2025-12-01_RAG_chatbot_Architecture_Massimo_Briceno_v2

Author: Massimo Briceno
Last Update: 01/12/2025

Qwen-2.5:3B + LlamaIndex + ChromaDB
Salesforce Commerce Cloud Documentation Assistant

Table of Contents

1. Project Overview

This project implements a **Retrieval-Augmented Generation (RAG)** chatbot capable of answering technical questions about the Salesforce Commerce Cloud platform.

The system uses:

- **Qwen-2.5 3B** as the LLM
- **LlamaIndex** as the orchestration and retrieval engine
- **ChromaDB** as the vector store
- **FastAPI** as the serving layer
- **Streamlit** and CLI as user interfaces

The chatbot runs **fully offline and locally**, processing both **official online documentation** and the **legacy PDF** provided in the assignment.

2. Objectives & Requirements

2.1 - Requirements

- Build a **local RAG chatbot**
- Use a dataset defined by Salesforce Commerce Cloud docs:
  - B2C Commerce Architecture
  - SFRA (Storefront Reference Architecture)
  - Composable Storefront
  - Hybrid Storefront
  - B2C Dev APIs
- The bot must answer questions like:
  - “Which API retrieves product stock?”
  - “How to extract category description for a given locale?”
  - “Difference between Primary and Secondary Instance Group?”
- Provide:
  - **Documentation of the architecture**
  - Explanation of technology choices
  - Setup instructions

2.2 - Additional non functional requirements (Exiters)

- Fast ingestion
- Modular codebase
- Portable: runs on Windows/Linux/macOS
- Lightweight model, CPU-friendly

3. Architectural Principles

The system follows industry standards for RAG systems:

3.1 - Separation of concerns

The System requirements were analyzed and responsibilities where classified as follows:

- Ingestion
- Indexing
- Retrieval
- Generation
- API Serving
- UI Layer

This choice was made to improve isolation and reduce use dependencies between components. The main goal is to improve maintenance and future developments.

3.2 - Local-first design

Everything must run without internet. This meant to be tested on more then one host. The tests where conducted over 2 different stations.

3.3 - Modularity

Each component can be replaced independently. This allow the project modules to be interchangeable at any given time. Another principle used and directly related to modularity is dependency injection, so the resulting modules only uses the import directives which are directly tied to its own legacy use dependency.

3.4 - Deterministic responses

Answers must rely strictly on retrieved context. The main goal was to avoid online consulting and to isolated knowledge context on the `docs/` project directory. Any documentation can be easily added at any given time.-

4. System Architecture

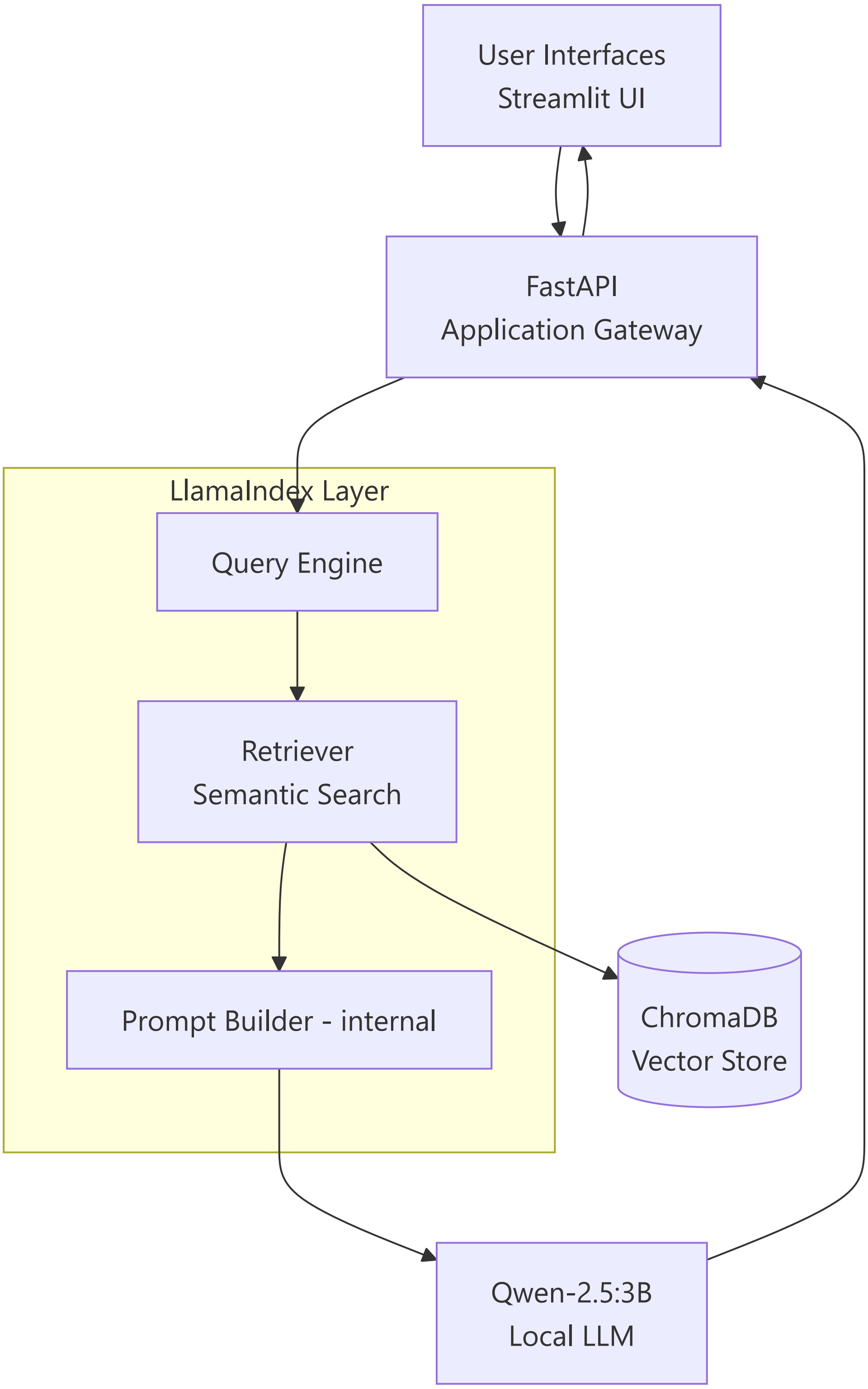
The following will show a high level system architecture, show cased with a `mermaid diagram` to help in visualization.

The prompt builder is show only as a future development note if testing case show hallucination. Internally LlamaIndex uses a default template but it can be changed with:

```
from llama_index.core import ServiceContext
from llama_index.core.prompts import PromptTemplate

template = """
You are a Salesforce Commerce Cloud assistant.
Answer ONLY using the provided context.

Context:
{context_str}

Question:
{query_str}
"""
```

```
qa_prompt = PromptTemplate(template)

service_context = ServiceContext.from_defaults(
    qa_prompt=qa_prompt
)

query_engine = index.as_query_engine(service_context=service_context)
```

4.1 High-Level Diagram



4.2 Architecture Components

4.2.1 - Ingestion Layer

- Extracts text from PDF/HTML using `unstructured`
- Chunks text via LlamaIndex NodeParser
- Embeds chunks using BGE-small embeddings
- Stores embeddings in ChromaDB

4.2.2 - Vector Store Layer

- Uses ChromaDB as persistent, local vector store
- Enables fast k-NN retrieval
- Supports filtering by metadata (file, section, locale)

4.2.3 - Retrieval Layer (LlamaIndex)

- Converts questions into semantic search queries
- Retrieves relevant chunks
- Reranks (optional)
- Prepares context prompt for model

4.2.4 - LLM Layer (Qwen-2.5:3B)

- Performs reasoning and synthesis
- Receives retrieved context
- Produces final grounded answer

4.2.5 - Backend Layer (FastAPI)

- `/ask` endpoint
- Central orchestrator
- Logging & monitoring

4.2.6 - UI Layer

- Streamlit → user-friendly UI
- CLI → debugging and automation

5. Data Flow

`User → UI → FastAPI → LlamaIndex Query Engine → → ChromaDB search → top-k chunks → → Prompt Builder → Qwen-2.5:3B → Response → → FastAPI → UI → User`

6. Architectural Choices

6.1 Why RAG and Not Fine-Tuning

Fine-tuning	RAG
Requires GPU and 10+ hours	No training needed
Cannot be updated easily	Add PDFs and re-index
Harder to control hallucinations	Much safer (grounded answers)
Expensive	Completely free

This assignment explicitly requires RAG.
The model must answer questions **based on indexed documentation**, not “memorized” content.

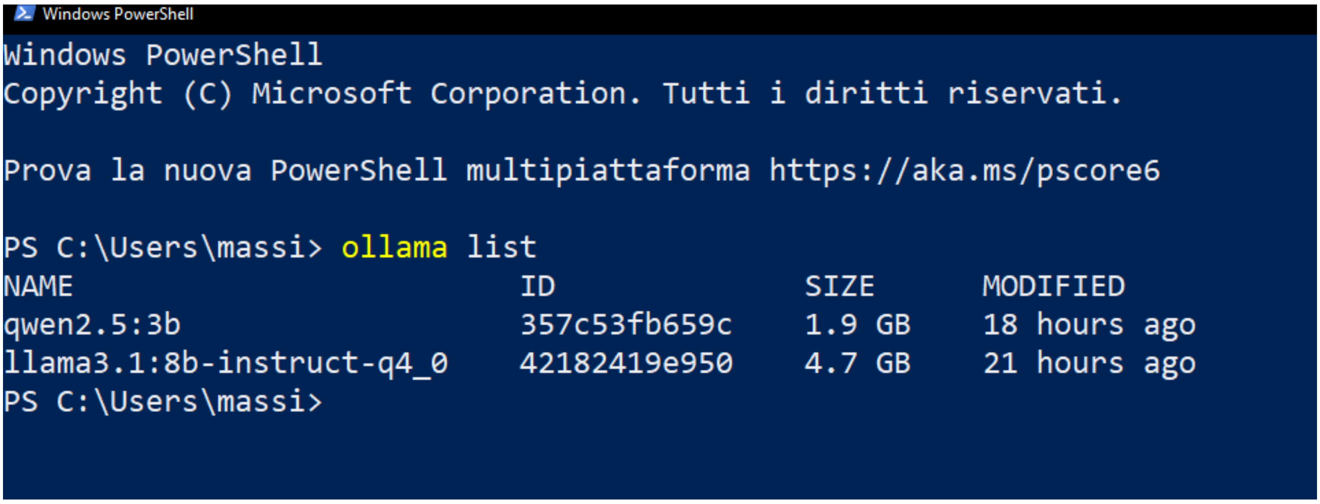
6.2 Why Qwen-2.5:3B Instead of Llama-3

Feature	Qwen-2.5:3B	Llama 3 8B
Speed	Much faster	Slower on CPU
VRAM needs	~4GB	8–10GB
Accuracy on RAG	High	High
Context window	Very large	Medium
Quality/Size ratio	Excellent	Good

Llama 3 8B was tested in the first testing host but even with a decent GPU (NVIDIA GeForce 1660 Super) the request model response waited on timeout, so it needed a timeout call setup to 120.

```
def set_llm():

        llm = Ollama(

            model="qwen2.5:3b",

            request_timeout=120,

            keep_alive="10m"

        )

        return llm
```

Another issue which emerged from using Llama-3 was its footprint and consequently CPU consumption as show in following image:



Reasons for choosing Qwen-2.5:3B:

- Designed for **CPU and edge devices**
- Excellent performance on RAG tasks
- Low memory footprint
- Faster inference → better user experience

- Runs locally even on laptops

6.3 Why ChromaDB Instead of Alternatives

Alternatives evaluated:

Vector Store	Pros	Cons
ChromaDB	Fast, local, simple, persistent	None for this scale
FAISS	Very fast	No persistence/metadata
Weaviate	Powerful	Requires server + Docker
Pinecone	Managed cloud	Not local

Why ChromaDB is selected:

- 100% local, persistent, lightweight
- Excellent Python API
- Perfect for small/medium corpora like Salesforce docs
- Zero maintenance

6.4 Why LlamaIndex

Alternatives considered:

- LangChain
- Custom RAG pipeline

Choice motives:

- Cleaner abstractions
- Less boilerplate than LangChain
- Purpose-built for document indexing
- Built-in adapters for Chroma
- Rich evaluation tools
- Easy modularity

7. Technology Stack

Model

- Qwen-2.5:3B (local inference)

Retrieval Engine

- LlamaIndex

  Vector Store
- ChromaDB

Parsing

- Unstructured

Backend

- FastAPI

User Interfaces

- Streamlit UI
- CLI tool for automated run

8. Ingestion Pipeline

8.1 - Steps:

1. Load files from `/data` folder
2. Parse documents with `unstructured`
3. Chunk using LlamaIndex NodeParser
4. Generate embeddings (BGE-small)
5. Store in ChromaDB with metadata
6. Persist index for future queries

8.2 - Output:

`/storage ├── docstore.json ├── index_store.json`

9. Query Pipeline

1. User submits a question
2. LlamaIndex Query Engine retrieves top-k relevant chunks
3. Ranking + optional rescoring
4. Context-packed prompt is sent to Qwen-2.5:3B
5. The model generates a grounded answer
6. FastAPI returns the output to Streamlit/CLI

10. Modularity & Extensibility

The architecture is fully modular:

Component	Can be replaced with
Qwen-2.5:3B	Llama 3, Mistral, Gemma
ChromaDB	FAISS, Weaviate, PGVector
LlamaIndex	LangChain
Streamlit	Vue.js, React, Django frontend, Flutter
FastAPI	Flask, Django REST Framework

Updating documents

Simply add new PDFs and run:

```
python src/ingestion/ingest.py
```

11. Documentation Consulted

Salesforce official sources used (online):

- B2C Commerce Architecture docs
- Storefront Reference Architecture (SFRA)
- Composable Storefront documentation
- Hybrid Storefront model
- B2C Dev API references

12. Deployment Notes

Works on:

- Windows 10/11
- macOS
- Linux (Ubuntu/Debian)

Dependencies:

`pip install -r requirements.txt`

Run services from root dir with:

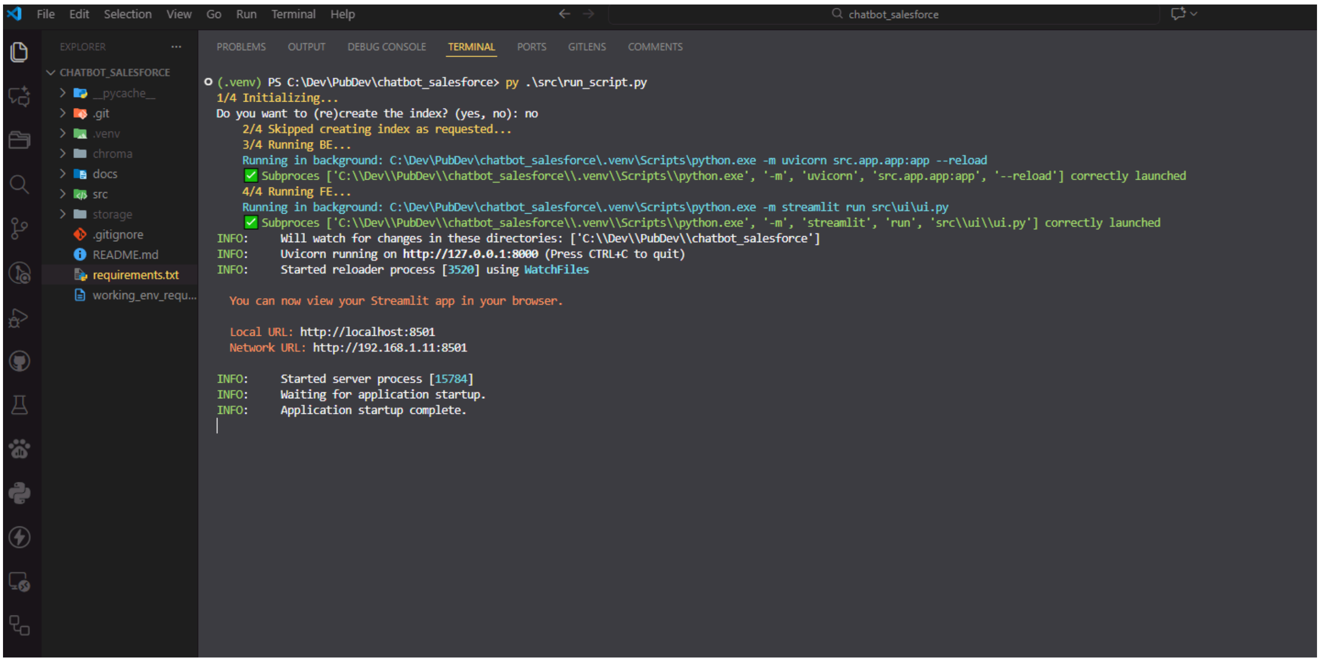
```
python src/ingestion/ingest.py
fastapi run src/app/app.py
streamlit run src/ui/ui.py
```

Or simply use the given `run_script.py` :

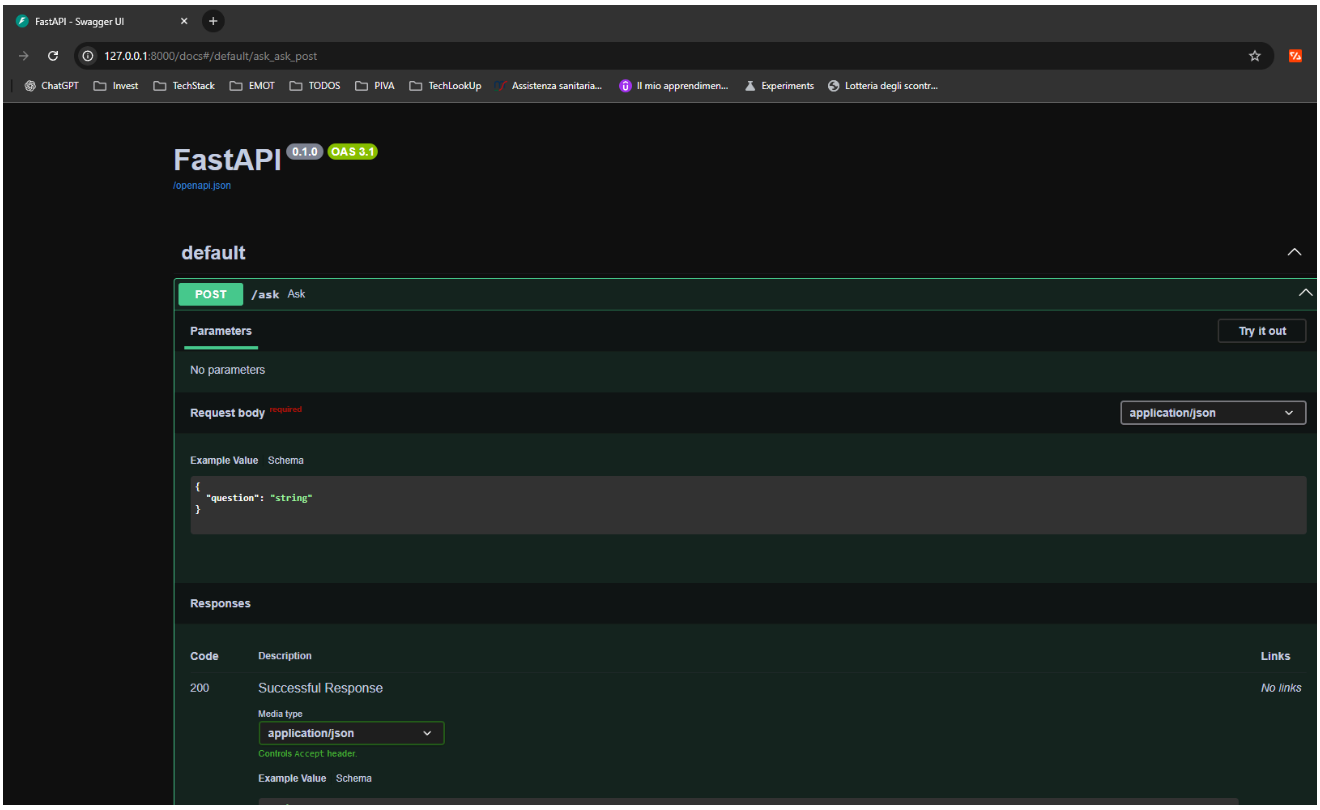
```
python run_script.py
```

13 - Demo Showcase

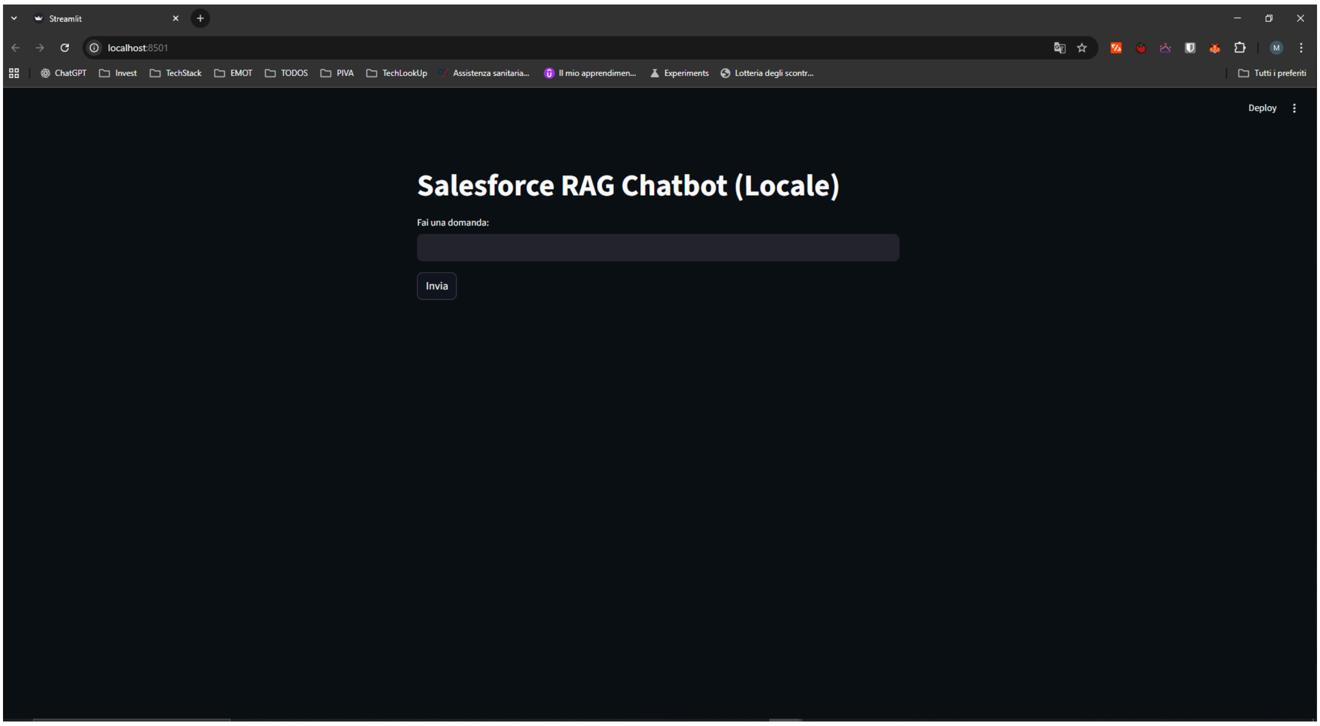
1. Running the script will use the logger to show progression and exit with error if occurs:



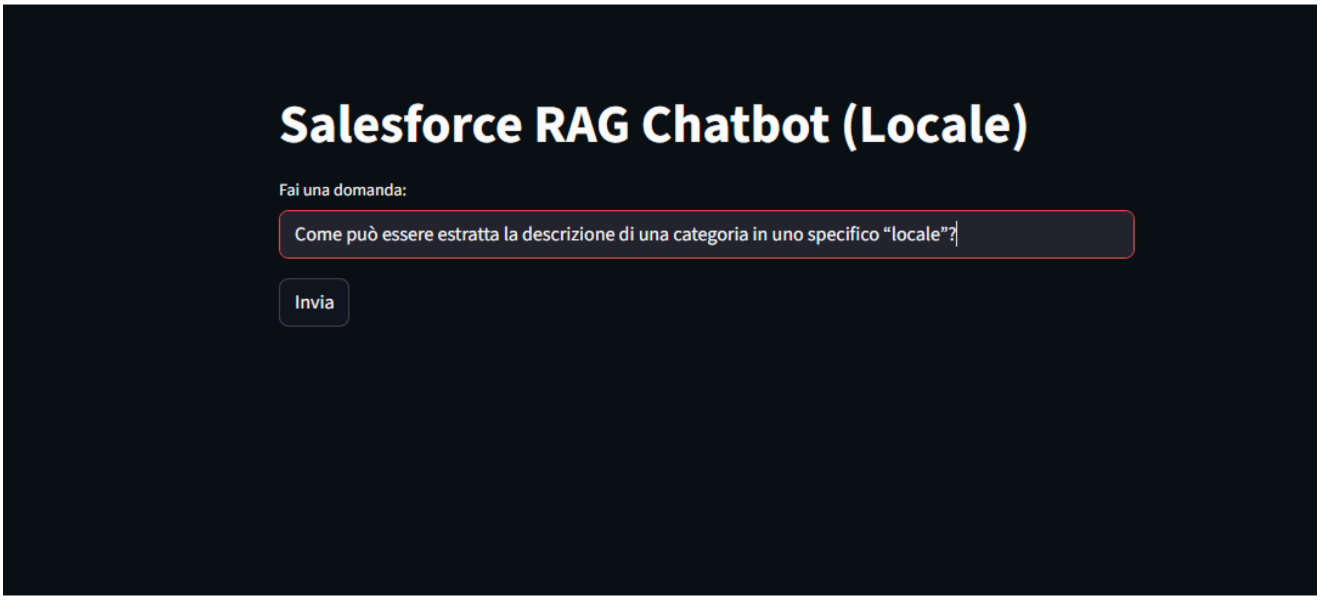
2. Once the BE FastAPI app is running as show above, it can be reached at: http://127.0.0.1:8000/docs - i.e. localhost address:



3. The UI runs in parallel if started from the script and the initial window has the following interface:



4. User inserts input question as follows:



5. Once pressed the "Invia" button, a circular indicator will be show until response is fetched:

6. Once the response returns status code 200 , it will be shown with a green success bar. In case of error an error bar and the error message will be shown isntead.

7. The BE still running can log activity on the running shell: