph-Guided Retrieval
    - Retriever Type: GNN-based retriever
      - use models such as Graph Convolutional Network (GCN) or Graph Attention Network (GAT) to extract features and identify important nodes and edges.
    - Retrieval Paradigm: Multi-stage retrieval
      - Preliminary Screening Phase: Quickly and roughly filter out potentially relevant subgraphs from a large dataset.
      - Refinement Phase: Further analyze and evaluate the results initially filtered, using more complex graph algorithms or models to refine the outcomes.
      - Re-ranking Phase: Sort the final candidates based on relevance and other metrics to ultimately determine the most relevant graph elements.
    - Retrieval Granularity: Subgraphs
    - Output: Retrieved relevant graph elements
  - Graph-Enhanced Generation
    - Generator: open source API
    - Graph Format: Natural language descriptions
    - Enhancement Method: Pre-generation enhancement
    - Output: Generated response
  - Workflow
    - User inputs a query
    - Retriever extracts relevant subgraphs from the indexed database
    - Retrieved information is formatted for the generator
    - Generator produces the final response
  - Evaluation Metrics
    - Relevance: NDCG@K
    - Answer Correctness: F1 score
    - Response Quality: Human evaluation
- RAG with knowledge graph
  - Use KG to complement and validate context
    - In query translation part, use entity substitution
    - In query construction part, use context from KG
    - In retrieval and generation part, use KG to correct and validate result

        - Extract entity from query and search subgraph from knowledge graph to be used as context
            - Extract entity from query
            - Search KG based on entity and extract subgraph
        - Use KG to improve relevance between chunks
            - Store the hierarchy between chunks when indexing
            - Use entity as embedding to filter result during reranking
            - In long-context scenario, use KG to do summarization and different levels of query
- Risk identification in financial statements/earning calls/finance news
    - This could be a use case of information retrieval result to identify risk signal for a company

4 Experimental Design

A general evaluation pipeline that can be adapted for various types of data, such as patents, financial statements, earnings calls, and financial news.

1. Data preparation

2. Embedding Model Evaluation

- **Objective:** Compare different embedding models to determine which provides the most semantically rich and useful embeddings for the downstream tasks.
- **Models to Compare:**
    - OpenAI's Embedding Model
    - BAAI/bge-large-zh-v1.5 (Hugging Face)
    - In-house trained models (if applicable)
- **Metrics:**
    - **Semantic similarity:** Use cosine similarity to measure how closely related embeddings are for similar queries/documents.
    - **Retrieval accuracy:** Measure the precision of top-k retrieved documents using embeddings for various test queries.
    - **Computational efficiency:** Time taken to generate embeddings and memory footprint.
- **Experimental Setup:**
    - Use a fixed set of test queries and corresponding ground truth documents.
    - Calculate the precision@k for retrieval using embeddings from each model.
    - Compare the time taken to generate embeddings and their memory usage.

3. Indexing Evaluation

- **Objective:** Determine the effectiveness and efficiency of different indexing techniques.
- **Techniques to Compare:**
    - Faiss, Milvus, RAPTOR, HNSW (for baseline vector databases)
    - Multi-representation Indexing (e.g., ME-BERT, Poly-Encoder)
    - Contextual Indexing (e.g., ULMFiT, ColBERT)
- **Metrics:**
    - **Recall@k:** The proportion of relevant documents retrieved among the top-k results.
    - **Latency:** Time taken to perform the retrieval operation.
    - **Indexing overhead:** Time and resources required to build the index.(optional)
- **Experimental Setup:**
    - Index a standardized corpus with different indexing methods.
    - Execute a set of test queries and evaluate recall@k and retrieval latency.
    - Measure the time taken to build each index and the storage requirements.

4. Retrieval Evaluation

- **Objective:** Assess the retrieval mechanisms, especially focusing on re-ranking, query rewriting, and query decomposition.
- **Techniques to Explore:**
    - Re-ranking strategies
    - Query rewriting, query decomposition
    - Hybrid methods (e.g., HyDE)
- **Metrics:**
    - **Precision@k:** The proportion of retrieved documents that are relevant.
    - **Re-ranking effectiveness:** Improvement in relevance after applying re-ranking strategies.
    - **Query rewriting/decomposition success:** Improvement in retrieval performance after applying these techniques.
- **Experimental Setup:**
    - Perform retrieval on a set of test queries using baseline and advanced retrieval techniques.
    - Evaluate the precision@k and assess improvements brought by re-ranking, query rewriting, and decomposition.
    - Conduct ablation studies to isolate the impact of each retrieval enhancement.

5. Generation Evaluation

- **Objective:** Evaluate the quality and relevance of generated responses.
- **Models to Compare:**
    - LLM APIs (e.g., OpenAI's GPT, other available models)
    - Open-source LLMs (e.g., GPT-NeoX, LLaMA)

- **Metrics:**
    - **Relevance:** How well the generated response aligns with the retrieved information and the original query.
    - **Fluency:** The linguistic quality of the generated text.
    - **Factual accuracy:** The correctness of the information provided in the response.
    - **Response diversity:** The variety of responses generated for the same input.
    - **Manual evaluation**
- **Experimental Setup:**
    - Generate responses for a standardized set of queries using different generation models.
    - Conduct human evaluations to assess relevance, fluency, and factual accuracy.
    - Use automated metrics (e.g., BLEU, ROUGE) to assess text quality and relevance.
    - Perform A/B testing to compare different generation strategies.

6. End-to-End System Evaluation

- **Objective:** Evaluate the performance of the entire RAG pipeline.
- **Metrics:**
    - **Overall response quality:** Combination of relevance, fluency, and factual accuracy.
    - **System latency:** Time taken to generate a complete response from input to final output.
    - **Scalability:** Ability to handle large volumes of queries.
    - **Manual evaluation: GSB**
- **Experimental Setup:**
    - Deploy the full RAG pipeline with various configurations (e.g., different combinations of embedding models, indexing methods, retrieval techniques, and generation models).
    - Collect a large set of queries to simulate real-world usage.
    - Evaluate using both automated metrics and user feedback.