Assignment: Development of an Autonomous QA Agent for Test Case and Script Generation

Objective

Build an intelligent, autonomous QA agent capable of constructing a "testing brain" from project documentation. The system will ingest support documents (e.g., product specifications, UI/UX guidelines, mock APIs) alongside the HTML structure of a target web project. Using these inputs, the agent should:

- **Generate Test Cases** — Produce comprehensive, documentation-grounded test plans and test viewpoints.
- **Generate Selenium Test Scripts** — Convert the generated test cases into executable Python Selenium scripts for automated testing.

The backend must be implemented using **FastAPI or Flask**, with a **Streamlit** user interface.
All test reasoning must be grounded strictly in the provided documents—no hallucinated features.

Submission Guidelines

Candidates must provide:

1. **Source Code Repository** (GitHub/GitLab/etc.)
2. **README.md** including:
   - Setup instructions (Python version, dependencies)
   - How to run FastAPI/Flask + Streamlit
   - Usage examples
   - Explanation of the included support documents
3. **Project Assets**
   - The checkout.html file
   - The 3–5 support documents used.
4. **Demo Video (5–10 minutes)** demonstrating:
   - Uploading documents + HTML
   - Building the knowledge base
   - Generating test cases
   - Selecting a test case
   - Generating Selenium scripts

Evaluation Criteria

1. Functionality

- Does the system fulfill all phases (ingestion → test cases → script generation)?

2. Knowledge Grounding

- Are test cases based strictly on provided documents?
- No hallucinations or fabricated features.

3. Script Quality

- Are Selenium scripts clean, correct, and runnable?
- Do selectors match the actual HTML?

4. Code Quality

- Modular, readable, well-structured code
- Clean backend with FastAPI/Flask + Streamlit interface

5. User Experience

- Simple, intuitive UI
- Clear system feedback (e.g., “Knowledge Base Built,” “Generating Script…”)

6. Documentation

- Clear, detailed README.md
- Instructions for setup, dependencies, and usage

Project Assets (To Be Created or Provided)

Your assignment will focus on a simple, single-page web application and a set of support documents.

1. Target Web Project (checkout.html)

A single-page **“E-Shop Checkout”** HTML file containing:

Features (5–10 items)

- 2–3 items with **“Add to Cart”** buttons
- A **cart summary** section with item quantity inputs and total price
- A **discount code** input field
- A **User Details form** (Name, Email, Address)
- **Form validation** with inline error messages (e.g., invalid email, required fields)
- **Shipping method** radio buttons (Standard, Express)
- **Payment method** radio buttons (Credit Card, PayPal)
- A **“Pay Now” button** that, if the form is valid, displays “Payment Successful!”

2. Support Documents (3–5 files)

Examples include:

1. **product_specs.md**
    - Contains feature rules—for example:
    - “The discount code SAVE15 applies a 15% discount.”
    - “Express shipping costs $10; Standard shipping is free.”
2. **ui_ux_guide.txt**
    - Contains UI/UX guidelines—for example:
    - “Form validation errors must be displayed in red text.”
    - “The ‘Pay Now’ button should be green.”
3. **api_endpoints.json (optional but recommended)**
    - Example:

```
    {
    "POST /apply_coupon": {"code": "string"},
    "POST /submit_order": {"name": "string", "email": "string"}
}
```

Functional Requirements

Phase 1: Knowledge Base Ingestion & UI (Streamlit)

Required UI Features

The Streamlit UI must allow users to:

1. **Upload support documents** (MD, TXT, JSON, PDF, etc.)
2. **Upload or paste the checkout.html file**
3. Click **“Build Knowledge Base”**

Content Parsing

Use appropriate libraries to extract text from documents, such as:

- unstructured
- pymupdf (fitz)
- Custom parsers for JSON, HTML, etc.

Vector Database Ingestion

- Implement text chunking (e.g., **RecursiveCharacterTextSplitter**)
- Preserve metadata (e.g., "source_document": "product_specs.md")
- Generate embeddings using a model such as those from Hugging Face
- Store vectors + metadata in a vector DB (Chroma, FAISS, Qdrant, etc.)

Phase 2: Test Case Generation Agent

UI

Provide an “Agent” section where the user can request test cases.
Example prompt:

“Generate all positive and negative test cases for the discount code feature.”

RAG Pipeline

1. Embed the user’s query
2. Retrieve relevant chunks from the vector database
3. Feed retrieved context + user query into an LLM (Ollama, Groq, local HF model, etc.)

LLM Output Requirements

The agent must respond with **structured test plans** in JSON or Markdown table format.

Example Test Case Output

Test_ID: TC-001
Feature: Discount Code
Test_Scenario: Apply a valid discount code 'SAVE15'.
Expected_Result: Total price is reduced by 15%.
Grounded_In: product_specs.md

All test reasoning must reference the source document(s).

Phase 3: Selenium Script Generation Agent

Extend the UI to allow:

- Selecting one of the generated test cases
- Clicking **"Generate Selenium Script"**

Agent Logic

The agent must:

1. Receive the selected test case
2. Retrieve the full content of **checkout.html**
3. Retrieve relevant documentation snippets from the vector DB
4. Use an LLM to generate a **runnable Selenium Python script**

Prompt Requirements

The LLM must be instructed to:

- Act as a **Selenium (Python) expert**
- Use appropriate selectors (IDs, names, CSS selectors) based on the actual HTML
- Produce high-quality, fully executable code

Output

Display the generated Python script in a code block for easy copying.