LEARNING OBJECTIVES

By the end of this course, learners will be able to:

1. Explain what Retrieval Augmented Generation (RAG) is and why it's valuable
2. Set up a local large language model using Ollama
3. Configure RAG implementation using Python programming
4. Apply RAG solutions to improve LLM accuracy for domain-specific tasks

SECTION 1: ATTENTION-GRABBING OPENER *(30 seconds)*

[Visual: Split screen showing ChatGPT giving incorrect legal advice vs. accurate RAG enhanced response]

Narrator: "What if I told you that the AI you're relying on for critical work might be giving you outdated or completely wrong information? And what if you could fix this problem yourself, without waiting for big tech companies to solve it for you? Today, you'll learn how to configure a large language model with your own data using Retrieval Augmented Generation"

SECTION 2: COURSE OVERVIEW *(60-90 seconds)*

[Visual: Clean title slide with course outline]

Narrator: "Welcome to Local Retrieval Augmented Generation in Python. I'm Alexander Harris, and in the next few minutes,

- you'll learn the core concepts of retrieval augmented generation (also known as RAG),
- you’ll discover why you should learn retrieval augmented generation,
- The strengths and weaknesses of retrieval augmented generation,
- we’ll go over a real world practical example of RAG,
- and finally we’ll look at a demonstration of RAG in Python programming language

Prerequisites: You should have basic Python knowledge, command line familiarity, understanding of client-server interactions, and basic knowledge of large language models."

SECTION 3: CONCEPT EXPLANATION *(2-3 minutes)*

[Visual: Simple diagram showing traditional LLM vs RAG-enhanced LLM]

Narrator: "So what exactly is Local Retrieval Augmented Generation? We can begin to understand Local RAG through definition

First the idea of "local" is hosting a resource on hardware that is nearby, which de-necessitates having to send requests over a network to a remote resource.

retrieval finds the most relevant information from a knowledge base that's based on a user's query. A query encoding step converts the user's input into a numerical representation also known as a vector. This vector is used to search similar vectors within the knowledge base, effectively retrieving relevant documents.

Next, augmentation integrates the retrieved information into the context of a large language model. This allows the LLM to take into consideration the provided documents and allows for access to specific, up-to--date information that's beyond its pre-trained knowledge.

Lastly the generation will produce a coherent and informative response based on the augmented context. The LLM, now equipped with the retrieved information, generates a response that is more accurate, detailed, and relevant to the user's query.

[Visual: Callout boxes highlighting benefits]

Why use RAG? Four critical advantages:

Security : with a local implementation, you won't have to send your data over the network

- **Accuracy** : Your AI now references YOUR authoritative sources
- **Currency** : Information stays up-to-date as you add new

  documents

Control : You decide what knowledge the LLM will preference

[RAG is great, but it’s not perfect]

Retrieval augmented generation is great, but it’s not perfect. While we discussed the main advantages on the previous slide, I would be remiss without mentioning some of the limitations of retrieval augmented generation.

The first limitation that I’ll mention is RAG’s inherent reliance on the quality and relevance of the knowledge base. You’ll want to ensure that the knowledge base that’s provided to the LLM is high quality because it will directly affect the outcome of a given prompt.

It has the potential for retrieval failures, where the LLM will not directly associate the user’s prompt with the knowledge base provided.

Lastly, the computational costs of running large language models can be great. You’ll want to ensure that your hardware can handle the models you are interested in.

Use Cases
Real-world applications include medical documentation, technical support, enterprise knowledge management, or legal research… for example… "

SECTION 4: LINUS SCENARIO *(1-2 minutes)*

"...Meet Linus, a software developer on a legal research team. Linus’s team is working on an affordable housing project and Linus needs secure, accurate, and current access to penal codes that are specific to his local jurisdiction.

Linus loves ChatGPT's capabilities, but he's facing two critical problems:

First, security. His legal curiosities are confidential - he cannot send sensitive project information over the internet to remote servers.

Second, accuracy. The LLM’s available online don’t know about recent changes to regulations in his jurisdiction.

[Visual: Split screen showing problems vs solutions]

Traditional solution? Wait for ChatGPT to update, compromise security, or manually research everything.
Linus's solution? Build his own system using Ollama for a localized large language model, Python for the application logic, and retrieval augmented generation to inject his legal queries directly into the LLM's knowledge base.

The result? ChatGPT-level intelligence with bank-level security and laser-focused accuracy on his specific legal domain."

SECTION 5: HANDS-ON DEMO *(3-4 minutes)*

[Visual: Clean code editor with callouts, no cursor pointing]

Narrator: Let's build Linus's solution step by step.First, you'll need to go through the setup process. I've prepared a github repository with detailed setup instructions. Once you have the proper software dependencies installed, we can take a look at our retrieval augmented generation implementation in python

Step 1: Environment Setup [Callout highlighting terminal commands]

Taking a closer look at the code, we can see that first we'll import ollama and chromadb dependencies

Next, we'll go ahead to declare and define the documents that we'll provide as our knowledge base to our large language model

Next, We're going to have to make sure to initialize our database client

In the following for-loop, we'll iterate over the set of documents, for each document we'll generate a vector embedding, and add each document to our database.

Next, we'll declare and define our input prompt that's tailored to Linus's specific scenario

We'll generate a vector embedding for our prompt so that we can query our database with a properly formatted input

Next, we'll submit our vectorized query, to the database containing our documents

Finally, we're submitting our prompt to the large language model and establishing the result of the vectorized query as our context when prompting the LLM.

Here we can see a typical LLM response to our query without RAG. The LLM provides a generic response without citation. Whereas the retrieval augmented generative response is highly specific and provides a local ordinance citation.

Linus can now query his legal database with natural language, maintain complete data privacy, and get responses based on authoritative sources rather than AI generalizations.

SECTION 6: CONCLUSION *(60 seconds)*

[Visual: Summary slide with key takeaways]

Narrator: "With that, I believe Congratulations are in order! You've just learned how to transform any large language model into a domain-specific expert using Localized Retrieval Augmented Generation.

Key takeaways:

Throughout this course you learned the core concepts of Local Retrieval augmented generation

You learned why you should use Retrieval augmented generation

Although RAG can be extremely useful, you've also learned some of it's limitations

We discussed real world use cases of this type of technology

And finally, we went over a python implementation of RAG

I hope you've enjoyed this course on local retrieval augmented generation. Hopefully you'll look for opportunities to utilize retrieval augmented generation when considering solutions for your next problem-set.