

Interoperability with Digital Twins

Gabriel Venezia
Group 349B

HSML Web App

Overview

- **Purpose:** The HSML Web App is a web-based interface designed to simplify the creation and management of digital twin metadata.
- **Functionality:** Users can generate HSML-compliant JSON files for Entities, Agents, and Credentials.
- **Technology Stack:**
 - **Frontend:** HTML templates, CSS for global styling, image assets
 - **Backend:** Flask (`app.py`) routes user inputs to the HSML API
 - **Storage:** `.env` file for database credentials, supports secure identity via `.pem` files

What it does

✓ 1. Register Entities, Agents, and Credentials

- **Entity** = a person or organization
- **Agent** = a digital representative that acts on behalf of an entity
- **Credential** = a digital certificate or claim about an entity (e.g. "This agent has permission to do X")

🔑 2. Generate and Download Cryptographic Keys

- Automatically creates .pem private keys used for secure authentication
- Lets users download the key for later use in signing or authentication

📄 3. View and Save HSML JSON Documents

- Automatically builds JSON documents that follow the HSML schema
- These documents are signed, timestamped, and ready to be used in HSML-compliant systems

HSML System

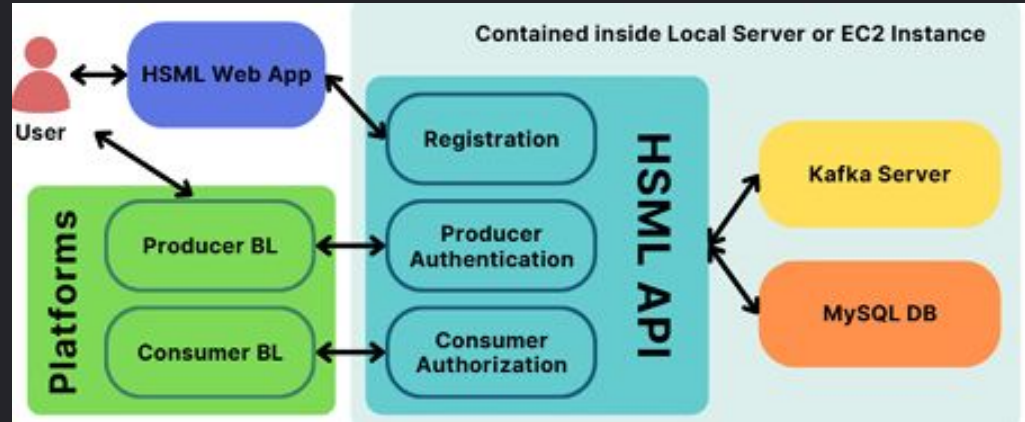
User uses the **Web App** to register/log in with a .pem key.

Web App sends data to the **HSML API** (hosted locally or on EC2).

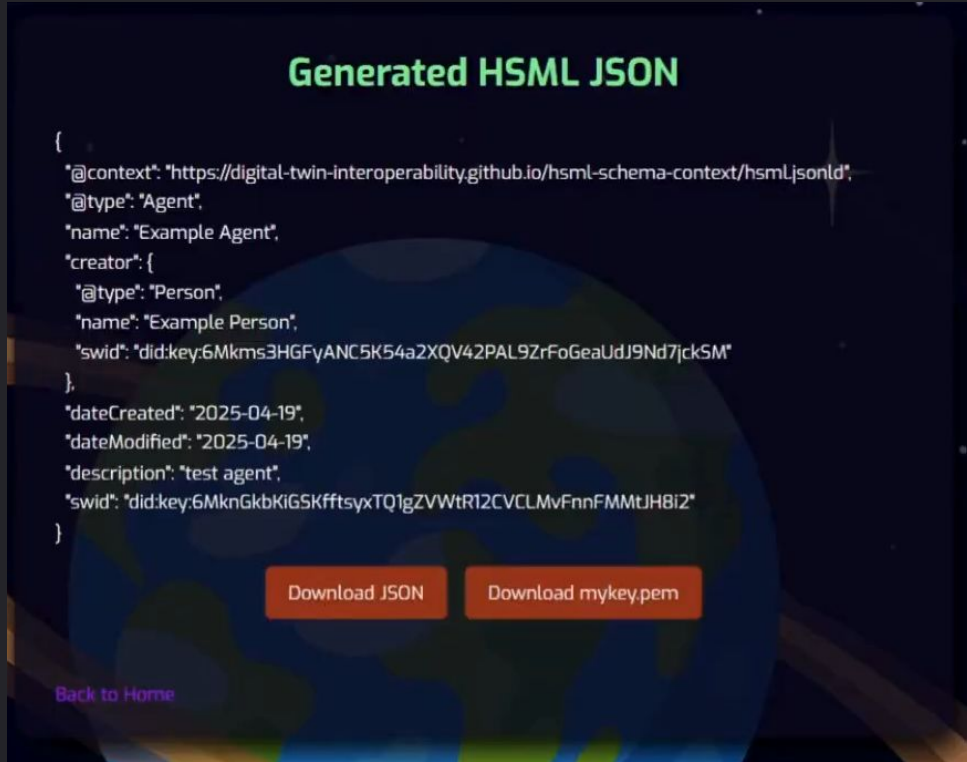
HSML API handles registration, authentication, and authorization.

MySQL stores identities; **Kafka** manages data streams.

Producers/Consumers connect to Kafka only if authorized via **Credentials**.



Example: Registering an Agent



After you register an Agent, the app:

1. **Sends your Agent data to the HSML API**, which creates and signs a digital record.
2. **Receives the response** (in JSON format) that includes all relevant metadata about the Agent.
3. **Displays the generated HSML-compliant JSON** to the user, showing:
 - The Agent's name, type, and unique ID (swid)
 - The creator (a Person object with their own swid)
 - Timestamps for creation and modification
 - A brief description you entered

Web App Results

- Create and manage entities
- Secure authorization
- Interacting with HSML API
- Downloading keys and JSON files
- User-friendly



RAGGY Chatbot



Overview: What Is RAGGY?

RAGGY is a local, document-aware chatbot built to support the Digital Twin project's workflows.



Purpose:

- Help users **ask natural language questions** and get **accurate answers** based on project-specific documents.
- Reduce hallucinations by **only using information from uploaded files**—no outside data.



How It Works:

- Users **upload documents** (manuals, PDFs, specs, etc.)
- RAGGY breaks them into chunks and indexes them
- When a question is asked, RAGGY finds the most relevant chunks and **generates an answer using only that information**

Problem with LLMs

We put a lot of faith in LLM-generated answers, without knowing if what we're being told by the machine is accurate. LLMs are VERY good at being confidently incorrect.

Problem with LLMs

LLMs (like ChatGPT or Claude) generate text based on what they learned during training.

- They **don't know about new or domain-specific information**.
- They often **hallucinate**—making up facts or giving wrong answers.
- They **can't cite sources**, which makes trust and verification hard.

Solution: RAG

Retrieval-Augmented Generation fixes this by:

1. **Retrieving relevant information** from a custom knowledge base (like your documents, manuals, files)
2. **Feeding it to the LLM**, so it answers based *only* on that trusted information

Why Raggy?

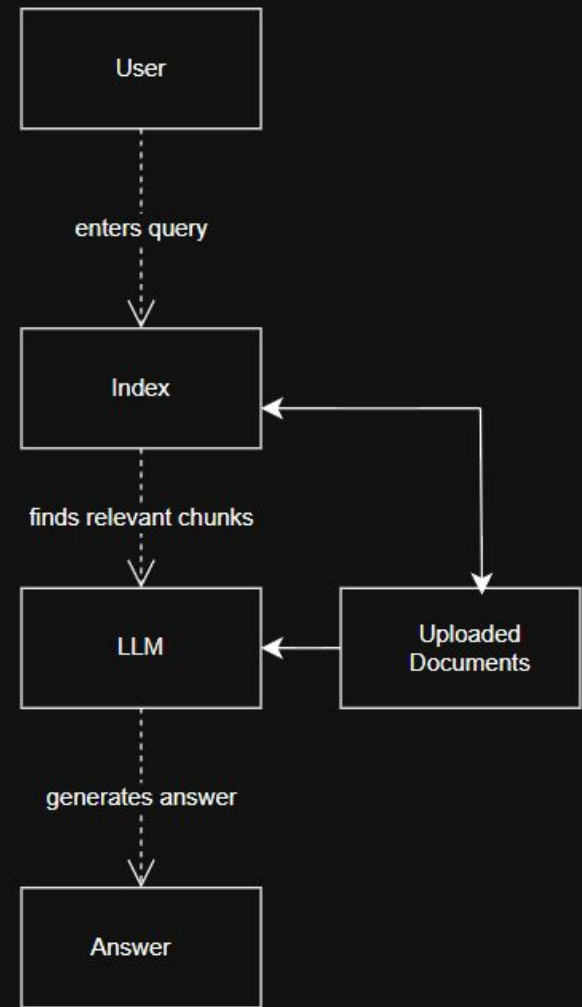
- Keeps answers grounded in trusted, local sources
- No internet access needed—**secure and offline**
- Saves time: instead of digging through files, just ask a question
- Promotes transparency by **citing sources** for every answer

File Structure:

- `RAGGY.py`
- `qa_system.py`
- `vector_store_generator.py`
- `config.py`

How RAG Works

1. User inputs question
2. Index searches uploaded documents (PDFs, manuals, etc.) for relevant info
3. Instead of searching everything, it uses a smart indexing strategy to find only the most relevant chunks
4. LLM receives:
 - a. The user's query
 - b. The retrieved document chunks
 - c. A prompt to guide the response
5. LLM generates an answer

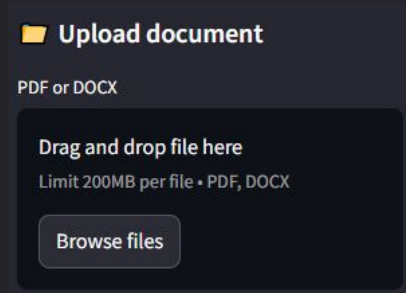


Embedding Documents

- Vector stores are where the chatbot stores and searches the "meaning" of text chunks to quickly find relevant information to answer your questions in a RAG system.


Think of a vector store like a smart library where:

- Each book (document) is broken into pages (chunks)
- Each page is given a unique fingerprint (vector)
- When you ask a question, the system finds pages with similar fingerprints



Vector Store Flow

Document → Split into Chunks → Convert to Vectors → Store in FAISS



When you ask a question, it:

1. Converts your question to a vector
2. Finds similar vectors in the store
3. Returns the most relevant chunks

Question: "What is a digital twin?"

- Converted to vector: [0.1, 0.2, 0.3, ...]
- Finds similar vectors in store
- Returns: "Digital twins are virtual representations of physical assets..."

Vector Store Breakdown

1. Break the document into smaller pieces
2. `loader(name).load()` loads the document
3. `split_documents()` breaks it into those 1000-character chunks
4. Each chunk becomes a separate “document” in the system
5. `DedicatedEmbeddings()` creates an object that converts text to vectors
6. Each chunk is converted into a list of numbers (vector) that represents its meaning
7. `FAISS.from_documents()` stores these vectors in a searchable database

```
# When documents are processed:
splitter = RecursiveCharacterTextSplitter(
    chunk_size=1000, # Each chunk is 1000 characters
    chunk_overlap=200 # Overlap to maintain context
)
docs = splitter.split_documents(loader(name).load())

# Convert to vectors and store
embedder = DedicatedEmbeddings()
faiss_db = FAISS.from_documents(docs, embedder)
```

The chatbot **understands the meaning** of your question as numbers, **finds similar meanings** stored from documents, and then **answers based on those documents**.

Conversation History

- Keeps context across user turns for coherent replies.
- Passes last 10 messages as conversation history to the model.
- History includes system prompt + past messages + current user input.
- Limits history size to avoid token overflow and keep relevance.
- Enables the assistant to remember and build on earlier conversation.

```
def run(self, query):  
    # Format conversation history  
    history_text = ""  
    if self.conversation_history:  
        history_parts = []  
        for i, (q, a) in enumerate(  
            self.conversation_history[-3:], 1  
        ): # Last 3 exchanges  
            history_parts.append(f"Previous Question {i}: {q}")  
            history_parts.append(  
                f"Previous Answer {i}: {a[:200]}..."  
            ) # Truncate for brevity  
            history_parts.append("")  
        history_text = "\n".join(history_parts)  
    else:  
        history_text = "No previous conversation."  
  
    # ...existing code...  
  
    # Add to conversation history  
    self.conversation_history.append((query, response))  
  
    # Keep only last 5 exchanges to prevent context overflow  
    if len(self.conversation_history) > 5:  
        self.conversation_history = self.conversation_history[-5:]
```

Citing Sources

- Compiles **sources cited** in the model's answer based on inline citations (e.g., [Source filename, Page 10]).
- Groups references by **document name** and shows relevant pages/chunks.
- Only includes **sources actually cited** in the answer to keep it relevant.
- Helps users **trace information back** to original documents for verification.

```
for source, source_doc_list in source_docs.items():
    context_parts.append(f"=== FROM DOCUMENT: {source} ===")
    for i, doc in enumerate(source_doc_list[:3]): # Limit per source
        page_num = doc.metadata.get("page", i + 1)
        chunk_num = doc.metadata.get("chunk", i + 1)
        content_type = self._identify_content_type(doc.page_content)

        if content_type["type"] == "table":
            citation = f"[{source} - PAGE: {page_num}, TABLE: {content_type['number']}]"
        elif content_type["type"] == "figure":
            citation = f"[{source} - PAGE: {page_num}, FIGURE: {content_type['number']}]"
        else:
            citation = f"[{source} - PAGE: {page_num}, CHUNK: {chunk_num}]"

        context_parts.append(citation)
        context_parts.append(doc.page_content)
        context_parts.append("") # Add spacing
```


Fine vs. Broad Search

Broad Search

- Queries *all* documents at once.
- Pulls and summarizes the top relevant chunks.
- Simple but may include noisy or less relevant info.

Fine Search

- Uses a small LLM “router” to pick the **most relevant document** first.
- An indexing agent pre-assigns **keywords** to documents on upload.
- Router selects document based on keywords, then searches **within** that document for relevant chunks.
- More precise, reducing noise and improving answer quality.

Raggy Results

- Source-Aware Answers
- Multi-Document Retrieval
- Context-Aware Responses
- User-friendly + convenient

Shows how we can use AI to minimize work

Future Steps

- **Explore better options for fine search**
 - Maybe using learned re-ranker models, semantic similarity,, hybrid search
- **Gather Documents**
 - Collect ~10 documents (simple to complex) as a test corpus.
- **Create Questions**
 - Develop 2-3 questions per document at varying difficulty levels (Easy, Medium, Hard).
- **Unit Testing**
 - Repeatedly ask these questions thousands of times.
Use an LLM evaluator to assign accuracy scores programmatically.
- **Aim for High Accuracy**
 - Target $\geq 90\%$ accuracy on Hard questions—significant milestone for the field.

Lessons Learned

- Clear communication
- Keeping track of progress
- Asking for help when needed
- Putting knowledge and skills into practice

Acknowledgements

Mentors: Thomas Lu and Edward Chow

Peer interns: Gerald, Alicia, Jared, Subhobrata, Aaron, Sydni, Joshua, Sohee, Niki, Diego

Sponsor: PCC Freeman Center and Noel Gonzalez