LAMINI

Memory RAG

Simple High-Accuracy LLMs using Embed-Time Compute

Memory RAG: Simple High-Accuracy LLMs Using Embed-Time Compute

Abstract

This paper introduces Memory RAG, a novel approach to Retrieval-Augmented Generation (RAG) that leverages embed-time compute to create more intelligent, validated data representations. Unlike traditional RAG implementations that struggle with balancing information coverage and context window size, Memory RAG transforms raw data into rich representations that capture both meaning and relationships more effectively.

By investing computational power during the embedding phase, Memory RAG creates higher-quality data representations from raw data that lead to more precise information retrieval, reducing retrieval misses and model hallucinations, while requiring smaller context windows. Our experimental results show dramatic improvements in accuracy across diverse use cases, including tests on real Fortune 500 company data, with Memory RAG achieving 91-95% accuracy compared to 20-59% for traditional RAG approaches.

This approach provides a simplified path to achieving higher accuracy in LLM applications without the complexity of advanced RAG techniques, while also offering a natural progression path to Memory Tuning for organizations ready to move towards fine-tuning solutions.

Figure 1: Memory RAG Results

Memory RAG uses embed-time compute to create higher-quality data representations.

Financial Document Analysis

91% accuracy

with Memory RAG compared to 59% with RAG on GPT4

Text-to-SQL Query Understanding

95% accuracy

with Memory RAG compared to 20% with RAG on GPT4

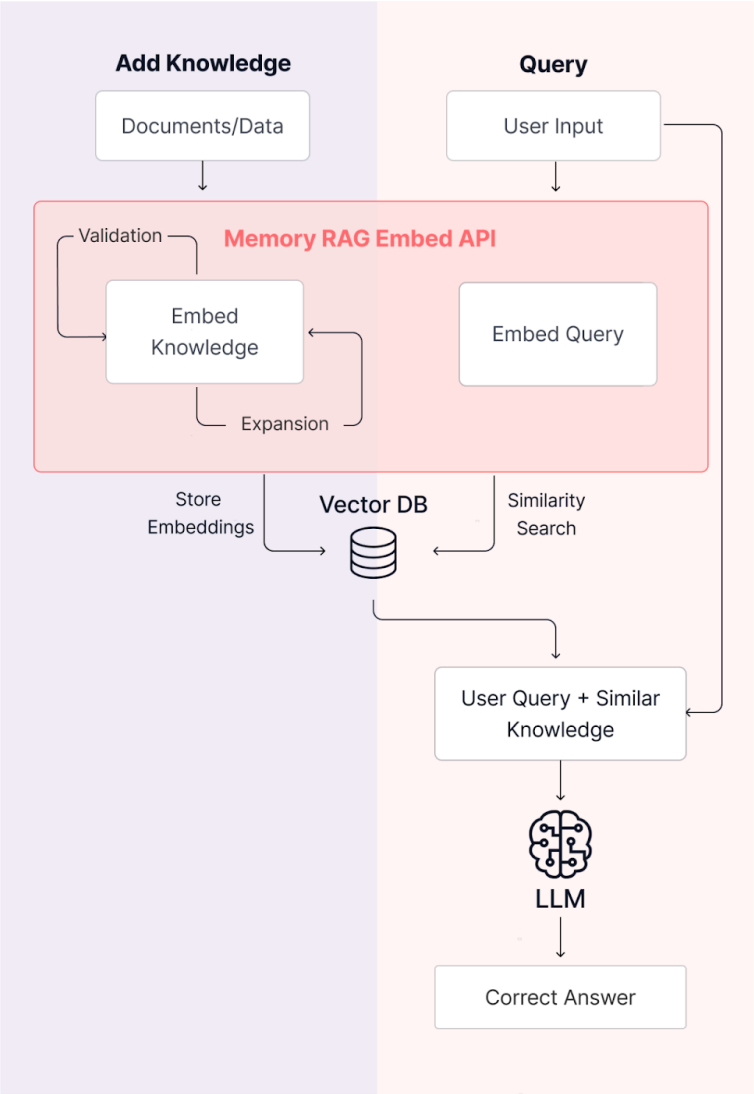


Table of content

1. Introduction

Large Language Models (LLMs) are becoming integral to modern enterprise applications, but achieving high accuracy while maintaining simplicity remains a significant challenge. Basic RAG systems face a fundamental limitation: they must balance between comprehensive data coverage and context window constraints. To ensure they catch all relevant information, they often need to include large amounts of content in the context window, which degrades accuracy as the LLM gets overwhelmed with information.

Traditional approaches present organizations with a difficult choice: either keep data as-is and burden the LLM with figuring out complex relationships, or implement increasingly complex architectures, leading to diminishing returns on engineering effort.

Our experimental results highlight this challenge: in a comprehensive evaluation of factual reasoning on financial documents using the latest GPT-4 endpoint with OpenAI's built-in RAG, accuracy reached only 59% despite using state-of-the-art models. Even more striking, in complex database querying scenarios for a Fortune 500 enterprise, traditional RAG approaches on query logs with a prompt describing the schema in detail achieved only 20% accuracy using GPT-4.

Figure 2: GPT-4 RAG vs. Memory RAG Accuracy

GPT-4 RAG (Accuracy)		Lamini Memory RAG (Accuracy)	
59%	20%	91%	95%
Factual Reasoning	Text-to-SQL	Factual Reasoning	Text-to-SQL

Meanwhile, fine-tuning offers superior accuracy but requires significant technical expertise to control it effectively and computational resources that most teams will slowly adopt in the coming years.

How should teams tackle these problems in the meantime?

Memory RAG takes a fundamentally different approach by focusing computational power where it matters most: the embedding generation phase. Instead of just converting text into basic numerical representations (embeddings), Memory RAG uses embed-time compute to process and enhance the data during embedding creation. This enhanced processing:

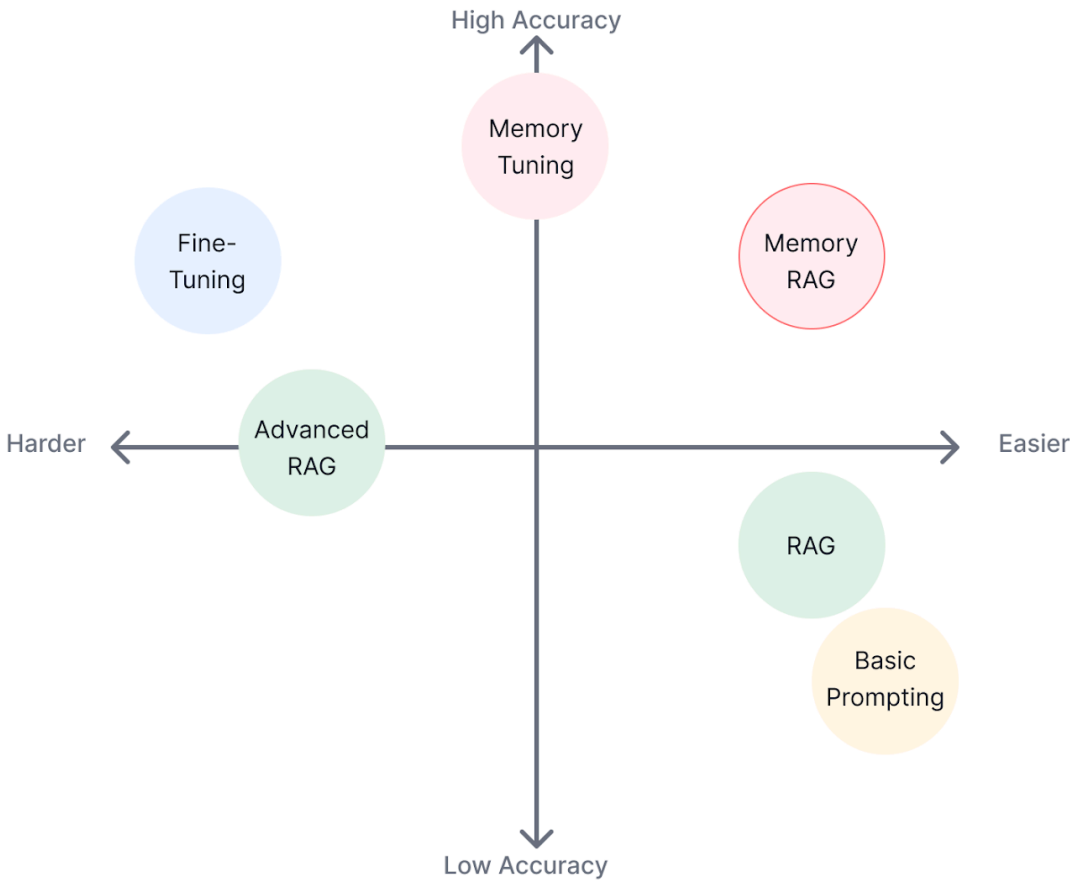
1. Identifies and preserves important relationships between different pieces of information
2. Validates data accuracy and reliability through proprietary checks, including specialized Memory Tuned models for validation
3. Creates optimized embeddings/representations that require less context to convey meaning
4. Enables more precise matching between queries and relevant information
5. Is exposed through a simple API endpoint on Lamini, and can easily be controlled by user prompts

By investing compute during embedding creation rather than retrieval, Memory RAG achieves two crucial benefits:

1. Higher accuracy through better data representation
2. Faster inference through smaller, more targeted context windows

Even smaller, simpler LLMs can achieve high accuracy, as they're working with pre-validated, well-structured embeddings rather than having to process complex raw data. This embed-time optimization approach means that while initial setup may take longer, each query benefits from the improved data quality without additional runtime overhead. Furthermore, this flexibility in model selection — ranging from lightweight mini-agents for simple tasks to more sophisticated ones for complex reasoning — makes Memory RAG adaptable for a future of agentic workflows.

FIGURE 3: Accuracy vs. Implementation Complexity

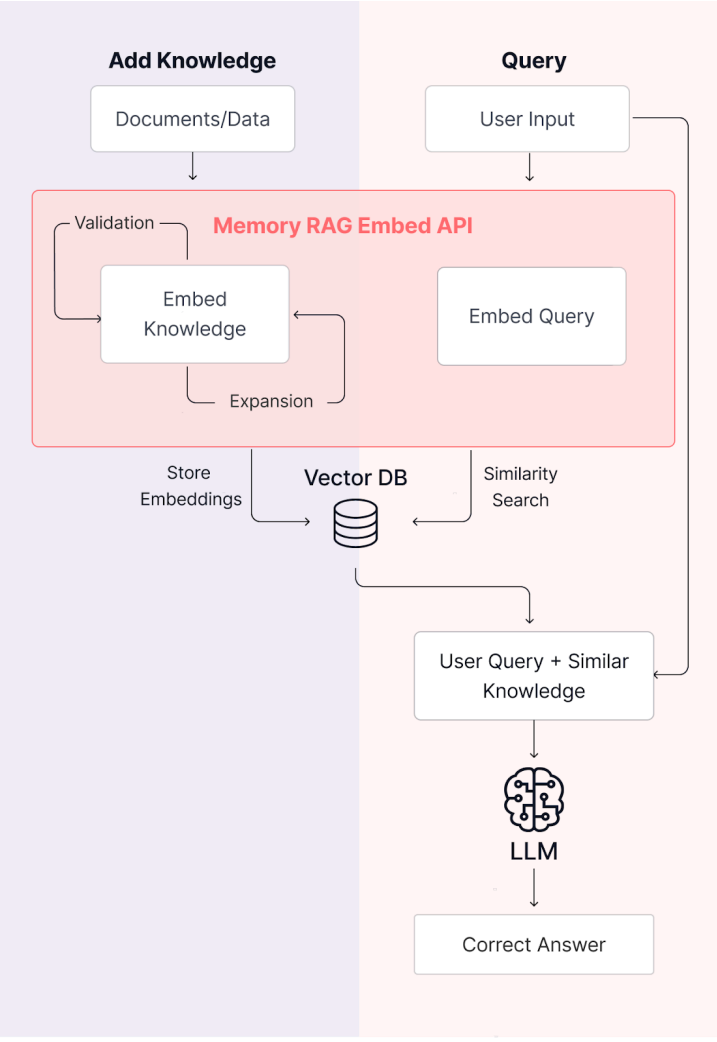


2. Technical Overview

2.1 Core Architecture

The core value of Memory RAG is through improved embedding generation — and the rest operates as a regular RAG system. The memory embedding phase transforms input documents into optimized embeddings using a distilled LLM approach, while the retrieval and inference phases leverage these highly reliable embeddings to provide accurate responses to user queries.

FIGURE 4: Memory RAG Process



2.2 Advanced Embedding and Validation using Embed-Time Compute

Memory RAG's core innovation lies in how it processes and validates information during the embedding generation phase using a distilled LLM approach. The system's validation capabilities lead to both immediate accuracy improvements and future agent-based extensions. While the underlying technology is sophisticated, using it is simple — users just need to provide their documents (PDFs, documentation, SQL schemas) and optionally add prompts to guide the process. The system handles all the complexity:

1. **Automated Knowledge Distillation**: Our system automatically specializes a large language model for your specific domain and data. Think of it like having an AI teaching assistant that rapidly learns your domain from the documents you provide, without requiring any special preparation or training data from you. Whether you're working with technical documentation, SQL schemas, or domain-specific content, the system adapts automatically.
2. **Enhanced Processing**: The specialized model then analyzes your content while embedding and indexing it, all behind a simple API:
    - Identifies key information and relationships
    - Recognizes domain-specific patterns and terminology
    - Creates rich, domain-aware representations
    - Validates information accuracy against known patterns
3. **Automated Validation**: Our system applies sophisticated checks to verify information accuracy and consistency — you don't need to define these checks yourself. This forms the foundation for agent-based operations through:
    - Learned validation patterns that can inform agent decision-making
    - Accumulated domain knowledge that agents can leverage
    - Established verification mechanisms that agents can use to self-validate
    - Flexible framework for incorporating new validation strategies as agent capabilities evolve

    The process works automatically across different types of content:

    - Technical documentation
    - Database schemas

    - Product information
    - Domain-specific content
4. **Intelligent Compression**: The system automatically preserves the most important aspects of your information while removing redundancy. You don't need to specify what's important - the system learns this from your data and any optional prompts you provide.
5. **Optimized Output**: The final representations are automatically structured to enable:
    - Quick, accurate matching with queries
    - Minimal context needed for understanding
    - Reliable validation of information accuracy
    - Efficient storage and retrieval

All of this complexity is abstracted behind a simple interface — you provide your documents and optionally some prompts to guide the process, and Memory RAG handles the rest. There's no need to understand the underlying distillation process or create special training data. This makes Memory RAG particularly valuable for teams that want the benefits of advanced LLM techniques without the complexity of implementing them.

2.3 Embed-Time Compute Advantage

Memory RAG's approach to embed-time compute is distinctive in that it frontloads computational effort during embedding generation:

1. **Embedding-Time Investment**: By using significant computational resources during embedding generation, Memory RAG creates higher-quality data representations that improve all future queries.
2. **Inference Efficiency**: This upfront investment means that during inference:
    - Less context is needed for accurate responses
    - Retrieval can be more precise
    - Response generation is faster and more reliable
3. **Scalability Benefits**: The embed-time compute approach means that:
    - Processing costs are incurred once while creating embeddings rather than for every query

    - Query latency remains low even as data volume grows
    - System accuracy improves with more thorough initial processing
4. **Quality Control**: The embed-time compute phase includes:
    - Automated content validation
    - Relationship verification
    - Consistency checking
    - Information completeness assessment

2.4 Bridge to Memory Tuning

Memory RAG is designed to provide a natural progression path to Memory Tuning, our fine-tuning solution:

1. **Shared Infrastructure**: Both Memory RAG and Memory Tuning use the same underlying data preparation and evaluation frameworks, enabling seamless transitions between the two approaches.
2. **Progressive Enhancement**: Organizations can start with Memory RAG for immediate accuracy improvements, then transition to Memory Tuning when ready for even greater control and accuracy, without rebuilding their data pipeline.
3. **Unified Evaluation**: The validation systems used in Memory RAG align with Memory Tuning's evaluation frameworks, providing consistent quality metrics across both approaches.
4. **Knowledge Transfer**: The semantic understanding developed through Memory RAG's validation system can inform and enhance the fine-tuning process in Memory Tuning, leading to more efficient model adaptation.

2.5 User Control Through Prompt Engineering

A key advantage of Memory RAG is that it provides direct user control over how information is processed and represented during the embedding phase. While the system uses advanced techniques for data optimization, users can guide this process through familiar prompt engineering approaches:

1. **Customizable Data Processing**: Users can define how information should be interpreted and represented through prompt engineering:
    - Specify important relationships to track
    - Define domain-specific terminology
    - Prioritize certain types of information
    - Guide the validation criteria
2. **Expert Knowledge Integration**: Domain expertise can be incorporated through prompts that:
    - Define industry-specific contexts
    - Establish validation rules
    - Specify relationship hierarchies
    - Guide information categorization
3. **Flexible Control Levels**: Users can exercise varying degrees of control:
    - Use default settings for quick deployment
    - Apply light customization through basic prompts
    - Implement detailed control through advanced prompt engineering
    - Fine-tune the system's behavior over time
4. **Transparent Iteration**: Users can:
    - Review how their prompts affect data representation
    - Adjust prompts based on results
    - Maintain consistent control across updates
    - Scale prompting patterns across similar data types

This user control layer means that while Memory RAG performs advanced optimizations under the hood, organizations maintain direct influence over how their domain knowledge is processed and represented. This combination of sophisticated processing with familiar control mechanisms makes Memory RAG both powerful and accessible.

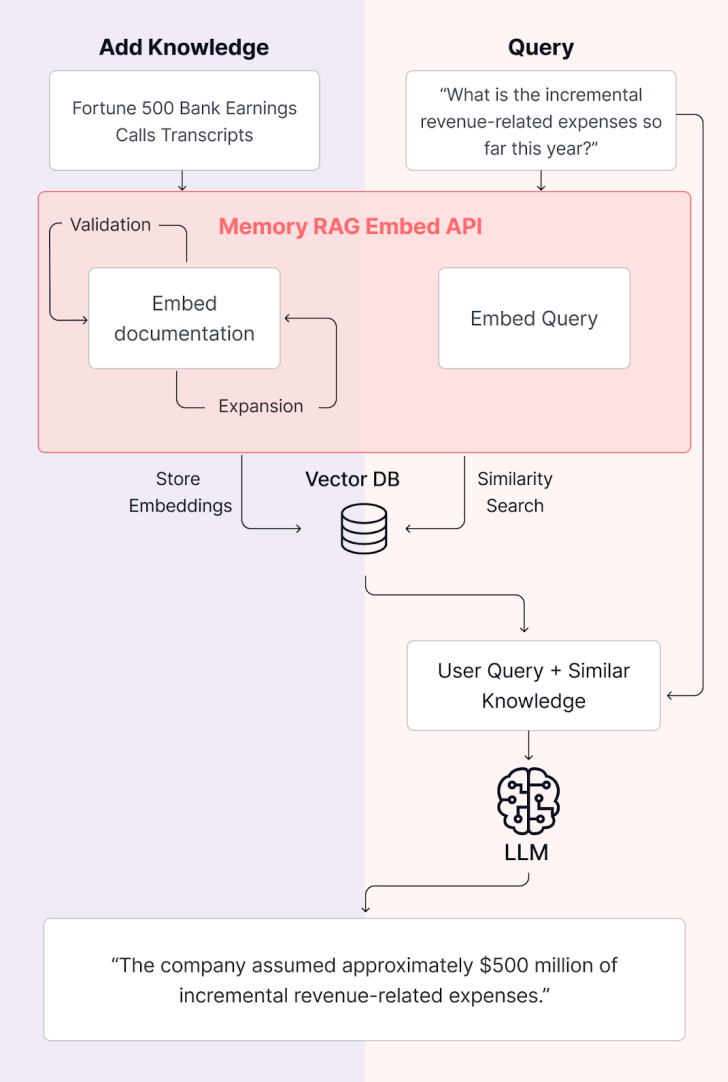
3. Key Applications

3.1 Factual Reasoning

Memory RAG excels in applications requiring high-accuracy factual responses, such as:

- Technical documentation queries
- Product information systems
- Regulatory compliance checking
- Medical information retrieval

FIGURE 5: Memory RAG for Factual Reasoning

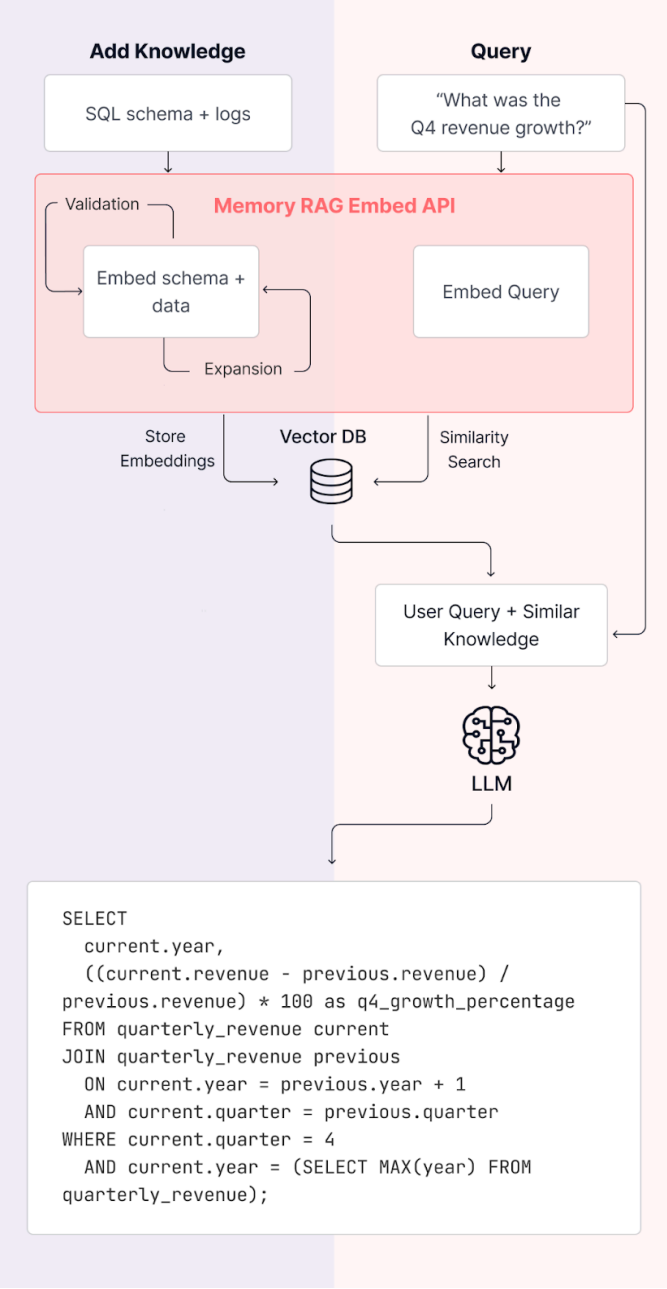


3.2 Text-to-SQL

Memory RAG provides particular advantages for text-to-SQL applications:

- Schema understanding and validation
- Query intent mapping
- Complex join relationship handling
- Business logic preservation

FIGURE 6: Memory RAG for Text-to-SQL



4. Implementation

4.1 API Design

Memory RAG provides a simplified API that follows familiar RAG patterns:

- POST - /train
  - Build out the Memory RAG index from the provided resources (PDFs). Iterate through the content and generate higher quality index content that is more relevant during inference time.
  - Parameters
    - files
      - Local file paths to be sourced during the REST call
    - model_name
      - LLM model ID on Hugging face

```
curl --location 'https://api.lamini.ai/alpha/memory-rag/train' \
            --header 'Authorization: Bearer $LAMINI_API_KEY' \
             --form 'files=@"{path/to/document}.pdf"' \
             --form
'model_name="meta-llama/Meta-Llama-3.1-8B-Instruct"
```

from lamini import MemoryRAG

```
client = MemoryRAG("meta-llama/Meta-Llama-3.1-8B-Instruct",)
```

```
response = client.memory_index(documents=["path/to/document.pdf"])
job_id = response['job_id']
```

- POST - /status
  - Check the status of the Memory RAG train job
  - Parameters
    - job_id
      - Memory RAG Index ID

```
curl --location 'https://api.lamini.ai/alpha/memory-rag/status \
            --header 'Authorization: Bearer $LAMINI_API_KEY' \
             --data '{
          "prompt": “Content to input into the index.”,
          "job_id": 1
      }'
```

from lamini import MemoryRAG

```
client = MemoryRAG("meta-llama/Meta-Llama-3.1-8B-Instruct”)
status = client.status(job_id)
```

- POST - /add-index
  - Update an existing index with provided content
  - Parameters
    - prompt
      - Content to be placed into the Memory RAG Index

        - job_id
            - Memory RAG Index ID

```
     curl --location
'https://api.lamini.ai/alpha/memory-rag/add-index \
                 --header 'Authorization: Bearer $LAMINI_API_KEY' \
                  --data '{
               "prompt": “Content to input into the index.”,
               "job_id": 1
          }'
```

from lamini import MemoryRAG

```
client = MemoryRAG("meta-llama/Meta-Llama-3.1-8B-Instruct")
response =
client.add_index("<|begin_of_text|><|start_header_id|>user<|end_header
_id|>\n\n How are you?
<|eot_id|><|start_header_id|>assistant<|end_header_id|>\n\n",
job_id=1)
```

- POST - /completions
    - Inference endpoint that will use the requested Memory RAG index
    - Parameters
        - prompt
            - Input for the Memory RAG Inference call
        - model_name
            - LLM to be used to generate the completion with the Memory RAG input

- job_id
  - Memory RAG index ID
- rag_query_size
  - Number of queries to return from the Index during inference

```
curl --location
'https://api.lamini.ai/alpha/memory-rag/completions' \
--header 'Authorization: Bearer $LAMINI_API_KEY' \
--header 'Content-Type: application/json' \
--data '{
"prompt":
"<|begin_of_text|><|start_header_id|>user<|end_header
_id|>\nRAG Query
<|eot_id|><|start_header_id|>assistant<|end_header_id
|>\n",
"job_id": 1,
“model_name”:
“model_name="meta-llama/Meta-Llama-3.1-8B-Instruct”
}'
```

from lamini import MemoryRAG

```
client = MemoryRAG("meta-llama/Meta-Llama-3.1-8B-Instruct")
response =
client.query("<|begin_of_text|><|start_header_id|>user<|end_header_id|
>\n\n How are you?
```

```
<|eot_id|><|start_header_id|>assistant<|end_header_id|>\n\n",
job_id=1)
```

So, just curl in your PDF to build a Memory RAG embeddings index:

```
curl --location 'https://api.lamini.ai/alpha/memory-rag/train' \
    --header 'Authorization: Bearer $LAMINI_API_KEY' \
    --form 'files=@"{path/to/document}.pdf"' \
    --form 'model_name="meta-llama/Meta-Llama-3.1-8B-Instruct"'
```

And after it's done embedding, query the Memory RAG model with a prompt:

```
curl --location 'https://api.lamini.ai/alpha/memory-rag/completions' \
    --header 'Authorization: Bearer $LAMINI_API_KEY' \
    --header 'Content-Type: application/json' \
    --data '{
        "prompt":
"<|begin_of_text|><|start_header_id|>user<|end_header_id|>\n\n What is
the growth rate for Wells Fargo in Q4
2024?<|eot_id|><|start_header_id|>assistant<|end_header_id|>\n\n",
        "job_id": 1
    }'
```

Or, if you're using python, here's a snippet from the SDK:

```python
from lamini import MemoryRAG
```

Build a Memory RAG embeddings index

```
memory_rag = MemoryRAG("meta-llama/Meta-Llama-3.1-8B-Instruct")
response = memory_rag.memory_index(documents=docs_in_pdf)
job_id = response['job_id']
```

Query the Memory RAG model with a prompt

```
response = memory_rag.query(llama_3_prompt_template("What is the
growth rate for Wells Fargo in Q4 2024?"))
```

4.2 Performance Optimization

Memory RAG includes built-in optimizations behind the simple API:

- Efficient embedding storage and retrieval
- Automatic batch processing for embedding
- Configurable validation thresholds
- Streamlined integration with Memory Tuning, to move from RAG to fine-tuning easily

5. Empirical Results

Our experiments across various use cases demonstrate significant improvements over traditional RAG implementations:

5.1 Financial Document Analysis

In a comprehensive evaluation using Wells Fargo earnings calls covering multiple years (2020-2024):

- Memory RAG achieved 91% accuracy
- Average response time of 5.76 seconds
- Traditional RAG with GPT-4 achieved only 59% accuracy

	Memory RAG	RAG on GPT4
Index build time	4 min 30 seconds	7.73 seconds
Avg response time	5.05 seconds	3.69 seconds
Accuracy	91%	59%

5.2 Database Query Understanding

In tests involving complex database schemas using real production data from a Fortune 500 company (174 examples):

- Memory RAG achieved 95% accuracy with average response time of 1.13 seconds
- Traditional RAG with GPT-4:
    - Basic prompt engineering: 0% accuracy
    - Enhanced RAG with SQL queries: 20% accuracy

	Memory RAG	RAG on GPT4
Index build time	32 seconds	1.04 seconds
Avg response time	1.13 seconds	3.14 seconds
Accuracy	95%	20%

These results demonstrate Memory RAG's ability to maintain high accuracy while providing responsive performance across different use cases. The system's approach with embed-time compute proves particularly valuable in complex scenarios like database query understanding, where traditional approaches struggle to maintain accuracy.

6. Conclusion

Memory RAG represents a significant step forward in making high-accuracy LLM applications more accessible to development teams. By providing a simplified approach that captures many benefits of fine-tuning without its complexity, Memory RAG enables organizations to improve their LLM applications while maintaining familiar development patterns.

Future Directions: Teaching agents through conversation

Looking ahead, Memory RAG opens up an exciting possibility in agent-based systems and iterative improvement of data pipelines and validators through conversation. Instead of manually configuring data transformations or writing complex validation rules, developers could refine their Memory RAG implementation through interactive debugging sessions.

For example, a developer could discuss specific cases where accuracy falls short, and the system could suggest pipeline adjustments or new validation rules, which the developer can then test and refine through further dialogue. Early research suggests this conversational approach could significantly accelerate the development cycle for high-accuracy LLM applications while maintaining precise control over the system's behavior.