An Introduction to RAG Applications

Building Conversational AI for Domain-specific Search

Justin Castilla

elastic | The Search AI Company

Justin Castilla

Developer Advocate @ Elastic

justin.castilla@elastic.co

Agenda

This lesson will have the majority of the text here, and the repository will house the code.

01-Introduction

Find out what we will learn today, ensure our environment is ready to develop, and understand what we will accomplish.

02-Motivation

Learn about the underlying technology that brought us to RAG technology and the responsibility of each component.

03-Foundation

Learn how to create an index in Elasticsearch and explore examples of full-text and vector searches

04-Connection

Send the results of Elasticsearch queries to an LLM as context to receive a Natural Language interpretation.

05-Context

Work on the response by exploring prompt engineering and fine-tuning our Natural Language requests around the context sent

06-Conversation

Connect the code to a Gradio UI library to have a conversation with our RAG application

Introduction - What you will learn

Today is a big day!

🔍 Foundations of Retrieval-Augmented Generation (RAG)

- Understand the limitations of standalone Large Language Models (LLMs) and how RAG addresses them.
- Learn the core components of a RAG system: document ingestion, embedding, retrieval, and generation.

🛠️ Hands-On Implementation

- Set up and configure Elasticsearch for vector storage and retrieval.
- Utilize Elastic's built-in E5 model for embedding documents.
- Integrate OpenAI's GPT models to generate responses based on retrieved documents.
- Build an interactive frontend using Gradio to interface with your RAG system.

📄 Working with Real-World Data

- Ingest and process Washington State regulation documents.
- Index documents into Elasticsearch for efficient search and retrieval.

🧪 Experimentation and Evaluation

- Compare different retrieval methods, including BM25 and vector-based approaches.
- Understand the impact of embedding models and chunking strategies on retrieval performance.
- Evaluate the effectiveness of your RAG system using real queries.

🚀 By the End of This Workshop, You Will:

- Have a functional RAG application tailored to Washington State regulations.
- Gain practical experience with Elasticsearch, OpenAI, and Gradio.
- Be equipped to adapt and extend RAG systems for various domains and datasets.

Introduction - What you will need

Docker Hub

- You will need Docker Desktop or a similar program downloaded and installed on your computer.
- You will be running Elastic Start Local, a local docker container of the most recent Elastic instance.

Code Editor

- We will be manipulating and executing code. I recommend using your favorite IDE. I use Visual Studio Code
- We will be using Python 3.12

Patience

- Everyone will have a different setup configuration on their computer. There may be some interesting quirks during development and setup. Email me beforehand if you have any questions.
- I have tried to make every effort to reduce the amount of internet bandwidth necessary to download and install everything needed to run a RAG application. Some slow internet may happen, though. 🐢

Introduction - Run elastic-start-local

Let’s Do: Start our local elastic docker container

Refer to 00-Introduction.md for instructions

Motivation - Full-Text Search (BM25 & TF/IDF)

TF/IDF

TF/IDF is the statistical measure of the frequency and importance of a query term based on how often it appears in an individual document and its rate of occurrence within an entire index of documents.

- **Term Frequency (TF):** This is a measure of how often a term occurs within a document. A higher occurrence of the term within a document, the higher the likelihood that the document will be relevant to the original query. This is measured as a raw count or normalized count based on the occurrence of the term divided by the total number of terms in the document
- I**nverse Document Frequency (IDF):** This is a measure of how important a query term is based on the overall frequency of use over all documents in an index. A term that occurs across more documents is considered less important and informative as is thus given less weight in the scoring. This is calculated as the logarithm of the total documents divided by the number of documents that contain the query term.

While still considered a powerful search algorithm in its own right, **TF/IDF** fails to prevent search bias on longer documents that may have a proportionately larger amount of term occurrences compared to a smaller document.

Motivation - Full-Text Search (BM25 & TF/IDF)

BM25

BM25 (Best Match 25) is an enhanced version of **TF/IDF** components with consideration of document length to create a score outlining the relevance of a document over others within an index.

- **Term Frequency Saturation:** It has been noted that at a certain point, having a high occurrence of a query term in a document does not significantly increase its relevance compared to other documents with a similarly high count. BM25 introduces a saturation parameter for term frequency, which reduces the impact of the term frequency logarithmically as the count increases. This prevents very large documents with high term occurrences from disproportionately affecting the relevance score, ensuring that all scores level off after a certain point.
- **Average Document Length:** This is a consideration of the actual length of the document. It's understood that longer documents may naturally have more occurrences of a term compared to shorter documents, which doesn't necessarily mean they are more relevant. This adjustment compensates for document length to avoid a bias towards longer documents simply due to their higher term frequencies.

BM25 is an excellent tool for efficient search and retrieval of exact-matches with text. It should be noted that **BM25** cannot provide semantic search as with vectors, which provides an understanding of the context of the query, rather than pure keyword search. Knowing when to choose traditional keyword search over semantic search is crucial to ensure efficient use of computational resources.

Motivation - Vectors

What is a vector?

A vector is a representation of data information projected into the mathematical realm as an array of numbers. With numbers instead of words, comparisons are very efficient for computers and thus offer a considerable performance boost. Nearly every conceivable data type (text, images, audio, video, etc.) used in computing may be converted into vector representations.

Images are broken down to the pixel and visual patterns such as textures, gradients, corners, and transparencies are captured into numeric representations. Words, words in specific phrases, and entire sentences are also analyzed, assigned various sentiment, contextual, and synonym values and converted to arrays of floating points. It is within these multidimensional matrices where systems are able to discern numeric similarities in certain portions of the vector to find similarly colored inventory in commerce sites, answers to coding questions on Elastic.co, or recognize the voice of a famous actor.

Each data type benefits from a dedicated Vector Embedding Model, which can best identify and store the various characteristics of that particular type. A text embedding model excels at understanding common phrases and nuanced alliteration, while completely failing to recognize the emotions displayed on a posing figure in an image.

Motivation - Vector Embedding Models

What is an Embedding Model?

When converting a vector of data, in this case, text, a model is used. It should be noted that an embedding model is a pre-trained machine-learning instance that converts text (words, phrases, and sentences) into numerical representations. These representations become multidimensional arrays of floats, with each dimension representing a different characteristic of the original text, such as sentiment, context, and syntactics. These various representations allow for comparison with other vectors to find similar documents and text fragments.

Different embedding models have been developed that provide various benefits; some are extremely hardware efficient and can be run with less computational power. Some have a greater “understanding” of the context and content within the index it is storing within and can answer questions, perform text summarization, and lead a threaded conversation with a user.

Motivation - Large Language Models

Why do we need an LLM?

A Large Language Model (LLM) is a type of deep learning model trained on massive amounts of text to understand and generate human-like language. Examples include OpenAI's GPT models and Google's Gemini. These models:

- Use transformer architectures to process text contextually.
- Can answer questions, summarize text, translate languages, and more.
- Are pre-trained on **general** internet data, books, and articles to develop a broad "knowledge base."

Natural Language Understanding and Generation

The LLM interprets user queries and generates coherent, human-like responses based on the retrieved content.

Semantic Reasoning

LLMs can connect dots across different sources, infer meaning, and answer nuanced questions using retrieved chunks.

Filling Gaps and Contextualizing Results

Retrieved documents often contain raw information. The LLM is what turns raw content into a readable, relevant answer tailored to the user's intent.

Foundation - The index

What is an Index?

An Elasticsearch index is a logical namespace that holds a collection of documents, where each document is a collection of fields — which, in turn, are key-value pairs that contain your data.

Elasticsearch indices are not the same as you'd find in a relational database. Think of an Elasticsearch cluster as a database that can contain many indices you can consider as a table, and within each index, you have many documents.

RDBMS ⇒ Databases ⇒ Tables ⇒ Columns/Rows
Elasticsearch ⇒ Clusters ⇒ Indices ⇒ Shards ⇒ Documents with key-value pairs

In relational databases, normalization is often applied to eliminate data redundancy and ensure data consistency. For example, you might have separate tables for customers, products, and orders.

In Elasticsearch, denormalization is a common practice. Instead of splitting data across multiple tables, you store all the relevant information in a single JSON document. An online order document would contain the customer information and the product information, rather than the order document holding foreign keys referring to separate product and customer indices. This allows for faster and more efficient retrieval of data in Elasticsearch during search operations. As a general rule of thumb, storage can be cheaper than compute costs for joining data.

Foundation - The index

Let’s Do: Create an Index

Refer to the repository to create a sample index

Then head to the repository to run a simple search

Foundation - The inference endpoint

What is an Inference?

An Inference Endpoint refers to a RESTful API interface that connects your index to a machine learning model. It is typically one used for inference tasks like classification, question answering, or text embedding. These endpoints let you invoke a trained model (hosted within Elastic or integrated from external services) to run inference on your data, either:

- At index time (when documents are ingested),
- Or search time (when users query and you enrich results with ML output).

For converting text into a vector for storage in an index, we will use our internal E5 machine learning model. This removes unnecessary connections between modules and if it can be done in one service efficiently, it should be done. We will also use the same inference endpoint to convert user queries to vectors for the final vector similarity search

Foundation - The inference endpoint

How does it work?

- **Deploy a model in Elasticsearch:** This can be an existing model (ELSER or E5), a model uploaded into Elastic (e.g., a PyTorch model), or a connector to a third-party service (like Hugging Face or OpenAI).
- **Create an inference endpoint:** This exposes the model as an API route inside Elastic.
- **Use it in a pipeline or query:**
  - In an **ingest pipeline**, to enrich documents as they come in.
  - In a search query, to add inference output dynamically (e.g., reranking results).

Let's do: Deploy an inference endpoint

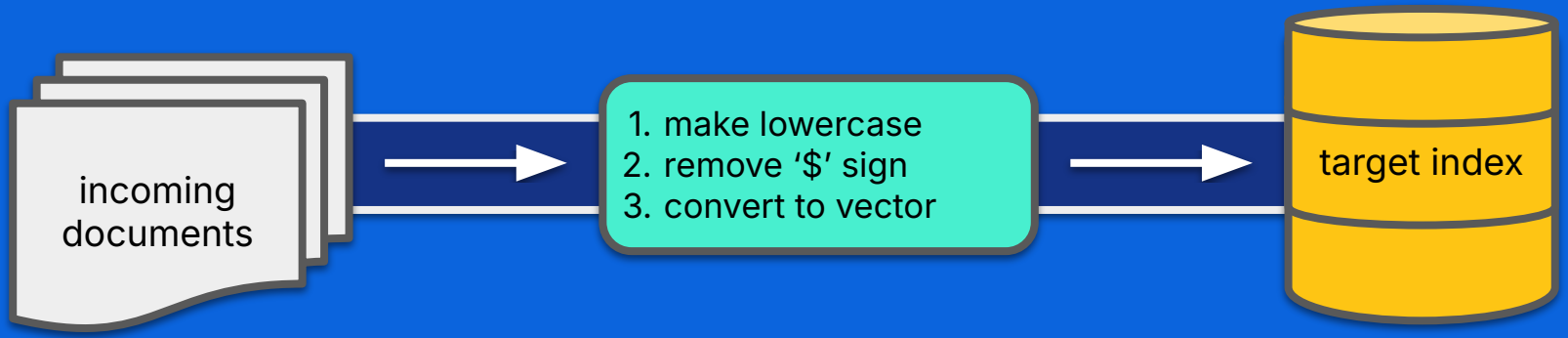
Refer to the repository to deploy an inference endpoint

Foundation - The ingest pipeline

What is an ingest pipeline?

Ingest pipelines let you perform common transformations on your data before indexing. For example, you can use pipelines to remove fields, extract values from text, and enrich your data.

A pipeline consists of a series of configurable tasks called processors. Each processor runs sequentially, making specific changes to incoming documents. After the processors have run, Elasticsearch adds the transformed documents to your data stream or index.



Note: For our RAG application, we will create an ingestion pipeline to convert the pdf text into vectors. This will occur automatically when we index our documents.

Foundation - The ingest pipeline

Let’s do: create an Inference Endpoint and Ingestion Pipeline

First, refer to the repository to create an inference endpoint

Then, refer to the repository to create an ingestion pipeline

Then, refer to the repository to create an ingestion pipeline

Foundation - Creating a Vector

What have we done so far?

Index

Now that we have defined an index with mappings of fields to data types, we can start entering data. Our index dictates a consistent schema of fields to expect from every document. It allows for efficient search

Index → Ingest Pipeline

We have an ingestion pipeline that will intercept that data on its way to our index and convert the text field to a vector embedding. This type of pipeline is called an inference pipeline.

Index → Inference Pipeline → Inference Endpoint

We have deployed an inference endpoint, which is an embedding model that we will use to convert our knowledge base into vectors. We will also convert user queries into vectors to run a similarity search.

Now, the only left to do is verify our instance configuration with a search!

Foundation - You know, for Search

Let’s do: Search within our Index

Refer to the repository to perform a semantic search

Connection - The LLM

Connecting to OpenAI

Now that we are able to search through our index using vector similarity, we can focus on the LLM.

Then we send the user's query along with the raw results (relevant text from the pdfs) to the LLM instance.

The LLM will then interpret the text and respond to the query using natural language, rather than simply returning JSON documents.

Let's do: Connect to our LLM

Refer to the repository to connect to OpenAI's LLM

Then, connect our search results to our LLM for better responses

Context - The LLM

Improve our results

Our results are returning interpreted by the LLM, but improvements can be made to the language, format, and style of the response.

After evaluating the results, let’s explore simple prompt engineering techniques to improve the quality of the responses.

Let’s do: Improve our LLM responses

Refer to the repository to implement prompt engineering

Conversation - Verify our RAG Application

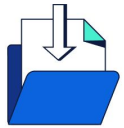
We are almost there!

We have improved our LLM's response to the user by implementing prompt engineering. Now we can implement a simple UI frontend to make conversations happen in real-time. There are many out-of-the-box UI frameworks designed to work seamlessly with your RAG scripts; Gradio was chosen due to its very simple implementation and feather-light footprint on our workshop material.

Let's do: Create our frontend UI

Refer to the repository to create our front end user experience

Review



We created an Elasticsearch Index designed to hold text from pdfs and the converted vectors

We set up an inference pipeline to ingest and convert documents

We set up a local embedding model and connected to OpenAI’s GPT models

We created a conversational UI for our RAG app and can now talk with our data

Tomorrow: Clean your filthy RAG!

We’re crawling. We need to run!

Retrieval-Augmented Generation (RAG) applications are becoming essential for companies, combining AI with real-time data retrieval to enhance customer experiences. While Large Language Models (LLMs) handle general conversation well, they struggle with domain-specific, up-to-date information, often producing inaccurate or unhelpful responses. This workshop will empower participants with the necessary skills to optimize RAG applications using existing best practices. Justin will walk through integrating RAGAS, a framework designed to evaluate, monitor, and fine-tune the performance of RAG applications.

Participants will use benchmarking tools to measure and improve data retrieval performance and explore semantic caching with Elasticsearch to reduce redundant LLM queries, speeding up response times in real-time applications.

Attendees will receive a Docker container with a Python-based RAG application powered by OpenAI's LLM and Elasticsearch for vector storage. Attendees will leave with hands-on experience and a fully functional RAG application they can customize with increased performance and efficiency.

Thank you!