Plans for Production-Ready RAG Chatbot: Deployment, Optimization, and Scalability Report

Juan David Vargas Mazuera
Université de Montréal, MILA - Quebec AI Institute
CHU Sainte-Justine Research Center

June 7, 2025

Abstract

This report presents a comprehensive analysis of deploying a Retrieval-Augmented Generation (RAG) chatbot for a production environment. The implementation leverages LangGraph's agentic framework, Grafana monitoring, Kubernetes orchestration, and advanced optimization techniques. We show how the system achieves performance through strategic deployment, monitoring, and optimization strategies while maintaining cost efficiency and reliability.

Contents

1 Introduction

The BMO RAG Chatbot represents prototype of a production-ready implementation of retrieval-augmented generation technology. This system combines state-of-the-art language models with agentic frameworks to provide accurate, contextual responses while maintaining comprehensive monitoring and optimization capabilities.

The implementation addresses critical production considerations including scalability, performance optimization, cost management, and reliability through systematic architectural decisions and deployment strategies.

2 System Architecture Overview

2.1 Core Components

The BMO RAG Chatbot consists of several interconnected components:

- **Core Application Files**: Main chatbot interface (`chatbot.py`), utility functions (`helper_functions.py`), and agentic implementation (`Langgraph_Agent.py`)
- **Deployment Infrastructure**: Automated setup (`setup.sh`) and Kubernetes deployment with Minikube (`deploy.sh`)
- **Monitoring Stack**: Prometheus metrics collection and Grafana visualization
- **Storage Layer**: Chroma vector database for document retrieval

The implementation utilizes LangGraph's agentic framework. The code demonstrates a sophisticated agent architecture with:

```
@tool
def retriever_tool(query: str) -> str:
    """This tool searches and returns information from a pdf document.
       """
    # Retrieval implementation

class AgentState(TypedDict):
    messages: Annotated[Sequence[BaseMessage], add_messages]
    user_query: str
    evaluation_score: float

# Multi-node agent graph
graph.add_node("llm", call_llm)
graph.add_node("retriever_agent", take_action)
graph.add_node("evaluator", evaluate_response)
```

3 Deployment Strategy

3.1 Platform Selection and Justification

3.1.1 Container Orchestration: Kubernetes

Platform Choice: Kubernetes with Minikube for local development

Justification:

- **Scalability**: Native horizontal scaling capabilities through replica sets
- **Reliability**: Self-healing mechanisms and health checks
- **Resource Management**: Efficient CPU/memory allocation and limits
- **Service Discovery**: Built-in load balancing and service mesh capabilities
- **Industry Standard**: Widely adopted in enterprise environments

3.1.2 Containerization: Docker

Implementation: Multi-stage Docker builds with optimized Python environments
Benefits:

- Environment consistency across development, staging, and production
- Dependency isolation and version control
- Efficient resource utilization through container sharing
- Simplified deployment and rollback procedures

3.2 Deployment Architecture

The deployment strategy implements three distinct versions:

1. **Version 1**: Base agentic RAG implementation
2. **Version 2**: Enhanced with LLM evaluator for response scoring
3. **Version 3**: Full implementation with context memory and temporal awareness

3.3 Cache Memory Optimization

Implementation Strategy: Streamlit's `@sl.cache_resource` decorator
Cache Usage Rationale:
Cache memory prevents repeated expensive operations on every Streamlit rerun:

- **Model Loading Prevention**: LLM and embedding model initialization (2-4 seconds saved per interaction)
- **Vector Store Persistence**: Document processing and indexing (2-4 seconds saved)
- **Memory Efficiency**: Single instance of large objects reduces memory footprint
- **User Experience**: an additional 10+ seconds without caching

Without caching, every user interaction would trigger:

```
# These expensive operations would run repeatedly:
llm, embeddings = initialize_llm_and_embeddings()  # ~2-4 seconds
pages_split = load_and_process_pdf(pdf_path)       # ~2-4 seconds
vectorstore = create_vector_store(pages_split, embeddings)  # ~2-4
    seconds
```

4 Performance Optimization

4.1 Implemented Optimization Techniques

The LangGraph implementation provides several optimization advantages:

- **Conditional Execution**: Tool usage only when necessary
- **State Management**: Efficient memory handling through typed dictionaries
- **Parallel Processing**: Concurrent tool execution capabilities
- **Error Handling**: Robust retry mechanisms and fallback strategies

4.2 Validation and Hallucination Mitigation

4.2.1 Content Validation Framework

Multi-Layer Validation Approach:

1. **Context Adherence Measurement**: Semantic and word-level overlap analysis
2. **LLM Evaluator Scoring**: Dedicated evaluator model assessing response quality (0.00-1.00 scale)
3. **Citation Requirements**: Mandatory source attribution in responses
4. **Temperature Control**: Set to 0 for deterministic, fact-based responses
5. **Chain of thought**: Implemented for llm evaluator

4.2.2 Hallucination Mitigation Strategies

- **Strict Prompting**: Clear instructions to cite sources and stay within context
- **Response Constraints**: Length and format limitations to prevent speculation
- **Evaluation Scoring**: Real-time quality assessment with threshold-based filtering (not yet implemented, there is a possibility to leverage the langgraph's architecture to loop on the llm's answer until a certain threshold score is achieved)
- **Human Feedback Loop**: User satisfaction scores for continuous improvement (implemented with reinforcement learning from human feedback to finetune the llm)

5 Agent Management and Orchestration

5.1 Agent Framework Benefits

Justification for Agentic Approach:
The agent framework is highly beneficial for this use case because:

- **Modular Design**: Separate agents for retrieval, generation, and evaluation
- **Dynamic Tool Usage**: Conditional retrieval based on query complexity
- **Quality Assurance**: Possibility to build automatic scoring mechanisms
- **Scalability**: Independent scaling of different agent components

5.2 Agent Orchestration Strategy

LangGraph State Management:

```
class AgentState(TypedDict):
    messages: Annotated[Sequence[BaseMessage], add_messages]
    user_query: str
    evaluation_score: float

# Orchestration flow
graph.add_conditional_edges(
    "llm",
    should_continue,
    {True: "retriever_agent", False: "evaluator"}
)
```

Orchestration Components:

1. **Primary LLM Agent**: Main response generation
2. **Retrieval Agent**: Document search and context gathering
3. **Evaluator Agent**: Response quality assessment
4. **State Coordinator**: Manages data flow between agents

5.3 Alternative: Model Context Protocol (MCP)

MCP vs. Vector Database Analysis:

MCP can enhance the chatbot but not replace vector databases entirely. MCP excels at real-time context synchronization and structured data access, potentially making it valuable for integrating live financial data at organizations like BMO capital markets.

6 Scalability Architecture

6.1 Horizontal Scaling Strategy

Kubernetes offers enterprise-grade native scaling. Several mechanisms make this possible:

Scaling Mechanisms:

1. **Horizontal Pod Autoscaler (HPA)**: Automatic scaling based on CPU/memory metrics
2. **Vertical Pod Autoscaler (VPA)**: Resource limit optimization
3. **Cluster Autoscaler**: Node-level scaling for demand fluctuations
4. **Load Balancing**: Even distribution of requests across replicas

6.2 Concurrent User Handling

Design for High Concurrency:

- **Stateless Design**: Each request is independent, enabling easy horizontal scaling
- **Connection Pooling**: Efficient database connection management
- **Caching Strategy**: Redis integration for session and result caching

7 Monitoring and Logging

7.1 Comprehensive Metrics Framework

Implemented Metrics and Justification:

Metric	Type	Business Justification
TOKENS_PER_QUERY	Gauge	Cost management and usage optimization
CONTEXT_ADHERENCE	Gauge	Quality assurance and hallucination prevention
CARBON_FOOTPRINT	Gauge	Environmental impact and sustainability reporting
USER_SATISFACTION	Gauge	Customer experience and product improvement
COST_PER_QUERY	Gauge	Financial efficiency and budget planning
ENERGY_PER_QUERY	Gauge	Resource optimization and cost calculation
LATENCY_PER_QUERY	Gauge	Performance monitoring and SLA compliance
SATISFACTION_PER_DOLLAR	Gauge	ROI measurement and value optimization
QUALITY_PER_DOLLAR	Gauge	Cost-effectiveness analysis
SCORE_LLM_EVALUATOR	Gauge	Automated quality assessment and improvement

7.2 Implemented Monitoring: Grafana Dashboard

Comprehensive Metrics Visualization:

The implemented Grafana dashboard provides real-time monitoring of critical chatbot performance metrics through a comprehensive visualization interface with more than 15 visualizations.

Monitoring Capabilities:

1. **Real-time Cost Tracking**: Immediate visibility into operational expenses and resource consumption. Also several metrics with performance vs cost trade-off.
2. **Quality Monitoring**: Continuous assessment of context adherence

3. **Performance Analysis**: Latency insights
4. **User Satisfaction Trends**: Detailed satisfaction score distributions and averages
5. **Environmental Impact**: Carbon footprint monitoring and sustainability reporting
6. **Operational Efficiency**: Token usage insights

8 Vector Database Analysis and Improvements

8.1 Current Implementation Limitations

Chroma Vector Database Constraints:

Chroma DB is primarily designed for single-node deployment with in-memory storage, limiting its scalability for enterprise-level applications with massive document collections. Its file-based persistence lacks the durability guarantees needed for production systems, and it offers minimal support for advanced features like metadata filtering, real-time updates, and complex queries.

8.2 Production-Ready Alternatives

Recommended Vector Database Solutions:

Unlike Chroma's development-focused design, Pinecone offers production-ready features including automatic backups, global distribution, API-first architecture, and seamless integration with existing MLOps pipelines, making it ideal for scaling RAG applications to handle thousands of concurrent users while maintaining consistent sub-second response times.

9 Retrieval and Reranking Improvements

The implementation of a hybrid retrieval approach combining semantic vector search with keyword-based (BM25) retrieval, followed by advanced reranking mechanisms, represents a significant opportunity for improving response accuracy and relevance. This multi-modal strategy leverages the strengths of both dense and sparse retrieval methods: semantic search excels at understanding conceptual similarity and context, while keyword search ensures exact term matching and handles specific terminology that embeddings might miss. The proposed cross-encoder reranking stage would provide precise query-document relevance scoring by evaluating the full interaction between query and candidate documents, rather than relying solely on embedding similarity. Additionally, the reranking phase enables sophisticated relevance scoring that considers query complexity, document completeness, and contextual appropriateness, ultimately reducing noise in retrieved context and improving the quality of generated responses.

10 Testing Framework

10.1 Comprehensive Testing Strategy

Multi-Layer Testing Approach:

1. **Unit Testing**: Individual component validation
2. **Integration Testing**: End-to-end system functionality
3. **Performance Testing**: Load and stress testing
4. **Quality Testing**: Response accuracy and relevance
5. **Security Testing**: Input validation and injection prevention

10.2 Quality Assurance Testing

Automated Quality Metrics:

- **Context Adherence**: Alignment with retrieved documents (0.00 - 1.00)
- **LLM Evaluator Score**: Automated quality assessment (0.00 - 1.00)
- **User Score**: Manual user score (0-5)

11 Cost Management

11.1 Cost Optimization Strategies

Multi-Dimensional Cost Reduction:
The BMO RAG Chatbot implements comprehensive cost optimization strategies across multiple operational dimensions to achieve maximum efficiency while maintaining high performance standards. Token optimization forms the foundation of cost control through efficient prompt engineering techniques that minimize unnecessary token usage, strategic context window optimization that maximizes information density per request (maximum 3 panels from the chat history). In addition,dition, intelligent caching mechanisms that prevent redundant API calls by storing frequently accessed results.

On the other hand, infraestructure optimization strategies could enhance cost-effectiveness. For one, the llm evaluator could be hosted locally, and an open-source llm should be implemented considering the simplicity of the task (evaluate the main llm's performance). This way, performance could be enhanced while costs are minimized.

12 Future Enhancements

12.1 Planned Improvements

1. Implementing hybrid retrieval

2. Host the application in the cloud, as well as redirect any outputs to a fine-tuning pipeline
3. Refine a pipeline that evaluates the llm's output, and iterates until a certain performance threshold is achieved
4. Reducing latency (e.g. evaluating other llms with a better performance vs latency trade-off)