Sure, let's break it down step by step.

## **Overview**

This code creates an **agentic AI system** that:

1. **Reads a CSV file** containing failure reasons and solutions.
2. **Embeds and stores the failure reasons** in a FAISS vector database.
3. **Retrieves relevant solutions** when a user inputs an error message.
4. **Uses an AI model (Llama 3.2 from Groq)** to generate resolution steps.
5. **Returns a structured response** containing:
   * The **error message**
   * The **retrieved or AI-generated solution**
   * **Resolution steps** (retrieved or AI-generated)

## **Step-by-Step Breakdown**

### **1. Importing Necessary Libraries**

from google.colab import userdata

groq\_api = userdata.get("groq\_api\_key")

* Fetches the **Groq API key** stored in Google Colab's user data. This API key is required to use Groq's **Llama 3.2 model**.

import pandas as pd

from langchain\_community.vectorstores import FAISS

from langchain\_community.embeddings import HuggingFaceEmbeddings

from langchain\_core.documents import Document

from langgraph.graph import END, StateGraph

from langchain\_core.prompts import ChatPromptTemplate

from langchain\_groq import ChatGroq

from typing import TypedDict, List

* **pandas**: For handling the CSV file.
* **FAISS**: A vector database for efficient similarity searches.
* **HuggingFaceEmbeddings**: Converts text (failure reasons) into **embeddings**.
* **Document**: Represents each failure reason and its solution as a **document**.
* **StateGraph**: Manages the workflow of our AI agent.
* **ChatPromptTemplate**: Defines how the AI model is prompted.
* **ChatGroq**: Connects to Groq’s AI model for generating responses.

### **2. Reading the CSV File & Preparing Data**

df = pd.read\_csv("/content/failure reason.csv")

* Reads the CSV file containing **failure reasons and solutions**.

docs = []

for \_, row in df.iterrows():

if pd.notna(row["Failure Reason"]) and pd.notna(row["Solution"]):

docs.append(Document(

page\_content=row["Failure Reason"],

metadata={"solution": row["Solution"]}

))

* **Iterates through the CSV rows**.
* **Creates a Document** for each failure reason, storing its **solution in metadata**.
* **Stores the documents in a list** (docs), which will be used for vector search.

### **3. Creating a FAISS Vector Store**

embeddings = HuggingFaceEmbeddings(model\_name="sentence-transformers/all-MiniLM-L6-v2")

vector\_store = FAISS.from\_documents(docs, embeddings)

retriever = vector\_store.as\_retriever(search\_kwargs={"k": 1})

* Uses the **sentence-transformers/all-MiniLM-L6-v2** model to **convert failure reasons into vectors**.
* **FAISS stores these vectors** so that similar failures can be retrieved quickly.
* retriever allows fetching the **most relevant failure reason (k=1)** based on similarity.

### **4. Defining the Agent's State**

class AgentState(TypedDict):

input: str

context: List[dict]

response: str

* Defines the **state structure** used by our AI agent.
* It has:
  + **input** → The error message given by the user.
  + **context** → The retrieved document (if any).
  + **response** → The AI-generated response.

### **5. Connecting to Llama 3.2 via Groq**

llm = ChatGroq(

groq\_api\_key = groq\_api,

temperature=0.3,

model\_name="llama-3.2-1b-preview",

)

* Connects to Groq’s Llama 3.2 model.
* **Temperature = 0.3** → Controls randomness in responses.

### **6. Retrieving the Solution from FAISS**

def retrieve(state: AgentState):

try:

relevant\_docs = retriever.invoke(state["input"])

return {"context": relevant\_docs}

except:

return {"context": []}

* **Searches FAISS** for the most relevant failure reason.
* **Returns the retrieved document** (solution) or an **empty list** if no match is found.

### **7. Generating the Response**

def generate\_response(state: AgentState):

try:

response\_template = """\*\*Error:\*\*\n{Error}\n\n\*\*Solution\*\*\n{Solution}\n\n\*\*Resolution Steps\*\*\n{Resolution}"""

* Defines a **formatted response template**.

#### **Case 1: If the error is known**

if state["context"] and "solution" in state["context"][0].metadata:

context = state["context"][0]

prompt\_template = """

[INST] Given this error and known solution:

Error: {error}

Solution: {solution}

Generate only Resolution Steps (numbered list of 5 steps) [/INST]

"""

result = llm.invoke(ChatPromptTemplate.from\_template(prompt\_template).format(

error=state["input"],

solution=context.metadata["solution"]

)).content

* If the **error is found in FAISS**, the **solution is retrieved**.
* The AI model is **only asked to generate resolution steps**.

resolution = "\n".join([line for line in result.split("\n") if line.strip() and line[0].isdigit() or line.startswith("-")])

return {"response": response\_template.format(

Error=state["input"],

Solution=context.metadata["solution"],

Resolution=resolution

)}

* Parses only **numbered steps** from AI’s response.

#### **Case 2: If the error is unknown**

else:

prompt\_template = """

[INST] As a system administrator, handle this new error:

Error: {error}

Generate:

1. Solution

2. Resolution Steps [/INST]

"""

full\_response = llm.invoke(ChatPromptTemplate.from\_template(prompt\_template).format(

error=state["input"]

)).content

* If the error **is not found**, AI **generates both the solution and steps**.

sections = full\_response.split("\n\n")

return {"response": response\_template.format(

Error=state["input"],

Solution=sections[0].replace("1. Solution: ", "").strip(),

Resolution=sections[1].replace("2. Resolution Steps\n", "").strip()

)}

* Extracts the **solution** and **steps** separately from AI’s response.

### **8. Defining the Agent Workflow**

workflow = StateGraph(AgentState)

workflow.add\_node("retrieve", retrieve)

workflow.add\_node("generate\_response", generate\_response)

workflow.set\_entry\_point("retrieve")

workflow.add\_edge("retrieve", "generate\_response")

workflow.add\_edge("generate\_response", END)

agent = workflow.compile()

* Creates a **state machine** where:
  1. retrieve finds the **solution in FAISS**.
  2. generate\_response **formats and refines** the output.
  3. The final response is **returned**.

### **9. Getting a Solution**

def get\_solution(error\_message):

result = agent.invoke({"input": error\_message.strip()})

return result["response"]

* Calls the **agent** to process the error message.

### **10. Testing with Example Errors**

print("\n=== Known Error ===")

print(get\_solution("API rate limiting not enforced"))

* Tests a **known error** (expected to retrieve a solution from FAISS).

print("\n======\n")

print("=== UnKnown Error ===")

print(get\_solution("Disk Read Error"))

* Tests an **unknown error** (AI must generate the solution & steps).

## **Summary**

* **Finds existing solutions** using FAISS.
* **Generates solutions** if none are found.
* **Formats structured responses**.
* **Uses Llama 3.2 for intelligent troubleshooting**.

Let me know if you need further clarification!

Using **FAISS** instead of direct lookup, fuzzy matching, or dictionary-based search has several advantages, especially when dealing with larger datasets or when you need **semantic search**. Here's how **FAISS** compares to the alternatives:

## **1. Key Benefits of FAISS**

### **1️⃣ Fast & Scalable for Large Datasets**

* FAISS is optimized for **millions of records** using **approximate nearest neighbor (ANN) search**.
* Alternatives like **Pandas or fuzzy matching** slow down significantly as data grows.

✅ **Use Case:** If your dataset grows beyond **1000+ failure reasons**, FAISS remains **fast and efficient**.

### **2️⃣ Handles Similar but Not Identical Queries (Semantic Search)**

* FAISS uses **vector embeddings**, which capture **semantic meaning** rather than just **word matching**.
* Fuzzy matching only works with **string similarity**, while FAISS understands meaning.

✅ **Example:**  
**Error in CSV:** "Database connection timeout due to high traffic."  
**User Input:** "DB timeout issue caused by too many users."

* **FAISS will match correctly (because it understands meaning)**
* **Fuzzy matching might fail (because the words are different)**

### **3️⃣ Handles Synonyms & Contextual Understanding**

* FAISS uses **Hugging Face embeddings**, which recognize **similar words and phrases**.
* Example:
  + "Low memory in server" → **FAISS understands** "Insufficient RAM in system" as similar.
  + Fuzzy matching might **fail** because the wording is different.

✅ **Use Case:** If **failure reasons are written in different ways**, FAISS will still find them.

### **4️⃣ Can Work for Partial Inputs**

* If a user inputs **only part of the error message**, FAISS can still find the closest match.
* Dictionary or exact match approaches will **fail** unless the input is **exact**.

✅ **Example:**  
If your dataset has **"Network request timeout"**, a user might type:

* **"Request timeout issue"** → FAISS will match correctly.
* **Fuzzy match might fail** (depending on similarity score).

### **5️⃣ FAISS is Efficient for Multi-Dimensional Data (Beyond Text)**

* If later you want to **match errors based on multiple factors**, such as **error type, logs, timestamps, and error codes**, FAISS can handle **multi-dimensional vector search**.
* A **simple dictionary or Pandas lookup cannot do this**.

✅ **Use Case:** If you want to **expand your system later** to handle logs or structured errors, FAISS is more future-proof.

## **Comparison Table: FAISS vs Alternatives**

## **When Should You Use FAISS?**

### **✅ Use FAISS If:**

* Your dataset is **growing** beyond **500-1000 failure reasons**.
* You want **semantic search** (meaning-based matching, not just words).
* Users might type errors **differently** but still expect correct results.
* You plan to **scale the system** (e.g., adding log analysis, multi-field search).

### **❌ Skip FAISS If:**

* You **only have 20-50 fixed failure reasons**.
* **Exact matching** is enough for your needs.
* **Speed is the highest priority**, and you don't need intelligent matching.

## **Final Recommendation**

Since your dataset is **currently small**, you can start with **dictionary or fuzzy matching**.  
But if you plan to **scale** or want **better matching**, **FAISS is the better choice**.

Let me know if you need help with implementation!