LLAMAIndex, Crew AI and Auto Gen

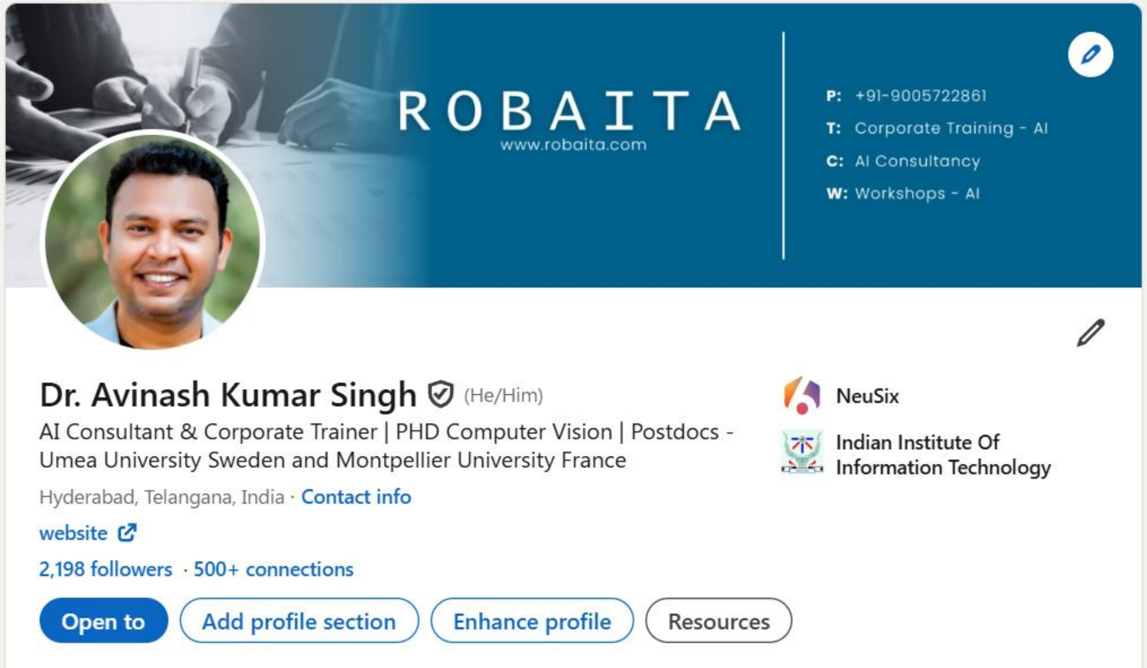
Building Resilient RAG Agents with Graph & LangGraph

Dr. Avinash Kumar Singh

AI Consultant and Coach, Robaita

Dr. Avinash Kumar Singh

- ❑ **Possess** 15+ years of **hands-on expertise** in Machine Learning, Computer Vision, NLP, IoT, Robotics, and Generative AI.
- ❑ **Founded** Robaita—an initiative **empowering** individuals and organizations to **build, educate, and implement** AI solutions.
- ❑ **Earned** a Ph.D. in Human-Robot Interaction from IIIT Allahabad in 2016.
- ❑ **Received** postdoctoral fellowships at Umeå University, Sweden (2020) and Montpellier University, France (2021).
- ❑ **Authored** 30+ research papers in **high-impact** SCI journals and international conferences.
- ❑ Unlearning, learning, making mistakes …



https://www.linkedin.com/in/dr-avinash-kumar-singh-2a570a31/

Discussion Points

- **Recap:** RAG, LangChain, Graph RAG, LangGraph, Security, Guardrails and Privacy
- **LlamaIndex**
  - Why LlamaIndex?
  - Core Components and RAG Pipeline Implementation
  - LlamaIndex in Multi-Tool Environments
- **Crew AI**
  - Why and What is Crew AI?
  - Crew AI execution flow
  - Integration of Crew AI with LangGraph and LlamaIndex
- **AutoGen**
  - Why and What is AutoGen?
  - AutoGen components
  - AutoGen execution flow
- **Next: Agent 2 Agent (A2A) Protocol**
  - Why and What is A2A Protocol.
  - Example and use cases.
  - Implementation Details

Retrieval Augmented Generation Architecture

Training

Data Ingestion
Uploading documents in the system to be used for model training

Data Chunking
Split the training documents into small chunks so that we can have better information representation

Data Embedding
Create embedding of each chunk (representing the information in vectors)

Embedding Storage
Store embedding of each chunk into the vector databases for the faster and efficient retrieval

Testing

Find the most relevant matches

Query

Query Embedding
Get the embedding of your query. A vector

Top-K Results
Retrieve top k results (chunks) from the database based on similarity

LLM
Provide Query + Prompt + Results to generate response

LlamaIndex

https://www.llamaindex.ai/

LlamaIndex

LlamaIndex is a data framework for LLM applications — it helps connect external data (like PDFs, databases, websites) to Large Language Models (LLMs), making it easier to build Retrieval-Augmented Generation (RAG) systems.

Core functionality includes:

📥 **Ingestion:** Load data from multiple formats (PDFs, Notion, SQL, etc.)

🧠 **Indexing:** Build searchable indexes using embedding models

🔍 **Retrieval:** Retrieve relevant chunks based on user queries

💬 **Querying:** Ask questions using LLMs with retrieved context

Creator: Jerry Liu, former ML engineer at Uber

Organization: Initially developed as an open-source project under the name GPT Index

Rebranded as: LlamaIndex in late 2022

Current Maintainer: LlamaIndex team — a dedicated company building the ecosystem

LlamaIndex

Boilerplate refers to **repetitive, standard code** that developer has to write regularly

Abstraction Over RAG Complexities

LlamaIndex offers a high-level API that abstracts away the boilerplate involved in:

Data ingestion (from PDFs, APIs, SQL, Notion, etc.).

Chunking, indexing, and **retrieval**.

Prompt orchestration and **context optimization.**

```
from llama_index.core import VectorStoreIndex, SimpleDirectoryReader

documents = SimpleDirectoryReader("data").load_data()
index = VectorStoreIndex.from_documents(documents)
query_engine = index.as_query_engine()
response = query_engine.query("What is the summary of Chapter 3?")
```

Plugin-Friendly Architecture

LlamaIndex integrates smoothly with:

Vector DBs like FAISS, Pinecone, Qdrant

LLMs: OpenAI, Claude, LLaMA, Mistral

Tools: LangChain, Weaviate, Chroma, etc.

```
from llama_index.core.retrievers import VectorIndexRetriever
from llama_index.retrievers.bm25 import BM25Retriever
from llama_index.core.retrievers import QueryFusionRetriever

bm25 = BM25Retriever.from_documents(documents)
vector = VectorIndexRetriever(index=index)
hybrid = QueryFusionRetriever([bm25, vector], mode="reciprocal_rerank", similarity_top_k=5)
```

LlamaIndex

Comparison with LangChain

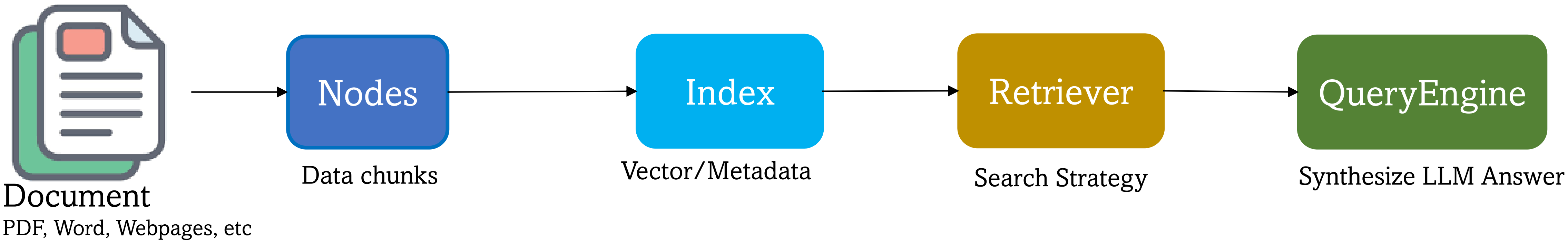
Feature	LangChain	LlamaIndex
Philosophy	Composable + low-level chains	Data-centric + optimized APIs
Custom Graphs	LangGraph + LangChain	Recently added (less mature)
Ease of Use	Requires stitching components	Unified interface for RAG
Best For	Custom pipelines + tools	Fast prototyping + data indexing

Real-World Adoption

LlamaIndex powers:

- Enterprise document Q&A bots
- Compliance search engines
- AI copilots for structured data

LlamaIndex - Architecture



Component	Description	Example
Document	Raw input data source (text, PDF, HTML)	"Ramayana.pdf" as a full document
Node	Semantically meaningful chunk derived from the Document	A paragraph about Rama's exile
Index	Organizes nodes for efficient retrieval (e.g., VectorIndex, TreeIndex, KeywordTableIndex)	VectorIndex built on RAMAYANA node embeddings
Retriever	Finds relevant nodes using similarity search, keyword lookup, etc.	Cosine similarity finds 3 chunks about Kaikeyi's role
QueryEngine	Executes queries, runs retrieval, and synthesizes answers using LLMs	Query: "Why did Rama go to the forest?" → Synthesized multi-node answer

LlamaIndex – Data Loading

Before indexing or querying, the data (regardless of format) must be:

- **Loaded** into memory
- **Parsed** into structured documents
- **Standardized** into the Document format understood by LlamaIndex

SimpleDirectoryReader

A built-in loader that recursively scans a directory for files and loads them into Document objects. Supports .pdf, .txt, .docx, .md, .csv, .json

```
from llama_index.core import SimpleDirectoryReader

documents = SimpleDirectoryReader(input_files=["data/RAMAYANA.pdf"]).load_data()
print(f"Loaded {len(documents)} document(s).")
print(documents[0].text[:100])  # Preview
```

Custom Loader

For structured formats or APIs, define custom loaders using BaseReader or by subclassing.

```
import pandas as pd
from llama_index.core.schema import Document

df = pd.read_csv("data/headcount_2025.csv")
doc_text = df.to_markdown(index=False)

document = Document(text=doc_text, metadata={"source": "headcount_2025.csv"})
print(document.text[:500])  # Preview)
```

LlamaIndex – Data Loading

Third-Party & Web Loaders

- Google Docs, Notion, Websites, YouTube transcripts
- Use llama-hub to access external data with plug-and-play loaders.

Loaders for:

- Notion → NotionPageReader
- YouTube → YoutubeTranscriptReader
- Slack → SlackReader
- Google Docs → GoogleDocsReader
- HTML → BeautifulSoupWebReader

```python
from llama_index.readers.web import SimpleWebPageReader

urls = ["https://en.wikipedia.org/wiki/Ramayana"]

# html_to_text=True is recommended for cleaner text extraction
reader = SimpleWebPageReader(html_to_text=True)
documents = reader.load_data(urls)

if documents:
    print(documents[0].text[:500])
    print("\nMetadata:")
    print(documents[0].metadata)
else:
    print("No documents loaded.")
```

LlamaIndex – Data Chunking

Overview of Chunking Techniques in LlamaIndex

Technique	How It Works	Example
Default Chunking	Simple fixed-length chunks (e.g., 512 tokens), breaks without semantic awareness	Splitting Ramayana.pdf into equal 512-token chunks
RecursiveTextSplitter	Tries to split by sentences → paragraphs → characters, fallback if structure isn't present	Breaks Chapter 1 at sentence boundaries if possible
SemanticSplitterNodeParser	Uses embeddings to find semantically coherent split points	Groups verses discussing Rama's birth together
Graph-Based Chunking	Builds a graph of entities, links semantically related passages based on co-occurrence/context	All mentions of Kaikeyi in different chapters form a connected chunk

LlamaIndex – Data Chunking

Default Chunking

```
from llama_index.core.node_parser import SimpleNodeParser

parser = SimpleNodeParser.from_defaults(chunk_size=512)
nodes = parser.get_nodes_from_documents(pdf_document)

for node in nodes[:2]:
    print(node.text)
```

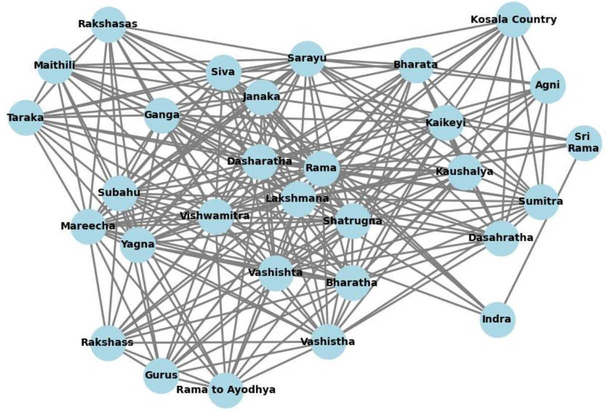
RecursiveTextSplitter

```
from llama_index.core.node_parser import LangchainNodeParser
from langchain.text_splitter import RecursiveCharacterTextSplitter

text_splitter = RecursiveCharacterTextSplitter(chunk_size=500, chunk_overlap=50)
parser = LangchainNodeParser(lc_splitter=text_splitter)

nodes = parser.get_nodes_from_documents(pdf_document)

for idx, node in enumerate(nodes[:2]):
    print(f"Chunk {idx+1}\n Chunk Text:{node.text}")
```

Semantic Chunking

```
from llama_index.core.node_parser import SemanticSplitterNodeParser
from llama_index.embeddings.huggingface import HuggingFaceEmbedding

embed_model = HuggingFaceEmbedding(model_name="all-MiniLM-L6-v2", device="cpu")
parser = SemanticSplitterNodeParser(embed_model=embed_model, chunk_size=512)

nodes = parser.get_nodes_from_documents(pdf_document)

for node in nodes[:2]:
    print(node.text)
```

Graph-based Chunking



LlamaIndex – Data Embedding

- Embeddings convert text into numerical vectors representing semantic meaning.
  - Embedding Models: OpenAI, HuggingFace, Ollama, etc.
- Vector Stores store and retrieve those embeddings efficiently for similarity search (used in RAG, semantic search, chatbots).
  - Vector Stores: FAISS (local), Chroma, Pinecone, Qdrant (hosted or local)

```python
from llama_index.core import VectorStoreIndex
from llama_index.vector_stores.faiss import FaissVectorStore
from llama_index.embeddings.huggingface import HuggingFaceEmbedding
import faiss

EMBEDDING_DIM = 384
embed_model = HuggingFaceEmbedding(model_name="all-MiniLM-L6-v2", device="cpu")

faiss_index = faiss.IndexFlatL2(EMBEDDING_DIM)
vector_store = FaissVectorStore(faiss_index=faiss_index)

index = VectorStoreIndex(
    nodes=nodes,
    vector_store=vector_store,
    embed_model=embed_model,
)

query_engine = index.as_query_engine()
response = query_engine.query("What is the role of Rama in Ramayana?")
print(response)
```

```
The Answer is taken from Page: 45, 29
🧠 Answer:
 Rama plays a central role in the Ramayana as the protagonist and hero
of the epic. He is depicted as an ideal king, husband, and son, embody
ing virtues such as righteousness, devotion, and courage. Rama's journ
ey, from being crowned the king of Ayodhya to his exile in the forest,
 his search for Sita, and the eventual battle against Ravana, showcase
s his unwavering commitment to dharma and his willingness to sacrifice
 personal happiness for the greater good. Throughout the epic, Rama's
character serves as a moral compass, inspiring readers to follow his i
deals of truth, duty, and compassion.
```

LlamaIndex - Choosing the Right Index

Picking the Right Tool for the Information Retrieval Needs

Index Type	Description	Best Used For
VectorStoreIndex	Embedding-based similarity search	Semantic Q&A, unstructured text
KeywordTableIndex	Inverted index for exact/keyword match	Document filtering, precise keyword lookup
ListIndex	Linear scan through documents	Sequential queries, story-based answers
KnowledgeGraphIndex	Constructs a graph from entities + relationships	Entity search, reasoning over concepts

Example Use Cases

Use Case	Recommended Index
Semantic Search over PDFs	VectorStoreIndex
“Find all documents mentioning X”	KeywordTableIndex
“Summarize this series of emails”	ListIndex
“Who is related to Ravana?”	KnowledgeGraphIndex

LlamaIndex - Choosing the Right Index

Semantic Search

```
from llama_index.core import VectorStoreIndex
from llama_index.embeddings.huggingface import HuggingFaceEmbedding

embed_model = HuggingFaceEmbedding(model_name="all-MiniLM-L6-v2", device="cpu")
index = VectorStoreIndex.from_documents(pdf_document, embed_model=embed_model)

query_engine = index.as_query_engine()
response = query_engine.query("What values does Ramayana teach?")
```

```
🧠 Answer:
 Ramayana teaches values such as devotion towards parents, ideal behavi
or as a king, respect for all individuals, ruling a kingdom well, trut
hfulness, and being virtuous and valiant.
```

Sequential Answering

```
from llama_index.core import ListIndex

index = ListIndex.from_documents(documents)

query_engine = index.as_query_engine()
response = query_engine.query("Give me a summary chapter by chapter")
```

```
🧠 Answer:
 The epic "Ramayana" is divided into seven main chapters, known as Kand
as. The chapters are as follows:

1. Bāla Kāṇḍa
2. Ayodhyā Kāṇḍa
3. Ar
aṇya Kāṇḍa
4. Kiṣkindhā Kāṇḍa
5. Sundara Kāṇḍa
6. Yuddha Kāṇḍa
7. Utta
ra Kanda

Each chapter focuses on different aspects of the story of Lo
rd Rama, his wife Sita, and his loyal companion Hanuman, as they navig
ate through various challenges and adventures.
```

Exact Match Search

```
from llama_index.core import KeywordTableIndex

index = KeywordTableIndex.from_documents(documents)

query_engine = index.as_query_engine()
response = query_engine.query("Mentions of Ayodhya and Lanka")
```

```
🧠 Answer:
 The epic _Ramayana_ narrates the life of Rama, a prince of Ayodhya in
the kingdom of Kosala, and his eventual return to Ayodhya to be crowne
d as a king. It also includes the kidnapping of Sita by Ravana, the ki
ng of Lanka.
```

LlamaIndex - Retrieval with Filters & Hybrid Search

Metadata Filtering: Use metadata (e.g., document type, topic, source) to filter documents during retrieval. This helps narrow the context to only relevant documents.

Example Use Case: Filter only chapters from the Ramayana that are tagged with "location": "Ayodhya".

```
from llama_index.core.vector_stores.types import MetadataFilters, MetadataFilter
from llama_index.core.retrievers import VectorIndexRetriever
from llama_index.core.query_engine import RetrieverQueryEngine

# Build filters using MetadataFilter and MetadataFilters
filters = MetadataFilters(
    filters=[MetadataFilter(key="location", value="Ayodhya")]
)

retriever = VectorIndexRetriever(index=index, filters=filters)
query_engine = RetrieverQueryEngine(retriever=retriever)
response = query_engine.query("Who ruled Ayodhya?")
print("🧠 Answer:\n", '\n'.join([''.join(response.response[i:i + 70]) for i in range(0, len(response.response), 70)]))
```

```
🧠 Answer:
Bharatha ruled Ayodhya in place of Rama during Rama's absence.
```

LlamaIndex - Retrieval with Filters & Hybrid Search

Hybrid Retrieval (Keyword + Vector Search)

Combine semantic vector similarity with keyword matching to improve recall and precision.

Example Use Case:

Get results that either contain the keyword “Rama” or are semantically similar to the query “prince of Ayodhya”.

```
🧠 Answer:
Rama
```

```python
from llama_index.core.indices.keyword_table import KeywordTableIndex
from llama_index.core.retrievers import VectorIndexRetriever
from llama_index.core.retrievers import QueryFusionRetriever

# Create both vector and keyword indexes
vector_index = VectorStoreIndex.from_documents(documents)
vector_retriever = VectorIndexRetriever(index=vector_index)
keyword_index = KeywordTableIndex.from_documents(documents)
keyword_retriever = keyword_index.as_retriever(similarity_top_k=5)

# hybrid = QueryFusionRetriever([vector_index, keyword_index], mode="reciprocal_rerank", similarity_top_k=5)
hybrid_retriever = QueryFusionRetriever(
    [vector_retriever, keyword_retriever],
    mode="reciprocal_rerank",
    similarity_top_k=5,
    num_queries=4, # Number of synthetic queries to generate
)

query_engine = RetrieverQueryEngine(retriever=hybrid_retriever)
response = query_engine.query("prince of Ayodhya")
print("🧠 Answer:\n", '\n'.join([''.join(response.response[i:i + 70]) for i in range(0, len(response.response), 70)]))
```

LlamaIndex - Graph RAG and Entity-Aware Retrieval

What is Graph RAG?

Graph RAG augments standard RAG by constructing a Knowledge Graph (KG) from documents using entities and relationships, enabling entity-aware, context-rich retrieval.

Key Steps

- Parse: Extract entities and map relationships from text
- Map: Represent text chunks as nodes with entity metadata
- Link: Build a Knowledge Graph from entity co-occurrence or semantic linkage

```python
import spacy
import networkx as nx

# Load NLP model for entity extraction
nlp = spacy.load("en_core_web_sm")

# Example text (replace this with your parsed Ramayana chapters)
text = """King Dasharath of Ayodhya had three queens: Kaushalya, Sumitra, and Kaikeyi.
Ram, the eldest, was born to Kaushalya. He later married the daughter of Janak, Sita, daughter of Janak from Mithila."""

# Step 1: Parse → Extract Entities
doc = nlp(text)
entities = list(set(ent.text for ent in doc.ents if ent.label_ in ["PERSON", "ORG", "GPE", "LOC"]))
print(f"Extracted Entities: {entities}")


# Step 2: Map → Create Node
node = {"text": text, "entities": entities}

# Step 3: Link → Graph Creation
G = nx.Graph()
G.add_node(text, entities=entities)

# Add edges based on entity co-occurrence
for i, e1 in enumerate(entities):
    for e2 in entities[i+1:]:
        G.add_edge(e1, e2, source_text=text)

print(f"📊 Entities: {entities}")
print(f"🔗 Graph edges: {G.edges(data=True)}")
```

LlamaIndex - Graph RAG and Entity-Aware Retrieval

Use Graph for Entity-Aware Retrieval

Query: “How Janak is related to Mithila?“

We can retrieve the **subgraph** or **nodes** where both “Janak" and “Mithila" co-occur.

```python
# Entity-aware retrieval
query_entities = ["Janak", "Mithila"]
matching_nodes = []

for node_text, node_data in G.nodes(data=True):
    entities_in_node = node_data.get("entities", [])

    if all(qe in entities_in_node for qe in query_entities):
            matching_nodes.append(node_text)

for match in matching_nodes:
    print("🔍 Matched Node:\n", match)
```

LlamaIndex - Memory Integration

Memory RAG combines:

- **Retrieval-Augmented Generation (RAG)** for pulling external knowledge
- **Memory** to retain **chat history** or **summaries** between user queries

This enables **context-aware**, **multi-turn** conversations where the AI remembers past exchanges and builds intelligent responses — not just in isolation, but as a running dialogue.

```
from llama_index.core import SimpleDirectoryReader, VectorStoreIndex
from llama_index.core.memory import ChatMemoryBuffer
from llama_index.core.chat_engine import SimpleChatEngine

# Load Ramayana documents
index = VectorStoreIndex.from_documents(pdf_document)

# Create memory buffer
memory = ChatMemoryBuffer.from_defaults(token_limit=1000)

# ✅ Use SimpleChatEngine instead of RetrieverQueryEngine
chat_engine = SimpleChatEngine.from_defaults(
    retriever=index.as_retriever(),
    memory=memory
)
```

```
# Simulate multi-turn conversation
print("User: Who is Rama?")
response1 = chat_engine.chat("Who is Rama?")
print("AI:", response1.response)

print("\nUser: Who is his wife?")
response2 = chat_engine.chat("Who is his wife?")
print("AI:", response2.response)

# ✅ Print stored memory
print("\n🧠 Stored Memory:")
for i, msg in enumerate(memory.get(), 1):
    role = msg.role.value.capitalize()
    content = msg.blocks[0].text if msg.blocks else "(No text)"
    print(f"Turn {i}: {role}: {content}")
```

LlamaIndex –Best Practices

Choosing Right Chunk Size and Overlap

Improper chunk sizes can either lead to loss of context (too small) or irrelevant information (too large). Overlap ensures continuity.

Best practice:

- For general documents: chunk_size = 512, chunk_overlap = 64
- For technical or narrative-heavy docs: slightly larger chunks with overlap

Embedding Caching

Avoids re-computing embeddings every time the index is built—saves time and cost.

Best practice:

- Use local or Redis-based cache
- Store and reuse embeddings across sessions

LlamaIndex –Best Practices

Index Persistence

Regenerating the index every time is inefficient in production. Persist it to disk and reload when needed.

Best practice:

- Save after building
- Reload on app start

Latency Optimization

Production systems require low latency. Optimize by:

- Using local embedding models (e.g., via Ollama)
- Querying smaller sets of documents
- Precomputing responses for frequent queries

Best practice:

- Tune top_k for retriever
- Use summarization or compression for long docs

LlamaIndex –Best Practices

PII Redaction and Access Control (RBAC)

- Security and privacy compliance is critical, especially with sensitive documents.

🔐 PII Redaction

- Use regex or NLP-based detection to mask PII.

```
def get_user_docs(user_role):
    if user_role == "finance":
        return load_docs_from_folder("finance_docs/")
    elif user_role == "hr":
        return load_docs_from_folder("hr_docs/")
```

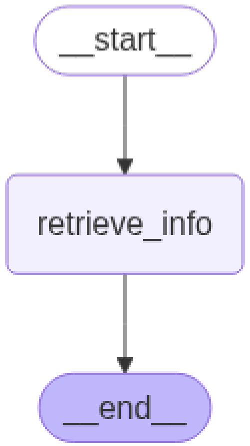
RBAC (Role-Based Access Control)

- Restrict access to certain document sets based on user roles.

Best Practice	Goal	Technique
Chunking Strategy	Context retention	SimpleNodeParser(chunk_size, overlap)
Embedding Caching	Speed, cost efficiency	cache_folder with embedding model
Index Persistence	Scalability	index.save_to_disk() and load_from_disk()
Latency Optimization	Fast responses	similarity_top_k, summarization
PII Redaction & RBAC	Privacy and compliance	regex, user_role filtering

LlamaIndex + LangChain + LangGraph

```python
from llama_index.core import VectorStoreIndex, SimpleDirectoryReader
from langchain_core.runnables import RunnableLambda
from dataclasses import dataclass
from langgraph.graph import StateGraph

# Step 1: Load and index documents
# For demonstration, let's create a dummy index if pdf_document is not available
index = VectorStoreIndex.from_documents(pdf_document, embed_model=embed_model)

# Step 2: Create a retriever or query engine
query_engine = index.as_query_engine()

@dataclass
class QAState:
    question: str
    result: str = "" # Initialize result to an empty string

graph = StateGraph(QAState)
```




```python
def llama_query_node(state: QAState): # Type hint for clarity
    response = query_engine.query(state.question)
    return {"result": str(response)}

llama_node = RunnableLambda(llama_query_node)

# Add nodes and flow as before
graph.add_node("retrieve_info", llama_node)
graph.set_entry_point("retrieve_info")
graph.set_finish_point("retrieve_info")

retrieval_graph = graph.compile()
output = retrieval_graph.invoke({"question": "Who is Rama's wife?", "result": ""})
print(output["result"])
```

```
Sita
```

CrewAI

https://www.crewai.com/open-source

CrewAI

CrewAI is a Python-based framework for orchestrating multi-agent AI systems. It allows you to define agents, assign tasks, group them into a crew, and run a coordinated process — much like human team workflows.

Key Features:

- Agent collaboration
- Modular task design
- Plug-and-play with LLMs, LangChain, LlamaIndex, etc.
- Prompt-engineered task delegation

CrewAI was publicly released in **early 2024**.

- CrewAI was developed by **Joao Moura**, who is also the founder of **CrewAI** as a project. Joao is a former VP of AI at Salesforce and the creator of the tool as an independent initiative.
- His goal was to make **agent orchestration** more accessible and modular, especially in LLM-based multi-agent systems.

CrewAI

Components

Component	Description
Agent	An AI persona with a role, goal, tools
Task	A prompt-defined responsibility for an agent
Crew	A group of agents that coordinate to solve a problem
Process	The actual execution pipeline of tasks by agents Collaboration/Sequential

Each agent performs their task one after another, where the **output of one** becomes the **input of the next**.

```
result = crew.run(process="sequential")
```

Agents can work in **parallel** or **exchange feedback** with one another to co-create an output.

```
collab_result = collab_crew.run(process="collaborative")
```

Research Agent Crew

Let's build a Crew with two agents using OpenAI LLM to:

- Search for Ramayana-related topics
- Summarize findings

CrewAI

Components

Component
Agent
Task
Crew
Process

```python
from crewai import Agent, Task, Crew
from langchain.chat_models import ChatOpenAI

# Setup LLM
llm = ChatOpenAI(model="gpt-4", temperature=0.7)

# Define Agents
researcher = Agent(
    role="Research Analyst",
    goal="Find references to Sita in Ramayana",
    backstory="Expert in mythology, skilled in extracting key facts",
    llm=llm
)

summarizer = Agent(
    role="Content Summarizer",
    goal="Summarize information clearly for educational use",
    backstory="Writes clear and concise summaries for students",
    llm=llm
)
```

```python
# Define Tasks
task1 = Task(
    description="Research all references to Sita across the Ramayana.",
    expected_output="A detailed list of instances with chapter references.",
    agent=researcher
)

task2 = Task(
    description="Summarize the references to Sita in under 300 words.",
    expected_output="A well-structured educational summary.",
    agent=summarizer
)
```

```python
# Assemble Crew
crew = Crew(
    agents=[researcher, summarizer],
    tasks=[task1, task2],
    verbose=False
)
```

```python
# Run Crew
results = crew.kickoff()

# Print output
print("\n✅ Final Output from Summarizer Agent:\n")
print(results)
```

CrewAI

Motivation for CrewAI in LLM Orchestration

As LLM use cases grow in complexity, you often need:

- Role-based task delegation (e.g., researcher, writer, verifier)
- Sequential or parallel execution of interdependent tasks.
- A clean abstraction for building agent teams rather than monolithic chains.

This is where CrewAI excels—it allows to:

- Create multiple agents, each with a persona, toolset, and task
- Define workflows using Crew and Process logic
- Track conversation memory and outcomes per agent

Limitations of Single-Agent Setups (LangChain Agents)

Challenge	Single Agent Limitation
Collaboration	Cannot share tasks among specialized agents
Role separation	All logic and reasoning in one place
Control flow	No built-in process orchestration
Explainability	Hard to attribute steps to agent roles

Why LangGraph Isn't Enough

LangGraph is good for **stateful workflows** but lacks:

- **Agent abstraction**
- **Persona-driven delegation**
- **Built-in memory per agent**
- **Natural division of labor**

CrewAI –Execution

Real-time Coordination of Agent Outputs.

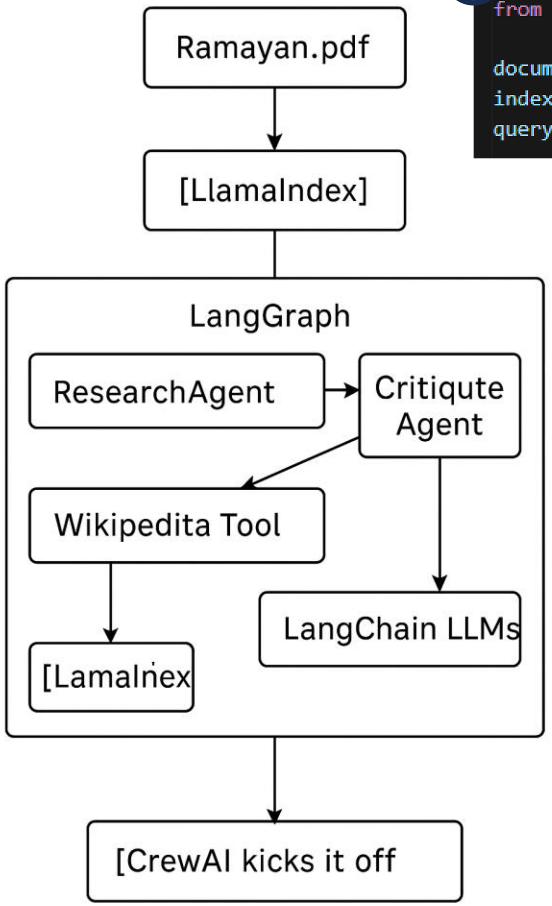
- Agents can share memory, enabling dynamic exchange of information.
- Coordination is handled through Crew that links tasks and agents.
- You can build dynamic task graphs where results are available to other agents immediately.

```
from crewai import Agent, Task, Crew

# Define two simple agents
researcher = Agent(name="Researcher", goal="Research climate change", verbose=True)
summarizer = Agent(name="Summarizer", goal="Summarize the research", verbose=True)

# Define tasks
research_task = Task(agent=researcher, description="Find recent studies on climate change.")
summary_task = Task(agent=summarizer, description="Summarize the findings from the research.")

# Sequential Execution
crew_seq = Crew(tasks=[research_task, summary_task])
crew_seq.kickoff()
```

Intermediate Output Inspection

```
result = crew_seq.kickoff()
print(result.intermediate_steps)
```

Integrating CrewAI with LangChain, and LlamaIndex



1

```python
from llama_index.core import SimpleDirectoryReader, VectorStoreIndex

documents = SimpleDirectoryReader(input_files=["data/RAMAYANA.pdf"]).load_data()
index = VectorStoreIndex.from_documents(documents)
query_engine = index.as_query_engine()
```

2

```python
from langchain.tools import Tool

def ask_ramayan(question: str):
    return query_engine.query(question).response

# Just use a plain dict
ramayan_tool = {
    "name": "AskRamayan",
    "description": "Answer questions from Ramayan knowledge base",
    "function": ask_ramayan  # <- this is your callable
}
```

3

```python
from crewai import Agent
from langchain.chat_models import ChatOpenAI

llm = ChatOpenAI(model="gpt-4")

research_agent = Agent(
    role="Ramayan Researcher",
    goal="Find accurate information from the Ramayan",
    backstory="Knows Ramayan inside out via vector search",
    tools=[ramayan_tool],
    llm=llm
)

summarizer_agent = Agent(
    role="Story Summarizer",
    goal="Summarize events and characters of the Ramayan",
    backstory="Great at concise storytelling",
    tools=[],
    llm=llm
)

critic_agent = Agent(
    role="Philosophical Critic",
    goal="Reflect critically on Ramayan’s messages",
    backstory="Understands cultural context and dharmic philosophy",
    tools=[],
    llm=llm
)
```

4

```python
from crewai import Task, Crew

task1 = Task(
    description="What are the major events in Rama’s life?",
    agent=research_agent
)

task2 = Task(
    description="Summarize Rama’s journey in 5 lines.",
    agent=summarizer_agent
)

task3 = Task(
    description="Provide a philosophical critique of Rama’s decisions during exile.",
    agent=critic_agent
)

crew = Crew(
    agents=[research_agent, summarizer_agent, critic_agent],
    tasks=[task1, task2, task3],
    verbose=True
)
```

```python
result = crew.kickoff()
print(result)
```

AutoGen

https://microsoft.github.io/autogen/stable//index.html

AutoGen

The Evolution of Agentic Systems

- LangChain – Tool Orchestration
  - LangChain introduced a way to **chain LLMs with tools** (e.g., calculators, search APIs) to perform multi-step reasoning.
- LangGraph – Workflow Control
  - Added **branching, statefulness, and looping** to LangChain—allowing you to model workflows like DAGs (Directed Acyclic Graphs).
- CrewAI – Role-Driven Teamwork
  - A system where **multiple agents (LLMs)** play specific **roles (e.g., Researcher, Writer, Critic)** and **collaborate** on a shared task.
- AutoGen – Conversational Multi-Agent Loops
  - A framework where **agents talk to each other in turns**, forming **conversational loops** to solve tasks collaboratively.
- Small Agents – Lightweight, Task-Focused, Composable
  - Inspired by *Unix philosophy*, Small Agents do **one thing well**—lightweight, composable agents you can plug in or swap out.

AutoGen

AutoGen is an **open-source programming framework** designed for building **agentic AI systems**, where multiple language-model agents collaborate—either autonomously or interactively—to complete complex tasks. It supports **multi-agent conversations, tool integration, human-in-the-loop workflows**, and offers no-code options like **AutoGen Studio.**

AutoGen is primarily developed by **Microsoft Research**, in collaboration with academic partners from:

- **Penn State University**
- **University of Washington**
- **Xidian University (China)**

Milestone	Date	Details
Concept Origin	Mar 2023	AutoGen spun off from FLAML (github.com) Paper "AutoGen: Enabling Next-Gen LLM Applications via Multi-Agent Conversation"
v0.2 Release	Aug 16, 2023	Initial open-source framework & academic paper
Ecosystem Buzz	Oct–Dec 2023	Trending on GitHub, recognized in top-100 lists
v0.4 Redesign	Mid-2024	Major rewrite: asynchronous/event-driven engine

Inside AutoGen’s Architecture

UserProxyAgent

Acts as a bridge between human users and the agent system. It receives user input and communicates with other agents on the user’s behalf.

- Receives a query like *“Summarize the Ramayana and suggest themes for reflection.”*
- Passes it to AssistantAgent or GroupChat.

AssistantAgent

Handles complex LLM reasoning tasks—like planning, summarizing, reasoning, or chaining thoughts.

- Breaks the Ramayana summarization task into smaller steps.
- Queries documents, plans response flow.

Inside AutoGen's Architecture

GroupChat

Orchestrates multi-agent collaboration. Agents communicate through this hub, like a roundtable.

- One agent retrieves context
- Another one summarizes
- A third one critiques or verifies

ExecutorAgent

Executes code, queries APIs, or runs tools. This is essential for tool-augmented LLM workflows.

- Executes Python code to analyze Ramayana character frequency.
- Runs scripts or APIs to fetch real-time data.

Agent Collaboration Flow Example

"Analyze Rama's life and generate a visual timeline."

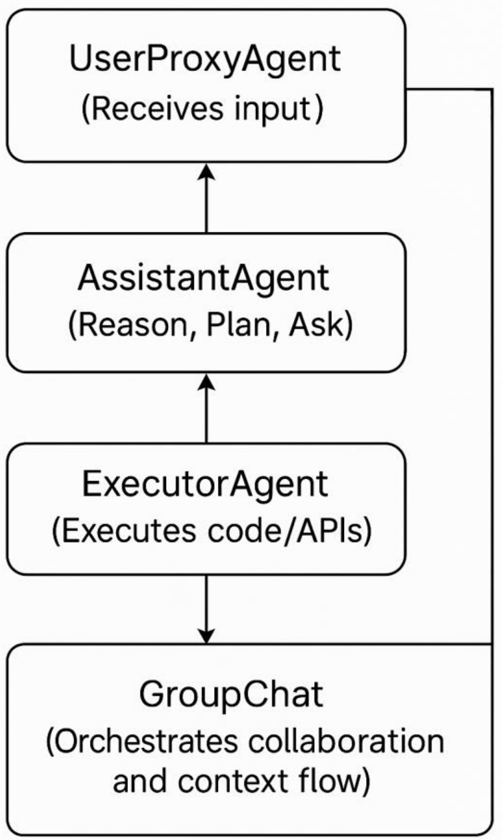
UserProxyAgent receives query.

AssistantAgent breaks down tasks:

- Fetch events from Ramayana
- Analyze sentiment or theme
- Ask ExecutorAgent to generate a timeline plot

ExecutorAgent runs the timeline-generation code.

GroupChat manages the flow of message passing between agents.



Agent Collaboration Flow Example

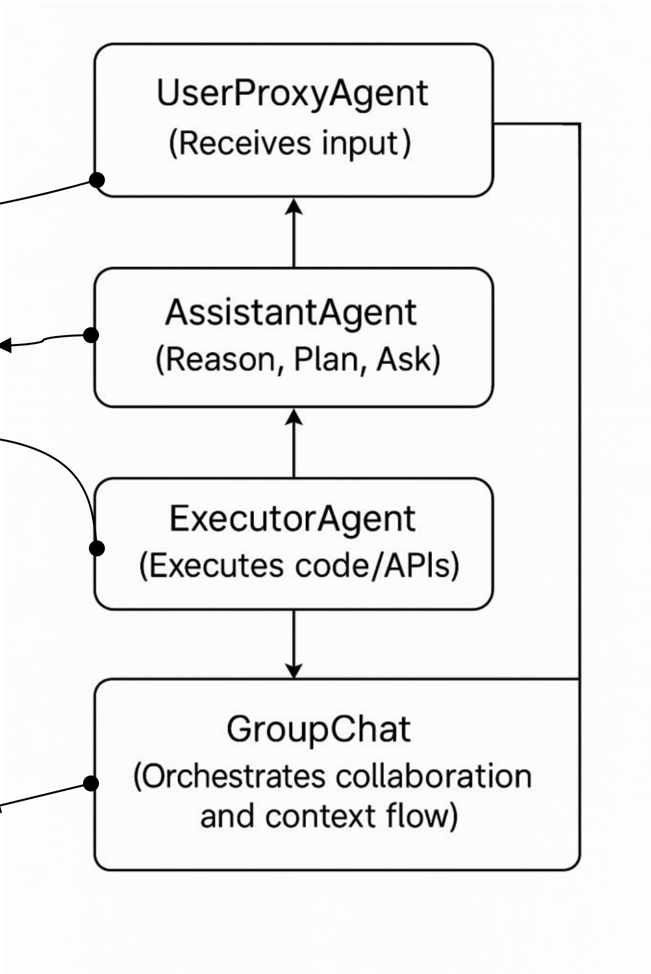
```python
# Load Ramayan context
ramayan_text = """
Rama, the prince of Ayodhya, is born to King Dasharatha and Queen Kaushalya.
He is sent into exile for 14 years, during which his wife Sita is abducted by the demon king Ravana.
Rama, with help from Hanuman and the monkey army, wages war against Ravana, defeats him, and returns to Ayodhya.
"""
```

```python
# User Agent
user = UserProxyAgent(name="user", human_input_mode="NEVER")
```

```python
# Assistant Agent
planner = AssistantAgent(name="planner", llm_config=llm_config)
```

```python
# Executor Agent
code_agent = AssistantAgent(
    name="code_agent",
    llm_config=llm_config,
    code_execution_config={"work_dir": "workspace", "use_docker": False}
)
```

```python
# GroupChat and Manager
groupchat = GroupChat(
    agents=[user, planner, code_agent],
    messages=[],
    max_round=5
)

group_manager = GroupChatManager(
    groupchat=groupchat,
    name="manager",
    llm_config=llm_config  # 👈 REQUIRED to resolve your error
)

# Start Interaction
user.initiate_chat(group_manager)
```



Agent to Agent Protocol

Agent to Agent Protocol

An **Agent-to-Agent Protocol** defines the **rules, format, and structure** by which **two or more AI agents communicate, coordinate, and collaborate**. It governs:

- **Message structure** (who says what, and in what format)
- **Turn-taking** and role negotiation
- **Task delegation and result sharing**
- **Failure handling and retries**
- **Termination conditions** (e.g., convergence or external stop)

In short: **It's the social contract + communication schema among AI agents.**

Agent to Agent Protocol

Core Components

Component	Description	Example
Agent Identity	Defines the agent's name, role, and permissions	{"name": "Retriever", "role": "ContextFetcher"}
Message Format	Standard structure for passing messages	JSON blob, structured prompt, or function call
Turn Handling	Determines who speaks next and when	Turn-based loops, async queues
Message Intent	What is the purpose of the message?	Ask, respond, critique, confirm, delegate
State Tracking	Maintains context of conversation or task	Memory objects or workflow logs

Agent to Agent Protocol

How It Fits in the Overall Agentic AI Landscape

Layer	Role	Tools/Examples
LLMs	Reasoning & response generation	GPT-4, Claude, Gemini
Memory Layer	Stores interaction history/context	LangChain Memory, Vector DBs
Orchestration Layer	Controls flow, branching, error recovery	LangGraph, CrewAI, AutoGen
Agent Layer	Individual units of capability	LangChain Agents, AutoGen Agents
Agent-to-Agent Protocol	Defines how agents interact and coordinate	AutoGen GroupChat, CrewAI process, ReAct JSON message passing
Interface Layer	UI/API for humans or external systems	Chat UI, Slack bots, APIs

Agent to Agent Protocol

AutoGen (by Microsoft)

- Implements agent protocol through structured message passing in GroupChat.
- Agents **"talk"** to each other in natural language with memory context.
- Protocol supports convergence, retries, and collaborative role switching.

CrewAI

- More task-centric protocol, but defines agent roles and expected outputs.
- Agents collaborate sequentially or in parallel with defined handoffs.

LangGraph

- Allows defining a **protocol implicitly** through edges (transitions) and node logic.
- Each node/agent knows **when and how to pass control**.

Open Agent Protocols (Emerging)

- **OAI plugin-style agents**: API calls as communication
- **JSON-RPC over websockets** or **Agent Messaging Layer (AML)** proposals
- Agent standardization efforts from OSS communities (e.g., LangChainHub)

Agent to Agent Protocol

Why it matters

Benefit	Impact
🔄 Interoperability	Enables agents from different vendors or platforms to work together
📡 Structured Coordination	Ensures agents don't talk over each other or loop indefinitely
🔍 Observability	Makes agent behavior auditable and traceable
🤖 Multi-agent Scalability	Supports scaling to 5, 10, or 50 agents with clear interaction rules
🧠 Aligns with Human Workflows	Mirrors organizational structures like teams, committees, or departments

Where You Can Use It

- **RAG QA Systems**: Retriever agent ↔ Generator agent ↔ Critique agent
- **Research Assistants**: Planner ↔ Writer ↔ Fact-checker
- **Customer Support Bots**: Greeter ↔ Troubleshooter ↔ Escalator
- **Simulation/Training**: Role-playing bots with defined personas

Thanks for your time