Guide to Building a Sovereign Synthetic Intelligence System

This system moves beyond traditional "chat with your PDF" tools by replacing static retrieval with a dynamic, query-driven cognitive structure that functions entirely offline.

1. Core Architecture: The Integrated Program

To create a robust local system, you must unify several functional units into a single orchestrator:

- **Knowledge Layer (GraphRAG):** Instead of simple vector searches, use a graph database like **Neo4j** or **NetworkX** to store entities and relationships. This allows the system to traverse disparate data points and "make signal out of noise" without hallucinating.
- **Reasoning Layer (Orchestrator):** Implement a graph-based workflow manager that breaks complex queries into subgoals.
- **Persona Layer:** A dictionary-based system (stored as `.yaml` or `.json`) where attributes (e.g., tone, objectivity, specialized knowledge) are weighted between 0.0 and 1.0.

2. Implementing the "Mixture-of-Experts" Persona Filter

In this framework, personas act as **epistemic lenses** rather than just stylistic templates.

- **Dynamic Weighting:** Upon receiving a query, a reasoning agent (using a model like **DeepSeekR1** or **Llama 3.2**) first analyzes the input and updates the weights in your persona file. If the input challenges the persona's "worldview," the system updates its internal state.
- **The MoE Constraint:** Multiple "expert" personas evaluate candidate knowledge paths retrieved from your graph. They challenge assumptions and ensure the final output is grounded in the explicit provenance of your local data.

3. Local Stack Selection

For true sovereignty and air-gapped security, the system must avoid cloud dependencies:

- **Inference Engine:** Use **Ollama** or **LM Studio** to run models locally (e.g., Mistral, Phi-3, or Llama 3.2).
- **Backend:** A **Python/Django** framework is recommended to manage the "soul" of the system—the `models.py` fields that store persona traits and harvested writing samples.
- **Frontend:** A **Vite/React** or **Next.js** UI allows you to tune persona traits in real-time using sliders for different weights.

4. The Synthetic Intelligence Loop (Step-by-Step)

1. **Ingestion:** Use a tool like **PRAW** (for Reddit) or local file scrapers to convert your history into `.md` files.
2. **Analysis:** A vision model (like **LLaVA**) or a text model extracts entities and relationship vectors from these files.
3. **Graph Construction:** Ingest these entities into **Neo4j**, defining relationships only as they become causally relevant to queries.
4. **Query Execution:** * The user asks a question.
   - **Step A:** The reasoning agent updates the persona `.yaml` weights based on the query's intent.
   - **Step B:** The system queries the graph and returns raw data chunks.
   - **Step C:** The persona "lens" (the updated `.yaml` values) is used as a system prompt wrapper to color and filter the final output.
5. **Validation:** Use **RLHF** (Reinforcement Learning from Human Feedback) to judge the output. If it passes a defined threshold, it is displayed; otherwise, it triggers a tool call for further refinement.

5. Sovereign Deployment and MCP

To allow these systems to evolve autonomously, integrate the **Model Context Protocol (MCP)**.

- **Contextual Evolution:** MCP allows your local agents to "push" knowledge or code back into the system, effectively allowing the persona to "learn" from its own analytical processes.
- **Air-Gapping:** Ensure your Dockerized environment has no outbound network rules, keeping your "Synthetic Intelligence" entirely contained on your hardware.

This architecture ensures that the "ghost in the machine" is not a statistical hallucination, but a traceable, grounded, and evolving persona that reflects your specific data and analytical goals.