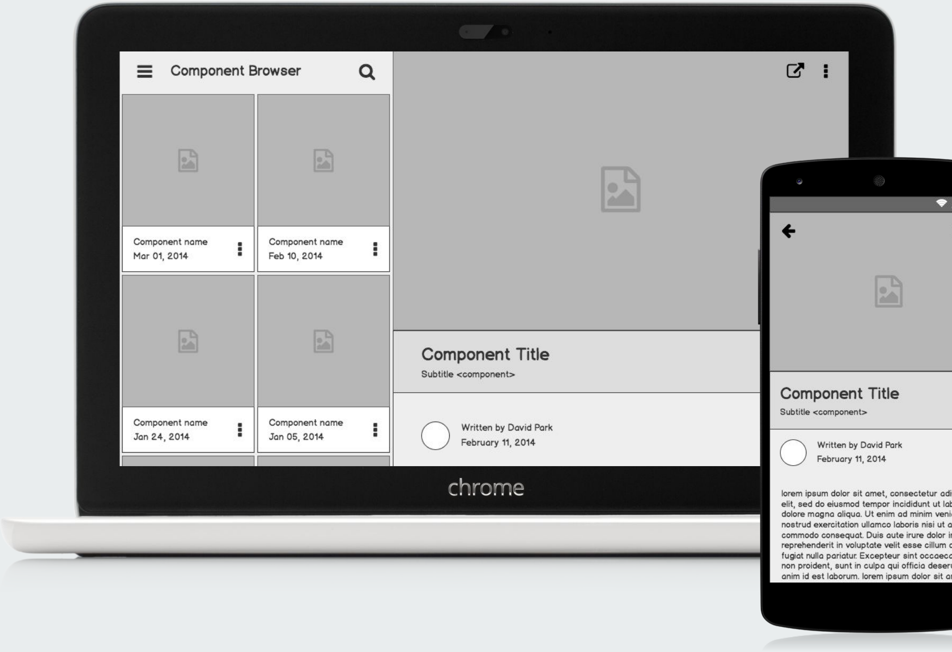
Build Your Gen-AI App

Rohan Srivastava



Outline

Overview of RAG Systems

Walkthrough: RAG Setup

Extensions & Resources

Q&A

Overview of RAG Systems

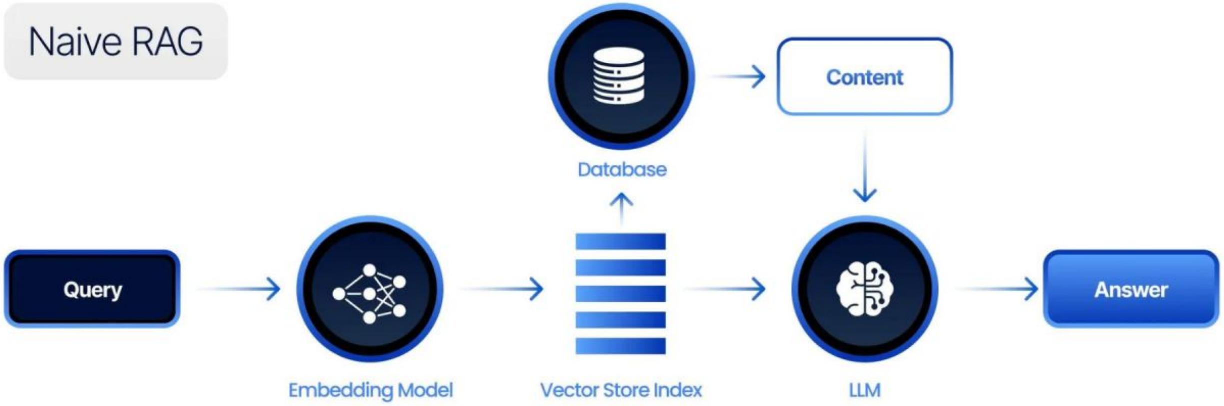
What is RAG?

Retrieval-Augmented Generation

RAG (Retrieval-Augmented Generation) is an AI framework that combines the strengths of traditional information retrieval systems (such as search and databases) with the capabilities of Large Language Models.

Examples: Perplexity, Google Gemini, How we do our assignments

Components of a RAG-System

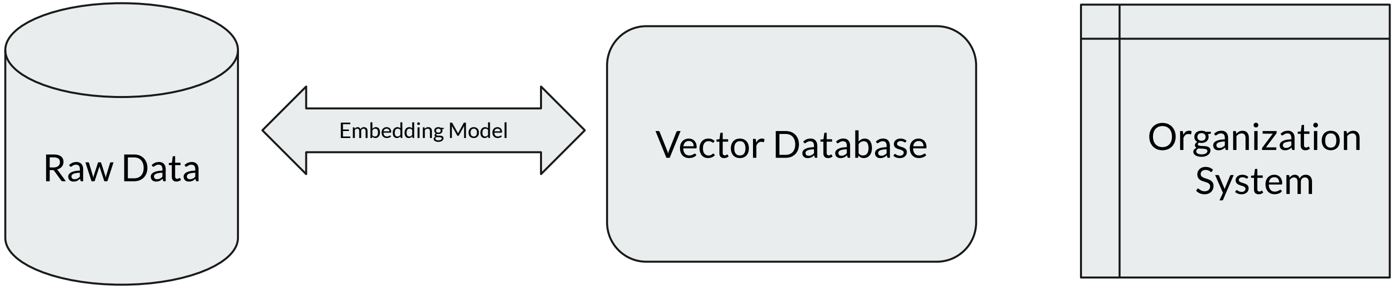


Traditional LLMs vs RAG Systems

- Traditional LLMs:
  - Have fixed knowledge
  - Prone to Hallucinations
  - Produce Generic Responses

- RAG Systems:
  - Custom-defined knowledge
  - Factual Outputs (Still some hallucination)
  - Domain-specific responses

Knowledge Base: Core Elements



Raw data in various formats that contains your custom knowledge

Converts text into numerical representations (embeddings) for efficient similarity search

Additional information like titles, dates, categories to enhance search quality

The Retrieval Process

- ➔ Query Embedding: Transforms your input question into a numerical vector representation, just like how we processed our knowledge base documents
- ➔ Similarity Search: Uses mathematical distance calculations to find the most similar content vectors in our database
- ➔ Top-K Selection: Selects the most relevant pieces of context based on similarity scores, typically choosing top 3-5 matches

Examples of Embedding Models: HuggingFace

Examples of Database Providers: Pinecone, Weaviate, ChromaDB, Milvius

The Generation Process

- ➔ Context Integration: Retrieved relevant documents are combined with the user query to create an enriched prompt
- ➔ Prompt Engineering: System formats the combined context and query using specific templates for optimal LLM understanding
- ➔ Response Generation: LLM generates response by balancing retrieved context with its base knowledge

Building RAG: Cover Letter Generator

Workshop Overview

- ➔ The goal of this workshop is to use your work experience for a cover letter generation
- ➔ Project Goal: Create a RAG system that matches your work experience (from Google Sheets) with new job requirements to generate relevant cover letters
- ➔ Input Design: Job description responsibilities as query, personal work history as knowledge base
- ➔ Output: Cover letter that highlights most relevant past experiences for the target role

General Steps to Write a Cover Letter

- ➔ Read the Job Description to understand roles & responsibilities
- ➔ Understand the team, company culture
- ➔ Look at what qualities align well with the job and use those to frame a story
- ➔ (Optional but great additions) Personal Interest in the Company

Workshop Requirements

- ➔ Gemini API Key
- ➔ Google Sheet with some work history
- ➔ Colab Notebook

Setup Knowledge Base: Google Sheet

- ➔ Core Data: Work experience descriptions become our embeddings for matching with job requirements
- ➔ Metadata: Organization, dates, position, and project names enrich our final outputs
- ➔ Format: Each row represents a distinct professional achievement, making retrieval granular

Retrieval Process

- ➔ Parse Job Description: Gemini first analyzes the input job posting to extract a clean, structured JSON of key responsibilities and requirements, making search more precise
- ➔ Query Formation: Each job responsibility is converted into a vector query to search our knowledge base effectively
- ➔ Context Retrieval: System finds the most relevant past experiences by matching requirement vectors with your work history embeddings

Prompt 2: Writing the Actual Letter

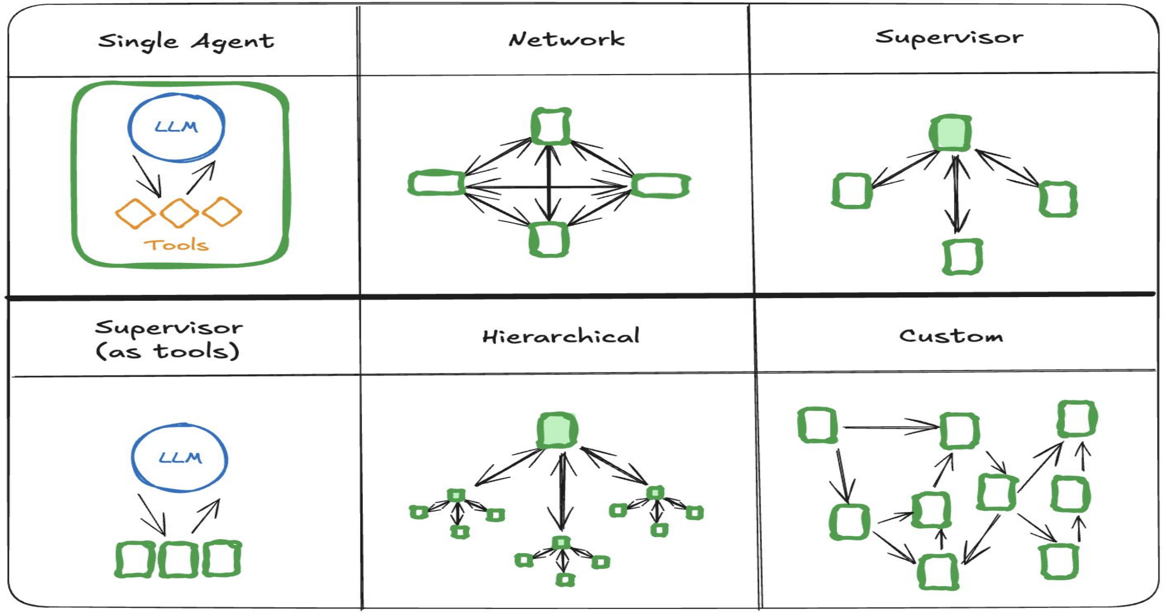
- ➔ Smart Matching: Code processes each job responsibility individually, finding top n most relevant experiences per requirement
- ➔ Template Structure: Uses a carefully crafted prompt with specific paragraph structure and word counts for consistency
- ➔ Dynamic Generation: Combines job details, matched experiences, and today's date to create a tailored cover letter

RAG Refinement & Extensions

Beyond Basic RAG

- ➔ Multi-Step Reasoning: Chaining RAG outputs through multiple queries for complex problem-solving -> Currently used in GPT-o1 and many other latest models
- ➔ Relevance Boosting: Introducing intentional Bias for more specific outputs
- ➔ Internet Search: Collecting online information as part of the RAG flow through Perplexity or Google Search APIs
- ➔ Memory Systems: Maintaining conversation history to provide contextually relevant responses
- ➔ Multiple Databases

AI Agents Architecture



Advanced Use-Cases

- ➔ Code Documentation: GitHub Copilot uses RAG to access repository-specific code context for better suggestions
- ➔ Legal Research: Harvey AI leverages RAG with law firm documents for case-specific legal research
- ➔ Medical Analysis: Mayo's implementation for analyzing patient records and medical literature
- ➔ Enterprise Search: Confluence and Notion using RAG to improve internal document search accuracy
- ➔ Customer Support: Intercom's implementation for retrieving company-specific support documentation

Future Directions

- Multi-Modal RAG: Systems that can retrieve and reference both text and image data (like Claude 3.5, GPT-4O,)
- Cross-Language RAG: Retrieval across multiple languages while maintaining semantic understanding
- Streaming RAG: Real-time document indexing and retrieval for constantly updating knowledge bases. Relevant for Stock Market Agents, Supply Chains, Healthcare, Cybersecurity
- Adaptive Retrieval: Systems that learn from user feedback to improve retrieval quality over time
- Compression Techniques: More efficient storage and retrieval of large-scale vector databases

Questions