Assignment 1: Building an Agentic AI System

In this assignment, you will incrementally build a complete LLM-based system, to understand how modern agent systems are structured

Architecture

- Basic LLM Chat (one-turn)
- Conversation Memory (multi-turn)
- Retrieval-Augmented Generation (RAG)
- Tool-based Computation
- Router-based Agent

Notes:

- Agentic AI is rapidly evolving, no single “correct” agent architecture.
- Students are encouraged to further explore alternative agentic workflows and concepts — such as **ReAct-style agents, tool-augmented reasoning, agent skills, multi-agent systems, or emerging standards like MCP** — beyond the scope of this assignment.

Section 1 & 2 How LLM Chat Works

- how llm generate answers from user inputs
- how to maintain conversation memory

Architecture

- User_input
- Tokenizer (text → token IDs)
- LLM generation (tokens → tokens)
- Tokenizer decode (tokens → text)

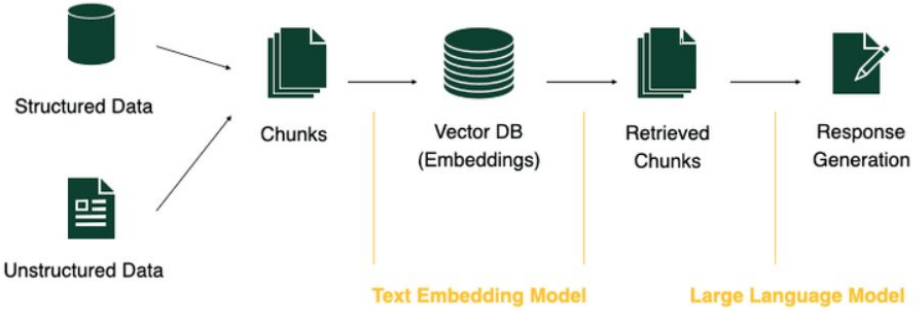
Memory (short-term in-context memory)

- LLM is stateless: each call can only see the current prompt
- We store message history as a list of {role, content}
- Every new user message + past history are included in the next prompt

```python
history = []

def chat(user_input):
    # 1. store user message
    history.append({"role": "user", "content": user_input})

    # 2. build prompt from history
    prompt = apply_chat_template(history)

    # 3. tokenize
    input_ids = tokenizer(prompt)

    # 4. generate new tokens
    output_ids = model.generate(input_ids)

    # 5. decode new tokens
    reply = tokenizer.decode(output_ids)

    # 6. store assistant reply
    history.append({"role": "assistant", "content": reply})

    return reply
```

Section 3 RAG

- LLMs have no access to your private or domain-specific data
- Model knowledge is frozen at training time
- Directly fine-tuning is expensive and inflexible



Architecture

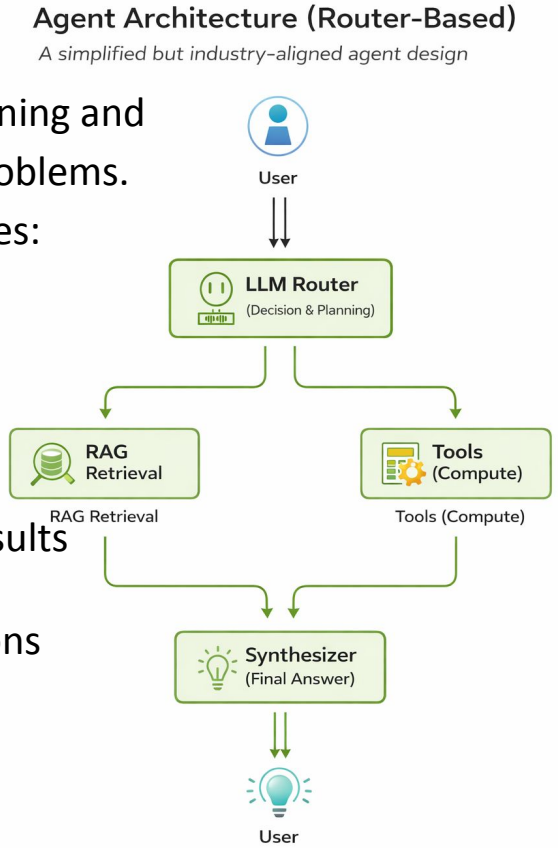
- **Embed and store** external knowledge chunks in DB
- **Retrieve** relevant chunks using vector similarity search
- **Build context** by combing retrieved chunks
- **Generate** an answer under the context of retrieved context

```python
def rag_build(question):
    # 1. load documents
    docs = load_documents()

    # 2. split text into chunks
    chunks = split_text(docs)

    # 3. embed chunks
    vectors = embed(chunks)

    # 4. build vector store
    vector_store = FAISS(vectors)

    return vector_store
```

```python
def rag_answer(question):
    # 1. embed the question
    q_vec = embed(question)

    # 2. retrieve top-k relevant chunks
    docs = vector_store.search(q_vec, k=3)

    # 3. build prompt with context
    prompt = build_prompt(docs, question)

    # 4. generate grounded answer
    answer = llm.generate(prompt)

    return answer
```

Section 5 Agentic AI

- Agentic AI systems use sophisticated reasoning, iterative planning and tool calling to autonomously solve complex and multi-step problems.
- Agentic AI systems typically have the following core capabilities:
  - **Perception**
    Interpret user goals and environment signals
  - **Reasoning & Planning**
    Decide what needs to be done and in what order
  - **Tool Use / Action**
    Call external tools, APIs, or code to obtain real-world results
  - **Memory**
    Maintain short-term or long-term state across interactions
  - **Learning / Adaptation**
    Improve behavior over time via feedback



Section 5 Agentic AI Architecture

- **Perceive**: gather inputs (user query, context, retrieved docs, tool outputs)
- **Reason / Plan**: decide next actions (which tools and which order)
- **Act**: execute tools / retrieval
- **Synthesize**: produce a grounded final answer
- *(Learning / feedback loops are optional and not required here.)*

```python
TOOLS = {
  "rag": rag_answer,
  "inspect_dataset": inspect_dataset,
  "compute_stat": compute_stat,
}

def run_agent(question: str) -> dict:
    # 1) ROUTE (LLM → JSON plan)
    # {"plan": [{"tool":..., "args":...}, ...]}
    plan = route_question(question)

    trace = []
    for step in plan["plan"]:
        tool_name = step["tool"]
        args = step.get("args", {})

        # 2) EXECUTE (deterministic calls)
        result = TOOLS[tool_name](**args)

        trace.append({
            "tool": tool_name,
            "args": args,
            "result": result
        })

    # 3) SYNTHESIZE (combined tool outputs and question)
    final_answer = synthesize_answer(question, trace)

    return {"plan": plan["plan"], "trace": trace, "final_answer": final_answer}
```