RAG (Retrieval Augmented Generation)

Understanding Advanced AI Architectures

Gen AI Academy

Easy AI Labs — Yash Kavaiya Ishita Koradia — Gen AI Guru

November 9, 2025

RAG Architecture

Master LLM Enhancement Techniques

Source Material

Retrieval Augmented Generation — What is RAG — How does RAG Work — RAG Explained

CampusX by Nitish Singh

YouTube Series — Comprehensive Langchain Tutorial

Compiled and Presented by

Gen AI Guru — Easy AI Labs

Acknowledgments & Source

Original Content:

This comprehensive document is based on the educational video series:

"Retrieval Augmented Generation — What is RAG — How does RAG Work — RAG Explained"

Created by:

CampusX
Instructor: **Nitish Singh**

Platform: YouTube
Channel: CampusX
Topic: RAG (Retrieval Augmented Generation)

Document Compilation:

Compiled and formatted by **Gen AI Guru**
Easy AI Labs — Ishita Koradia Yash Kavaiya

This document aims to provide a comprehensive written reference for the excellent video tutorial series by CampusX. All credit for the original content, concepts, and teaching methodology goes to Nitish Singh and the CampusX team.

Note: This is an educational reference document.
Please visit the original CampusX channel for complete video lessons.

Contents

1. Overview & Introduction

Topic Overview

Topic: Retrieval Augmented Generation (RAG)
Source: CampusX by Nitish Singh
Format: Two-part video series (Theory + Practical Implementation)
Focus: Theoretical background, conceptual understanding, historical perspective

1.1. Course Structure (Previous 4 Videos)

1. **Document Loaders** – Loading data from any data source
2. **Text Splitters** – Dividing large text into chunks
3. **Vector Stores** – Converting and storing text as embeddings
4. **Retrievers** – Performing semantic search in vector stores

1.2. RAG Core Definition

RAG Definition

RAG is a technique that combines **information retrieval** with **language generation**, allowing models to use retrieved documents as context to generate **grounded, accurate responses**.

1.2.1. Key Components

- **RAG Components:** Document Loaders, Text Splitters, Vector Databases, Retrieval
- **LangChain Components:** Models, Chain, Prompts, Runnables

2. Why RAG? Understanding the Problem

2.1. Understanding LLMs First

What are LLMs:

- Giant transformer-based neural network architectures
- Numerous parameters (weights and biases)
- Pre-trained on huge amounts of data (internet-scale)

2.1.1. Parametric Knowledge

- **Definition:** All knowledge is stored in the model's parameters (weights and biases)
- **Parameter Scale:** More parameters = More powerful model
  - 7B parameters → 13B parameters → 70B parameters (increasing power)

2.2. How LLMs Work (Standard Flow)

1. User sends a query (prompt) to LLM
2. LLM understands the prompt
3. LLM accesses its parametric knowledge
4. LLM generates word-by-word correct answer

2.3. Three Major Problems with Standard LLM Flow

Critical Limitations

Standard LLM prompting fails in three specific scenarios that require alternative approaches.

2.3.1. Problem 1: Private Data

Issue: LLMs cannot answer questions about your private data

Reason: During pre-training, the LLM never accessed your private data

Example: Asking ChatGPT about specific videos on your website (learn.campusx.in)

Result: Direct prompting doesn't work for private data

2.3.2. Problem 2: Knowledge Cutoff Date

Issue: LLMs have a knowledge cutoff date

Example: Model last pre-trained on Jan 1st won't know today's news

Limitation: Cannot answer questions about recent/current events

Note: ChatGPT works because it has internet search access

Result: Open-source models downloaded from Hugging Face won't answer recent questions

2.3.3. Problem 3: Hallucination

Issue: LLMs sometimes provide factually incorrect information with high confidence

Example: Model confidently stating "Einstein played football for Germany in his early years" (completely false)

Reason: LLMs work probabilistically, may imagine facts instead of providing correct information

Result: Users don't get correct answers to their questions

3. Solution 1: Fine-Tuning

3.1. What is Fine-Tuning?

Fine-Tuning Definition

Taking a pre-trained LLM and re-training it on a smaller, domain-specific dataset

Goal: Give LLM general knowledge + domain-specific knowledge

3.2. Analogy: Engineering Student

LLM	≡	Engineering student
Pre-training	≡	Engineering degree education
Fine-tuning	≡	2-3 months company training after joining

3.3. Types of Fine-Tuning

3.3.1. 1. Supervised Fine-Tuning (SFT)

- Provide labeled dataset in format: `prompt` → `desired output`
- Contains 1,000 to 1,000,000 Q&A pairs
- Model trains on these labeled examples

```
# Example training data format
{
    "prompt": "What is gradient descent?",
    "desired_output": "Gradient descent is an optimization algorithm..."
}
```

Listing 1: SFT Data Format

3.3.2. 2. Continued Pre-Training

- Unsupervised method (no labels required)
- Feed raw data (e.g., lecture transcripts) to the model
- Training happens same way as pre-training stage
- **Use case:** Building chatbot for website lectures

3.3.3. 3. Other Techniques

- **RLHF** (Reinforcement Learning from Human Feedback)
- **LoRA** (Low-Rank Adaptation)
- **QLoRA** (Quantized LoRA)

3.4. Fine-Tuning Process (4 Steps)

1. Data Collection

- Collect domain-specific data
- For SFT: Need labeled data (prompts + desired outputs)

2. Method Selection

- Full parameter fine-tuning vs Parameter-efficient methods (LoRA, QLoRA)

3. Training

- Train for few epochs (computationally expensive)
- Full parameter: Retrain all weights
- LoRA/QLoRA: Freeze base weights, train remaining weights

4. Evaluation

- Apply safety tests
- Check exact match, factuality, hallucination rate
- Use various evaluation methods

3.5. How Fine-Tuning Solves the 3 Problems

Problem	Status	Solution
Private Data	Solved	Private data becomes part of parametric knowledge
Recent Data	Partial	Need to re-fine-tune whenever new data arrives
Hallucination	Reduced	Add examples showing model to say "I don't know" for tricky prompts

Table 1: Fine-Tuning Effectiveness

3.6. Major Problems with Fine-Tuning

Fine-Tuning Limitations

1. **Computationally Expensive:** Training large models costs money
2. **Technical Expertise Required:** Need proper AI engineers and data scientists
3. **Frequent Updates Problem:** Must re-fine-tune every time data changes
   - Adding new courses → Re-fine-tune
   - Removing old courses → Re-fine-tune
4. **Not Suitable:** For domains with fast-changing information

4. Solution 2: In-Context Learning

4.1. What is In-Context Learning?

In-Context Learning Definition

Core capability of large language models (GPT-3, Claude, Llama) where the model learns to solve a task **purely by seeing examples in the prompt** without updating its weights.

4.2. Key Characteristics

- **No weight updates**
- Learning from examples provided in the prompt
- Task-solving based on demonstrated patterns

4.3. Example: Sentiment Analysis

```
Below are examples of text labeled with their sentiment.
Use the examples to determine the sentiment of the final text.

"I love this phone. It's so smooth." -> Positive
"This app crashes a lot." -> Negative
"The camera is amazing." -> Positive

"I hate the battery life." -> ?
```

Listing 2: Few-Shot Prompting Example

Model learns pattern and answers: Negative

4.4. Other Applications

- Named Entity Recognition (NER)
- Math problem solving
- Domain-specific problems

4.5. Few-Shot Prompting

The technique of providing few examples in the prompt = **Few-shot prompting**

4.6. Emergent Property

Emergent Property

Definition: Behavior/ability that suddenly appears in a system when it reaches certain scale and complexity, even though it was **not explicitly programmed**.

4.6.1. Historical Context

- **GPT-1, GPT-2:** No in-context learning (smaller models)
- **GPT-3** (175B parameters): In-context learning emerged automatically
- **Landmark Paper:** "Language Models are Few-Shot Learners"

4.6.2. Key Insights from Paper

- **Traditional NLP:** Pre-train → Fine-tune (requires 10K-1M labeled examples)
- **Humans:** Can perform new language tasks from just a few examples
- **GPT-3 scale models:** Can learn from examples in prompt and solve tasks

4.6.3. Post-GPT-3 Improvements

- Supervised fine-tuning
- RLHF (Reinforcement Learning from Human Feedback)
- Models 3.5, 4.0+ became very good at in-context learning

Important Note

Not a universal solution - doesn't always give good results for every task

5. RAG (Retrieval Augmented Generation)

RAG: The Complete Solution

Combining Information Retrieval with Text Generation

5.1. Evolution from In-Context Learning

Current Approach	*Evolution*	**RAG Approach**
Few-shot prompting (giving examples)		Send entire **context** needed to solve task

5.2. Practical Example: Educational Website

Scenario

- Website with video lectures (2-3 hour lectures)
- Student has doubt about specific part
- Want chatbot to help solve doubts

5.2.1. Traditional Approach

- Send student's question to LLM
- **Problem:** LLM doesn't have lecture content

5.2.2. RAG Approach

1. Send student's question
2. Send relevant lecture transcript (e.g., minutes 5-25 discussing gradient descent)
3. Transcript acts as **context**
4. Model uses context to solve query

5.3. RAG Definition

Official RAG Definition

*"RAG is a way to make a language model smarter by giving it **extra information** at the time you ask your question."*

5.4. Key Concept

- **Not** sending examples of how to solve task
- Yes sending complete **context** required to solve question

- Context is injected into prompt
- Enhances model's parametric knowledge

5.5. RAG Flow

User Query + Context → Prompt → LLM → Response

5.6. Prompt Structure

```
You are a helpful assistant.
Answer the question ONLY from the provided context.
If the context is insufficient, just say "I don't know"

Context: [Gradient descent transcript from minutes 5-25]

Question: [Student's doubt about gradient descent]
```

Listing 3: RAG Prompt Template

6. How RAG Works (4-Step Process)

RAG Foundation

RAG = Information Retrieval + Text Generation

- **Information Retrieval:** Old concept from computer science
- **Text Generation:** Famous after LLMs
- RAG is the marriage of these two concepts

6.1. Step 1: INDEXING

INDEXING Phase

Definition: Process of preparing your knowledge base so it can be efficiently searched at query time.

6.1.1. Sub-step 1.1: Document Ingestion

What: Load source knowledge into memory

Examples:

- Video transcripts from server
- Company documents from Google Drive
- Documents from AWS S3

Tools: LangChain document loaders (PyPDF, YouTube, WebBase, etc.)

```python
from langchain.document_loaders import TextLoader, PyPDFLoader

# Text loading
loader = TextLoader(file_path='data.txt', encoding='utf-8')
docs = loader.load()

# PDF loading
pdf_loader = PyPDFLoader('document.pdf')
pdf_docs = pdf_loader.load()
```

Listing 4: Document Loading Example

6.1.2. Sub-step 1.2: Text Chunking

What: Break large document into smaller, semantically meaningful chunks

Why:

1. LLM context length limitations
2. Semantic search quality is poor on large documents

Requirements: Chunks should be meaningful (one chunk = one topic)

Tools:

- Recursive Character Text Splitter (most famous)
- Semantic Chunker
- HTML/Markdown-specific splitters

```
from langchain.text_splitter import CharacterTextSplitter

splitter = CharacterTextSplitter(
    chunk_size=100,
    chunk_overlap=0
)
chunks = splitter.split_documents(docs)
```

Listing 5: Text Chunking Example

6.1.3. Sub-step 1.3: Embedding Generation

What: Convert each chunk into dense vectors that capture semantic meaning

Why: Future semantic search happens between vectors

Process: Text chunk → Embedding Model → Dense Vector

Models: OpenAI Embeddings, Sentence Transformers, etc.

```
from langchain.embeddings import OpenAIEmbeddings

embeddings = OpenAIEmbeddings()
vectors = embeddings.embed_documents([chunk.page_content for chunk in chunks])
```

Listing 6: Embedding Generation

6.1.4. Sub-step 1.4: Vector Storage

What: Store vectors along with original chunk text + metadata in vector database

Local Options: FAISS, Chroma
Cloud Options: Pinecone, Weaviate, Milvus, Qdrant

```
from langchain.vectorstores import Chroma

vector_store = Chroma(
    embedding_function=OpenAIEmbeddings(),
    persist_directory="my_db",
    collection_name="sample"
)
vector_store.add_documents(chunks)
```

Listing 7: Vector Storage Example

Result

External knowledge base ready for searching

6.2. Step 2: RETRIEVAL

RETRIEVAL Phase

Definition: Real-time process of finding the most relevant pieces of information from a pre-built index based on user's question.

6.2.1. Example Scenario

- 2-hour Linear Regression lecture
- User asks about gradient descent
- **Don't** send entire 2-hour transcript
- **Do** search for relevant segments (e.g., minutes 5-25 and 1:43-1:47)

6.2.2. Retriever Component (4 Sub-steps)

Sub-step 2.1: Query Embedding

- Convert user query into embedding vector
- Use **SAME** embedding model used for chunks

Sub-step 2.2: Semantic Search

- Find vectors closest to query vector
- Techniques:
  - Simple semantic search
  - MMR (Maximal Marginal Relevance)
  - Contextual compression

Sub-step 2.3: Ranking

- Rank results by closeness
- Methods:
  - Cosine similarity
  - Advanced re-ranking algorithms

Sub-step 2.4: Fetch Context

- Retrieve text chunks of top results
- This becomes your **context**

```python
# Simple retrieval
retriever = vector_store.as_retriever(search_kwargs={"k": 2})
relevant_docs = retriever.get_relevant_documents(query)

# MMR retrieval
mmr_retriever = vector_store.as_retriever(
    search_type="mmr",
    search_kwargs={"k": 1, "lambda_mult": 1.5}
)
```

Listing 8: Retrieval Example

6.3. Step 3: AUGMENTATION

AUGMENTATION Phase

Definition: Creating a prompt by combining user query and retrieved context.

Query + Context → Augmented Prompt

6.3.1. Example Prompt

```
You are a helpful assistant.
Answer the questions ONLY from the provided context.
If the context is insufficient, just say "I don't know"

Context: [Retrieved relevant chunks]

Question: [User query]
```

Listing 9: Augmented Prompt Structure

Why "Augmentation"?

Adding extra knowledge on top of LLM's parametric knowledge

6.4. Step 4: GENERATION

GENERATION Phase

Definition: LLM uses its text generation capability and in-context learning to answer the query.

6.4.1. Process

1. Augmented prompt sent to LLM
2. LLM combines:
   - Its parametric knowledge
   - Additional context provided
3. LLM generates response

6.4.2. Complete Flow

Query + Context → Augmented Prompt → LLM → Final Response

7. How RAG Solves the 3 Problems

7.1. Problem 1: Private Data

SOLVED

Solution: External knowledge base built from YOUR data

Result: Context derived from your data → Answers based on your data

Verdict: Obviously solved

7.2. Problem 2: Recent Data

SOLVED

Solution: Add recent articles/news to external knowledge base

Advantage over Fine-tuning:

- No need to retrain model
- Simply add new document
- Generate embedding
- Store in vector store

Verdict: Much less costly, easily solved

7.3. Problem 3: Hallucination

GREATLY REDUCED

Solution: Provide exact context for query

Instruction: "Answer ONLY from provided context"

Fallback: "If insufficient information, say 'I don't know'"

Result: Response is **grounded** against context

Verdict: Hallucination chances greatly reduced

8. RAG vs Fine-Tuning Comparison

8.1. Advantages of RAG

1. Cost-Effective (Cheaper)

- No model training required
- No labeled dataset needed
- Simply add documents to vector store

2. Simpler

- Less complex than fine-tuning
- No actual training happening
- Easier to implement and maintain

3. Dynamic Updates

- Easy to add new information
- Easy to remove old information
- No retraining required

4. No Technical Expertise

- Doesn't require AI engineers
- Doesn't require data scientists
- Can be implemented by developers

8.2. When to Use What

accentblue!20 **Use Fine-Tuning When**	**Use RAG When**
Need to change model behavior permanently	Working with private data
Want to specialize model for specific domain	Need recent information
Have large labeled dataset	Want to reduce hallucinations
Updates are infrequent	Data updates frequently
	Cost is a constraint
	Quick implementation needed

Table 2: Decision Matrix: Fine-Tuning vs RAG

9. RAG Architecture Summary

INDEXING PHASE (Done once, updated when knowledge base changes)

1. Document Ingestion → Load documents
2. Text Chunking → Break into chunks
3. Embedding Generation → Convert to vectors
4. Vector Storage → Store in vector database

⇓

RETRIEVAL PHASE (Real-time)

1. Query Embedding → Convert query to vector
2. Semantic Search → Find similar vectors
3. Ranking → Order by relevance
4. Fetch Context → Get original text chunks

⇓

AUGMENTATION PHASE

Combine: User Query + Retrieved Context → Prompt

⇓

GENERATION PHASE

LLM (Parametric Knowledge + Context) → Response

Figure 1: Complete RAG Pipeline Architecture

10. Key Takeaways

1. **RAG Necessity**
   - Solves 3 critical LLM limitations (private data, outdated knowledge, hallucination)
2. **Core Components**
   - Requires understanding of document loaders, text splitters, vector stores, and retrievers
3. **Superior to Fine-tuning**
   - For most use cases involving dynamic data and cost constraints
4. **Four-Step Process**
   - Indexing → Retrieval → Augmentation → Generation
5. **Foundation**
   - Built on information retrieval (old CS concept) + text generation (modern LLM capability)
6. **Production Ready**
   - Basic setup can be enhanced to advanced RAG systems
7. **Next Steps**
   - Practical implementation using LangChain (covered in next video)

11. Important Terms Glossary

Parametric Knowledge
Knowledge stored in model's parameters

In-Context Learning
Learning from examples in prompt without weight updates

Few-Shot Prompting
Providing few examples in prompt

Emergent Property
Capability appearing at scale without explicit programming

Embeddings Dense vector representations capturing semantic meaning

Semantic Search
Search based on meaning, not just keywords

Vector Store Database optimized for storing and searching vectors

Chunking Breaking large text into smaller meaningful pieces

Hallucination LLM generating factually incorrect information

Grounding Constraining LLM responses to provided context

Document Loaders
Components to load data from various sources into standardized format

Text Splitters Tools to break large text into manageable chunks

Retrievers Components that fetch relevant documents based on query

Augmentation Process of enriching prompts with additional context

12. Document Loaders (Detailed)

Document Loaders Overview

Definition: Components in LangChain used to **load data from various sources** into a standardized format (Usually as a Document object).

12.1. Document Object Structure

```
Document {
    page_content = "This is text",
    metadata = {"source": "filename.pdf", ...}
}
```

Listing 10: Document Object

12.2. Common Loaders

12.2.1. TextLoader

For `.txt` files.

```
Loader = TextLoader(file_path.txt, encoding = 'utf-8')
docs = Loader.load()
```

12.2.2. PyPDFLoader

Loads the content of each page into a document object.

12.2.3. WebBaseLoader

Extracts content from webpages.
Uses Python libraries: **Request** (To request webpage) and **BeautifulSoup** (To understand web page structure) for static web pages.

12.2.4. CSVLoader

Extracts CSV data.

12.2.5. DirectoryLoader

For loading all files from a folder.

```
loader = DirectoryLoader(
    path = "book",
    glob = "*.pdf",
    loader_cls = PyPDFLoader
)
```

Use Case	Loader
Simple PDF	PyPDFLoader
PDF with tables/columns	PDFPlumberLoader
Scanned Images	UnstructuredPDFLoader
Need layout & data	PyMuPDFLoader
Best structure extraction	UnstructuredPDFLoader

Table 3: PDF Loader Selection Guide

12.3. PDF Loader Alternatives

12.4. Loading Methods

- `load()`: Loads all data at once.
- `lazyload()`: Returns a list of document generators. Loads documents one by one, which is suitable for large files (Stream processing).

13. Text Splitters (Detailed)

Text Splitters Overview

Definition: Breaking large chunks of text into small, manageable pieces that an LLM can handle effectively.

13.1. Types of Splitting

13.1.1. 1. Length Based

Based on characters or tokens.

```
from langchain.text_splitter import CharacterTextSplitter

splitter = CharacterTextSplitter(
    chunk_size = 100,
    chunk_overlap = 0
)
```

13.1.2. 2. Text Structure Based

`RecursiveCharacterTextSplitter` (Method used for Text Based)

13.1.3. 3. Document Structure Based

Uses keywords like `class`, `def`, etc. (e.g., for code).

13.1.4. 4. Semantic Structure Based

```
from langchain_experimental import semantictextsplitter
```

14. Vector Stores & Databases (Detailed)

Vector Stores Overview

Vector Stores: A system designed to store and retrieve data represented as **numerical vectors**.

14.1. Key Features

1. Storage
2. Similarity Search
3. Indexing
4. CRUD operations

14.2. Vector Store vs. Vector Database

accentblue!20 **Vector Store (e.g., FAISS)**	**Vector Database (e.g., Qdrant, Weaviate, Pinecone)**
Has storage & retrieval capabilities	Includes advanced features
	Distributed systems Backup ACID Transactions Concurrency Authentication
Vector Database = Vector Store + RDBMS features	

Table 4: Vector Store vs Vector Database

14.3. ChromaDB

An open-source vector database.

Analogy: `ChromaDB Collection` is like an `RDBMS Table`.

Flow: Collection → Doc → Embedding → Metadata

14.3.1. Usage Example

```python
from langchain.vectorstore import Chroma

# Create vector store
Vector_store = Chroma(
    embedding_func = OpenAIEmbeddings(),
    persist_directory = "my_db",
    collection_name = "sample"
)

# Add documents
Vector_store.add_documents(docs)

# Search
Vector_store.similarity_search(query = " ", filter = {}, k=2)
```

15. Retrievers (Detailed)

Retrievers Overview

Definition: A component in LangChain that **fetches relevant documents** from a data source based on a user query. All retrievers are **Runnables**.

15.1. Types of Retrievers

15.1.1. 1. Data Source Based

- **Wikipedia Retrievers**: Fetch data from Wikipedia API
- **Vector store based Retrievers**
- **Arxiv Retrievers**

15.1.2. 2. Search Strategy Based

- **Contextual Compression Retrievers**
- **MMR (Maximum Margin Relevance Retrievers)**
- **Multi-Query Retriever**

15.2. Specific Retriever Details

15.2.1. VectorStore Retrievers

Based on semantic similarity.

```
Vectorstore.as_retriever(search_kwargs = {"k": 2})
```

15.2.2. MMR (Maximum Margin Relevance)

For relevant & diverse results.

```
Vectorstore.as_retriever(
    search_type = "mmr",
    search_kwargs = {"k": 1, "lambda_mult": 1.5}
)
# Note: lambda_mult 0 -> creative, 1 -> normal
```

15.2.3. Multi-Query Retrievers

Flow:

[Query] → [LLM] → (Generate diverse & related queries: q1, q2, q3...) → [Retriever] → (Get results for all queries)

15.2.4. Contextual Compression Retrievers

Compresses documents *after* retrieval. **Compressors** will only keep the relevant parts.

16. Fine-Tuning Concepts (Detailed)

16.1. LORA vs. RAG Tuning

LORA (Low-Rank Adaptation)	**RAG Tuning**
Supervised Tuning	Unsupervised Tuning
Uses Labelled data	Uses Unlabelled data
[Prompt → Desired Output]	[Context-based retrieval]

Table 5: LORA vs RAG Tuning

16.2. Fine-Tuning Process

1. **Collect Data**
2. **Choose Model**
   - Full Parameter Fine Tuning
   - LORA / QLORA
3. **Train**
   - Train for few epochs (e.g., 4 new epochs)
   - Keep base weights frozen or partially frozen
   - Update a small subset of weights
4. **Evaluate**
   - Measure exact match
   - Measure factual consistency
   - Test against safety set (check for hallucination)

16.3. RAG for Hallucination Reduction

RAG can be used to ground models in specific data to reduce hallucinations.

- **[RAG] → Private Data [Company Data]**
- **[RAG] → Recent Data**

Important Note

This is the theoretical foundation. The basic setup can be improved with advanced techniques for production RAG systems.

Thank You! Stay Connected

Follow for More AI & Neo4j Updates!

Special thanks to CampusX by Nitish Singh for the original educational content

Conclusion

Summary

RAG (Retrieval Augmented Generation) represents a paradigm shift in how we enhance Large Language Models. By combining the power of information retrieval with advanced text generation capabilities, RAG provides a cost-effective, scalable, and maintainable solution to three critical challenges:

1. Accessing private and proprietary data
2. Incorporating recent and up-to-date information
3. Reducing hallucinations through grounded responses

The four-step RAG pipeline (Indexing, Retrieval, Augmentation, Generation) offers a robust framework that can be adapted and enhanced for various production use cases, making it superior to traditional fine-tuning for most practical applications.

Credits & Attribution

Original Content Source:
CampusX by Nitish Singh

Video Series: "Retrieval Augmented Generation — What is RAG — How does RAG Work — RAG Explained"

Document Compilation:
Gen AI Guru — Easy AI Labs
Ishita Koradia

This document serves as a comprehensive reference guide compiled from the excellent educational content provided by CampusX. All original concepts, teaching methodology, and content credit belongs to Nitish Singh and the CampusX team.

For the complete video tutorials, please visit:

CampusX YouTube Channel
by Nitish Singh