RAG_Local_HF_Weaviate_v3.1 (TicTec Lab Workbook)

Introduction & How to Use This Manual

Welcome to the Local RAG System Project

This manual walks you through creating a **Local Retrieval-Augmented Generation (RAG)** system using:

- **Weaviate** (vector database)
- **Hugging Face** (pre-trained AI models)
- **Python** (the glue that ties everything together)
- **Docker** (coming soon in Step 2)

A Companion to the TicTec Series

We're pairing this manual with the **TicTec** article series, which gives you context, storytelling, and daily how-to guides. The **manual** is your lab workbook—where you'll find the **exact steps** to follow along.

🎧 Full Discography: Discover all tracks and volumes of the Lo-Fi AI series. Visit the hub.

📂 Project Repository: Access prompts, resources, and related materials for Lo-Fi AI projects. View the repo.

Step 1: Setting Up Your Environment

Introduction: Building the Foundation

Welcome to your journey of creating a Local Retrieval-Augmented Generation (RAG) system! In this step, we'll set up your local development environment to handle the tools and code required for the project. By the end of this step, you'll have:

- A working installation of Python.
- Version control with Git and GitHub.

- (Optionally) a virtual environment to manage dependencies cleanly.

This step ensures you're ready to tackle containerization and other advanced concepts in future steps.

1.1 Install & Verify Python

Why Python Matters

Python is the glue of your Local RAG system. It connects your vector database (Weaviate), pre-trained AI models (Hugging Face), and containerized services (Docker).

Installation Steps

1. **Download Python 3.9 or Higher:**
   - Visit python.org/downloads and select the latest stable version (3.9 or higher).
   - During installation on Windows, check the box for "Add Python to PATH."
2. **Verify Your Installation:**
   - Open a terminal or command prompt and type:
     ```
     python --version
     ```
   - You should see output like `Python 3.9.x`.
3. **Test Python Installation:**
   - Create a file named `hello.py` with the following content:
     ```
     print("Hello, TicTec!")
     ```
   - Run the script:
     ```
     python hello.py
     ```
   - If it prints "Hello, TicTec!", you're ready to move on.

1.2 Set Up Git & GitHub

Why Version Control Is Essential

GitHub is your mission control for this project. It tracks code changes, facilitates collaboration, and keeps your files organized. Even for solo projects, version control is a lifesaver.

Steps to Get Started

1. **Install Git:**
   - Download Git for your OS from git-scm.com.
   - Verify installation:

     ```
     git --version
     ```

   - You should see output like `git version 2.42.0`.

2. **Create a GitHub Account:**
   - Go to github.com and sign up if you don't already have an account.

3. **Create a Repository:**
   - Log in to GitHub and create a new repository named `local_rag_project` (or similar).
   - Optionally initialize it with a README file.

4. **Clone or Initialize Locally:**
   - **Option A**: Clone the repository:

     ```
     git clone
     https://github.com/<YOUR-USERNAME>/local_rag_project.git

     cd local_rag_project
     ```

- **Option B**: Initialize locally and connect to GitHub:

```
mkdir local_rag_project

cd local_rag_project

git init

git remote add origin

 https://github.com/<YOUR-USERNAME>/local_rag_project.git
```

5. **Make Your First Commit:**

   Create a simple file (e.g., `readme.md`):

```
echo "TicTec RAG Project" > readme.md

git add .

git commit -m "Initial commit"

git push origin main
```

1.3 (Optional) Create a Virtual Environment

Why Use a Virtual Environment?

Virtual environments keep your dependencies clean and isolated, especially when working on multiple Python projects. This ensures your RAG system remains self-contained and portable.

Steps to Create and Activate a Virtual Environment

1. **Create the Environment:**

```
python -m venv venv
```

2. **Activate It:**
    - **Windows:**
      ```
      venv\Scripts\activate
      ```
    - **Mac/Linux:**
      ```
      source venv/bin/activate
      ```
3. **Document It:**
    - Add activation instructions to your `readme.md` or project documentation.
    - Add `venv` to `.gitignore` to exclude it from version control:

```
venv/
```

Next Steps: A Preview of Docker

Congratulations! You've completed Step 1 of your RAG system setup. Here's what you've accomplished:

- Installed Python and verified it works.
- Set up Git and GitHub for version control.
- (Optionally) created a virtual environment for clean dependency management.

In **Step 2**, we'll introduce Docker, the powerful tool that will containerize services like Weaviate and Elasticsearch. Keep following the TicTec series for context and detailed walkthroughs as we dive into containerization next.

Docker section of RAG_Local_HF_Weaviate_v3 (TicTec Lab Workbook)

Step 2: Introduction to Docker

Picking Up Where We Left Off

Welcome back to your RAG system journey! In Step 1, you set up Python, Git, and GitHub, creating the foundation for your Local RAG system. Now it's time to tackle Docker—the tool that ensures your environment is as consistent as your code. With Docker, we'll eliminate the classic "works on my machine" problem and make running services like Weaviate and Elasticsearch seamless and predictable.

Before diving in, remember that this manual is designed to complement the **TicTec Docker Week articles**, which provide additional insights, storytelling, and practical examples. Refer to them whenever you'd like more context or a narrative perspective.

2.1 Why Docker Matters

The Problem

Imagine sharing your Python application with a teammate or deploying it to a new server, only to encounter missing libraries, version mismatches, or OS-specific quirks. Sound familiar? These environment inconsistencies can grind development to a halt.

The Solution

Docker solves this by packaging your application and its dependencies into portable, isolated containers. These containers run consistently, whether on your laptop, your teammate's machine, or in production.

2.2 Installing Docker

Step-by-Step Installation

Docker installation depends on your operating system. Follow these steps:

For Windows:

1. **Download Docker Desktop**: Visit docker.com and download Docker Desktop for Windows.
2. **Enable WSL 2**: Ensure Windows Subsystem for Linux 2 (WSL 2) is enabled. Follow Microsoft's WSL 2 guide.
3. **Run the Installer**: Launch the Docker Desktop installer and follow the prompts.
4. **Verify Installation**: Open PowerShell and type:

   ```
   docker --version
   docker run hello-world
   ```

You should see Docker’s "Hello, World!" message.

For macOS:

1. **Download Docker Desktop**: Visit docker.com and download Docker Desktop for macOS.
2. **Install and Launch**: Drag Docker into your Applications folder, then open it.
3. **Verify Installation**: Open a terminal and type:

   ```
   docker --version
   docker run hello-world
   ```

For Linux:

1. **Install Docker Engine**

   ```
   curl -o /tmp/get-docker.sh https://get.docker.com
   sh /tmp/get-docker.sh
   sudo usermod -aG docker $USER
   ```

2. **Verify Installation**:

   ```
   docker --version
   docker run hello-world
   ```

2.3 Your First Docker Container

Creating a Simple Dockerfile

Dockerfiles are the blueprint for containers. Let’s create a simple one:

1. Create a directory and a file named `Dockerfile`.
2. Add this content:

   ```
   FROM alpine:latest

   CMD ["echo", "Hello, Docker World!"]
   ```

3. Build the image:

```
docker build -t hello-docker .
```

4. Run the container:

```
docker run hello-docker
```

You should see: `Hello, Docker World!`

2.4 Writing Dockerfiles for Local RAG Systems

Crafting an Effective Dockerfile

For our Local RAG project, we'll use a Dockerfile to containerize a Python app. Here's an example:

```
# Use a lightweight Python image
FROM python:3.9-slim

# Set the working directory
WORKDIR /app

# Copy dependency file and install libraries
COPY requirements.txt .
RUN pip install --no-cache-dir -r requirements.txt

# Copy the app code
COPY . .

# Set environment variables
ENV FLASK_APP=app.py

# Define the default command
CMD ["flask", "run", "--host=0.0.0.0"]
```

Best Practices

- Minimize layers by combining commands.
- Use `.dockerignore` to exclude unnecessary files.
- Avoid hardcoding sensitive information (e.g., API keys).

2.5 Running Elasticsearch in Docker

Getting Started with Elasticsearch

Elasticsearch is critical for RAG systems. Let's containerize it:

1. **Pull the Image**:

```
docker pull docker.elastic.co/elasticsearch/elasticsearch:8.10.2
```

2. **Run the Container**:

```
docker run -d \
  --name es-container \
  -p 9200:9200 \
  -e "discovery.type=single-node" \
  docker.elastic.co/elasticsearch/elasticsearch:8.10.2
```

3. **Verify**:
   - Logs: `docker logs es-container`
   - Browser: Visit `http://localhost:9200`

Next Steps: Docker + Python Integration

You've completed Docker basics and are now ready to integrate containerized components with Python scripts in Step 3. Keep following the TicTec series for more insights and practical examples as we dive deeper into connecting Weaviate, Elasticsearch, and your local RAG system.

Congratulations on mastering Step 2—your RAG system is taking shape!

TicTec Lab Workbook: Week 3 – Python and Beyond

Introduction: The Language of Machines

Python is the glue of modern AI workflows. It connects vector databases, pre-trained models, and containerized services, making it an essential skill for any tech enthusiast. This week, we'll dive into Python basics and progress toward building your first Retrieval-Augmented Generation (RAG) script.

3.1 Python: The Swiss Army Knife of AI

Why Python?

Python is widely used in AI because:

- **Readability**: Its simple syntax makes it beginner-friendly.
- **Libraries**: It has powerful libraries like NumPy, Pandas, and Hugging Face.
- **Community**: An active community ensures extensive documentation and support.

Setting Up Python

Ensure Python is installed on your machine (refer to Step 1.1 from Week 1).

Hands-On Exercise: Your First Python Script

1. Open a terminal or command prompt.
2. Create a file named `hello_tictec.py` with the following content:

```
print("Hello, TicTec!")
```

3. Run the script:

```
python hello_tictec.py
```

Expected output: `Hello, TicTec!`

3.2 Python for the Uninitiated: A Beginner’s Guide

Core Concepts

- **Variables**: Store data for reuse.

```
 name = "TicTec"
print(f"Hello, {name}!")
```

- **Loops**: Perform repetitive tasks.

```
 for i in range(3):
    print("Iteration", i)
```

- **Functions**: Encapsulate reusable code.

```
 def greet(name):
    return f"Hello, {name}!"

print(greet("TicTec"))
```

Hands-On Exercise: Building Blocks

- Write a script that calculates the sum of numbers from 1 to 10.

```
 total = sum(range(1, 11))
print("Sum:", total)
```

- Run it and verify the output: `Sum: 55`

3.3 Introducing Libraries: Tools for AI Wizards

Key Libraries

- **NumPy**: For numerical computations.

```
 import numpy as np
array = np.array([1, 2, 3])
print(array * 2)
```

- **Pandas**: For data manipulation.

```
 import pandas as pd
data = pd.DataFrame({"Name": ["Alice", "Bob"], "Score": [85, 90]})
print(data)
```

- **Hugging Face Transformers**: For AI model interactions.

```
 from transformers import pipeline
summarizer = pipeline("summarization")
print(summarizer("TicTec is a hands-on AI learning journey."))
```

Hands-On Exercise: Exploring Libraries

- Install the libraries:

```
 pip install numpy pandas transformers
```

- Test each example above in a new script.

3.4 Writing Modular Python Code for AI Projects

Best Practices

1. **Modularity**: Break code into functions and modules.
2. **Readability**: Use meaningful variable names and comments.
3. **Reusability**: Write functions that can be applied to different use cases.

Example: Modular Code for Data Summarization

- Create `data_utils.py`:

```
def summarize_text(text):
    from transformers import pipeline
    summarizer = pipeline("summarization")
    return summarizer(text)
```

- Create `main.py`:

```
from data_utils import summarize_text

text = "TicTec is a hands-on AI learning journey."
summary = summarize_text(text)
print(summary)
```

- Run `main.py` to see the output.

3.5 Making Python Work: Writing Your First RAG Script

Objective

Combine Python basics, libraries, and Dockerized Elasticsearch to build a simple RAG workflow.

Steps

- **Set Up Elasticsearch**

Ensure Elasticsearch is running in Docker (refer to Week 2).

- **Install Required Libraries**

```
pip install elasticsearch transformers
```

- **Write the RAG Script**

```
from elasticsearch import Elasticsearch
from transformers import pipeline

# Connect to Elasticsearch
es = Elasticsearch(["http://localhost:9200"])

# Add a document to Elasticsearch
doc = {"content": "TicTec helps you learn AI step by step."}
es.index(index="documents", id=1, document=doc)

# Search Elasticsearch
query = "AI learning"
response = es.search(index="documents", query={"match": {"content":
query}})
retrieved_doc = response['hits']['hits'][0]['_source']['content']

# Summarize the retrieved document
summarizer = pipeline("summarization")
summary = summarizer(retrieved_doc)

print("Summary:", summary)
```

- **Run the Script**

```
python rag_script.py
```

  - Expected Output: A summary of the retrieved document.

Next Steps

Congratulations! You've completed Week 3. By now, you should:

- Understand Python's role in AI workflows.
- Write basic and modular Python scripts.
- Use libraries like NumPy, Pandas, and Hugging Face.
- Build a simple RAG workflow.

Preview of Week 4: We'll dive into vector databases and explore how to integrate them with Python and Hugging Face. Get ready to power up your AI workflows!

TicTec Lab Workbook: Week 4 – Vector Databases and Hugging Face

Theme: "Brains for Your Machine"

Welcome to Week 4 of the TicTec journey! Thus far, you've:

- Set up your local environment with Python and Git (Weeks 1 & 2).
- Explored Docker for consistent deployments (Week 2).
- Learned Python fundamentals and created a simple Retrieval-Augmented Generation (RAG) script (Week 3).

Now, we'll integrate **vector databases** like Weaviate and **Hugging Face** models to power semantic search and text generation. This week is divided into four parts, corresponding to four daily tracks (Tuesday to Friday).

4.1 – The Memory Machine: Setting Up Weaviate

4.1.1 Why Vector Databases?

Traditional search vs. Semantic search

- **Keyword-based search** (like a classic SQL or search engine) matches exact terms or phrases.
- **Semantic search** uses vector embeddings to capture the meaning of text, enabling more relevant results even when keywords don't match exactly.

Why Weaviate?

- **Easy to use:** Minimal setup via Docker.
- **Scalable & flexible:** Weaviate supports a variety of data types and embedding approaches.
- **GraphQL-based queries:** Makes advanced searching intuitive.

4.1.2 Installing and Running Weaviate Locally

Pull the Weaviate image from Docker Hub

```
docker pull semitechnologies/weaviate:latest
```

Run Weaviate in a Docker container

```
docker run -d \
  --name weaviate-container \
  -p 8080:8080 \
  -e QUERY_DEFAULTS_LIMIT=50 \
  semitechnologies/weaviate:latest
```

  - `-d`: Runs the container in detached mode.
  - `--name weaviate-container`: Assigns a container name for easy reference.
  - `-p 8080:8080`: Exposes Weaviate's default port.
  - `-e QUERY_DEFAULTS_LIMIT=50`: Sets a default query limit (optional).

Verify Weaviate is running

Docker logs:

```
docker logs weaviate-container
```

- Look for startup messages indicating successful initialization.
- **Browser**: Visit `http://localhost:8080/v1/.well-known/ready`
  If you see a `{"status":"READY"}`, Weaviate is ready to roll.

4.1.3 Creating a Basic Schema

Weaviate organizes data via **Classes** and **Properties**. Let's create a minimal "Document" class for storing text.

Create a schema.json file (or adapt the sample below):

```
{
  "classes": [
    {
      "class": "Document",
      "properties": [
        {
          "name": "content",
          "dataType": ["text"]
        }
      ]
    }
  ]
}
```

Ingest the schema (using Weaviate's REST API or GraphQL):

```
curl -X POST "http://localhost:8080/v1/schema" \
-H "Content-Type: application/json" \
-d @schema.json
```

Check the schema

```
curl "http://localhost:8080/v1/schema"
```

Verify that the new **Document** class is present.

4.1.4 Testing Data Ingestion

We'll store a sample document in Weaviate to ensure everything works:

```
curl -X POST "http://localhost:8080/v1/objects" \
-H "Content-Type: application/json" \
-d '{
  "class": "Document",
  "properties": {
    "content": "Welcome to TicTec, a hands-on AI learning journey."
  }
}'
```

Expect a **200** or **201** response indicating success.

4.1.5 Next Steps

Now that Weaviate is up and running locally, you're ready to explore **semantic search** on Wednesday.

4.2 – The Art of Retrieval: Semantic Search in Action

4.2.1 Understanding Semantic Search

1. **Embedding Text**
   - Text is converted into vectors (high-dimensional numeric arrays) that capture semantic meaning.
2. **Similarity Metrics**
   - Commonly used metrics include **cosine similarity** and **dot product**.
   - The closer two vectors are in this space, the more semantically related they are.

4.2.2 Generating Embeddings in Weaviate

Weaviate can automatically embed text if configured with a **vectorizer** (e.g., `text2vec-transformers`).

1. **Check if your Docker container supports text2vec** (it may be built-in, depending on the image tag).
2. **Alternatively**, you can generate embeddings externally (e.g., Hugging Face) and send them to Weaviate. We'll explore a combined approach later this week.

4.2.3 Querying Documents via GraphQL

Basic Query

```
curl -X POST "http://localhost:8080/v1/graphql" \
-H "Content-Type: application/json" \
-d '{
  "query": "{
    Get {
      Document(
        limit: 5
        nearText: { concepts: [\"AI learning\"] }
      ) {
        content
        _additional {
          distance
        }
      }
    }
  }"
}'
```

- `nearText: { concepts: [\"AI learning\"] }` instructs Weaviate to find semantically similar documents related to *AI learning*.

Interpreting the Response

- `content`: The matched text.
- `distance`: (or `certainty`) indicates how close the document is to the query concept.

4.2.4 Hands-On Exercise: Semantic Queries

1. **Add Multiple Documents**
   - Ingest at least **3–5** documents referencing different AI topics.
2. **Experiment**

    - Run queries with various **nearText** concepts: `"machine learning"`, `"neural networks"`, `"robotics"`, etc.
    - Observe how Weaviate ranks the documents.

4.2.5 Next Steps

With retrieval in hand, we'll introduce **Hugging Face** text generation on Thursday, setting the stage for a complete RAG pipeline.

4.3 – Text Magic: Introducing Hugging Face Models

4.3.1 Why Hugging Face?

- **Massive model hub**: Access to thousands of pre-trained transformer models.
- **Easy integration**: Python libraries like `transformers` and `sentence-transformers` streamline usage.

4.3.2 Installing and Testing Transformers

Install via pip

```
pip install transformers sentence-transformers
```

Quick Test

```
from transformers import pipeline

classifier = pipeline("sentiment-analysis")
result = classifier("TicTec is a fantastic AI journey!")
print(result)
```

Expect an output showing a sentiment label (`POSITIVE`) and a confidence score.

4.3.3 Working with Embeddings

Sentence-Transformers Example

```
from sentence_transformers import SentenceTransformer
```

```
model = SentenceTransformer('sentence-transformers/all-MiniLM-L6-v2')
embeddings = model.encode(["Hello, Weaviate!", "AI is awesome."])
print(embeddings)
```

- These embeddings can be pushed to Weaviate to store them directly.

Sending Embeddings to Weaviate *(Optional Preview)*

- Convert your text to vectors with Sentence Transformers.
- Use Weaviate's `/v1/objects` endpoint to upload the vector as `vector` property.
- This approach bypasses the built-in text2vec module, giving you finer control.

4.3.4 Generating Text

Text Generation Pipeline

```
generator = pipeline("text-generation", model="gpt2")
output = generator("TicTec is", max_length=30, num_return_sequences=1)
print(output)
```

1.
    - The model completes your prompt.
2. **Summarization, Q&A, etc.**
    - Explore pipelines like `"summarization"`, `"question-answering"`, or `"translation_en_to_fr"`.

4.3.5 Next Steps

You now have the building blocks—semantic retrieval with Weaviate and text generation with Hugging Face. On Friday, we'll **combine** these into a unified **RAG workflow**.

4.4 – Building a RAG Workflow with Weaviate and Hugging Face

4.4.1 RAG Overview

Retrieval-Augmented Generation (RAG) merges:

1. **Retrieval**: Pulling semantically relevant content from a vector database (Weaviate).
2. **Generation**: Passing retrieved context into a model (Hugging Face) to produce an answer, summary, or narrative.

4.4.2 Core Steps

1. **Preprocessing & Embedding**: Convert your documents into embeddings (either via Weaviate's built-in vectorization or external embedding with Sentence Transformers).
2. **Storage**: Store documents (plus optional metadata) in Weaviate.
3. **Query**: Retrieve top-matching documents based on a user question or prompt.
4. **Generate**: Use Hugging Face to incorporate those relevant documents and generate a cohesive result.

4.4.3 Hands-On: Basic RAG Script

Create a file named `rag_weaviate.py`:

```
import requests
from sentence_transformers import SentenceTransformer
from transformers import pipeline

# 1. Initialize Models
embed_model =
SentenceTransformer('sentence-transformers/all-MiniLM-L6-v2')
generator = pipeline("text-generation", model="gpt2")

# 2. User Prompt
user_query = "How does TicTec help me learn AI?"

# 3. Generate Embedding for the user query
query_embedding = embed_model.encode([user_query]).tolist()[0]

# 4. Semantic Search in Weaviate
graphql_query = {
    "query": """{
      Get {
        Document(
          nearVector: {
            vector: """ + str(query_embedding) + """
          }
          limit: 2
        ) {
          content
        }
      }
    }"""
}

response = requests.post("http://localhost:8080/v1/graphql",
json=graphql_query)
data = response.json()
retrieved_docs = data["data"]["Get"]["Document"]

# 5. Combine retrieved docs into a context string
context = " ".join([doc["content"] for doc in retrieved_docs])

# 6. Generate response
prompt = f"Context: {context}\nQuestion: {user_query}\nAnswer:"
generated_text = generator(prompt, max_length=50,
num_return_sequences=1)[0]["generated_text"]

print("=== RAG Output ===")
print(generated_text)
```

Explanation

1. **Embedding the user query**: We create a single query embedding.
2. **GraphQL nearVector**: We pass that embedding to Weaviate, retrieving the top 2 matching "Document" objects.
3. **Prompt construction**: Combine retrieved text as "Context" for the Hugging Face model.
4. **Generation**: Use GPT-2 (or any Hugging Face model) to produce an answer guided by the context.

4.4.4 Testing & Troubleshooting

1. **Weaviate Running**
    - Ensure the container is active: `docker ps`
    - Check logs if no documents are returned.
2. **Models Installed**
    - Make sure `sentence-transformers` and `transformers` are up to date.

3. **Schema or Data Issues**
    - If your GraphQL query returns an error, verify your schema definitions and the data you've stored.

4.4.5 Where to Go Next

- **Optimization**: In Week 5, we'll address performance tuning, Docker Compose orchestration, and potentially cloud deployment.
- **Security & Scalability**: Later, we'll introduce best practices to protect and scale your Local RAG system.

Conclusion: Week 4 Recap

Congratulations! You've now:

- Spun up **Weaviate** and explored vector-based storage.
- Mastered the basics of **semantic search**.
- Experimented with **Hugging Face** for both embeddings and text generation.
- Combined all elements into a **simple RAG pipeline**.

Stay tuned for **Week 5**, where we'll take this local setup to the next level—optimizing your system, introducing Docker Compose, and exploring production-like deployments.