Cheat Sheet: Agentic Frameworks and Design Patterns for Effective AI Systems

Estimated time: 20 minutes

---

1. Agentic AI Design Patterns

Agentic design patterns are reusable strategies for organizing multiple language model "agents" into structured workflows. These agents collaborate—each with specialized roles—to handle complex tasks that go beyond what a single prompt can manage. These patterns improve clarity, scalability, and control in LLM-powered applications. They are used to decompose complex problems into smaller, specialized tasks and add structure and memory to multi-step reasoning or decision flows.

1.1 Fundamental Components

- **Agent**: a specialized LLM prompt + logic unit
- **Orchestrator**: routes inputs, maintains state, aggregates outputs
- **Worker**: executes a single responsibility (for example, summarization, translation)
- **Router**: dispatches tasks based on intent or condition
- **Evaluator**: inspects and scores outputs for quality or correctness

These components embody the principle of separation of concerns. Decomposing your workflow into distinct agent roles makes it easier to debug, test, and reuse logic. It mirrors real-world software architecture by assigning specialized responsibilities to each part of the system.

1.2 Core Patterns

Pattern	Description	Use Cases
Orchestration	Central controller coordinates agents and tracks state	Document pipelines, decision trees, role-based workflows
Reflection	Evaluate and refine outputs with internal feedback loops	Quality improvement, self-correction, iterative generation
Sequential Coordination	Chain agents in a fixed order	Multi-step data pipelines (ingest → summarize → refine)
Intent-Based Routing	Dispatch inputs to agents based on user intent/class.	Multi-domain assistants (finance vs. weather vs. chat)
Parallel Execution	Fan-out tasks to multiple agents concurrently	Batch translations, multi-hypothesis generation
Prompt Chaining	Decompose a complex prompt into a sequence of simpler prompts	Complex content creation (outline → draft → edit)

> While "agentic frameworks" can include a variety of libraries and platforms (e.g., LangChain, AutoGen), this reading focuses exclusively on LangGraph as the agentic framework. All examples, patterns, and code snippets use LangGraph's APIs and conventions to illustrate core design patterns for building effective, multi-agent AI workflows.

---

2. LangGraph Workflows

2.1 Agents with Structured Output

First, we need chains of LLMs and prompts that configure agents to perform specific tasks. These agents will later be used in workers to perform certain tasks in the workflow. It's important that these agents have structured outputs so they return data in our desired format. This ensures that **LangGraph workflows** remain reliable and composable, especially when downstream agents depend on the outputs of earlier ones.

```
from pydantic import BaseModel, Field
from langchain_openai import ChatOpenAI
from langchain_core.prompts import ChatPromptTemplate
# instantiate LLM
llm = ChatOpenAI(model="gpt-4o-mini")
# define output schema
class Output(BaseModel):
    name: str = Field(
        description="Name of the structured output"
    )
    field: str = Field(
        description="Field of the structured output"
    )
# create the LLM prompt template
prompt = ChatPromptTemplate.from_messages([
    (
        "human/system",
        "Prompt with {input}"
    )
])
# use LCEL to pipe the prompt to the LLM and pass in the structured output schema
pipe = prompt | llm.with_structured_output(Output)
```

2.2 States

A simple typed dictionary that tracks variables, values and states across a workflow. This state is what gets upated across a LangGraph `StateGraph` workflow so the **field names** and **data types** are **important**.

```
class State(TypedDict):
    string_field: str
    string_list_field: List[str]
    int_field: int
    input_field: str
    output_field: Output
```

In LangGraph, the state serves as a contract between nodes. It defines the schema for what data is expected, produced, and passed between components. This contract enables better debugging, validation, and scalability. It also ensures agents interact through a shared vocabulary, reducing unexpected type mismatches and silent failures.

2.3 Worker Nodes

These are functions that make up the graph and execute certain tasks with the provided input data. They return an update to the persistent state across a `StateGraph`, updating field(s) with the new data.

```
def worker(state: State):
     """Worker that generates the pipe to fill in the output_field of a state"""
    # use the pipe LLM to generate an Output
    output = pipe.invoke({"input": state["input_field"]})
    # return the Output object as a dictionary to update the state
    return {"output_field": output}
```

2.4 Building the Workflow

Component	What It Does	How to Add It
Node	Encapsulates a function (agent/tool) that processes input and returns output	`graph.add_node("name", fn)` or `Node("name", func=fn)`
Edge	Defines a direct connection between two nodes	`graph.add_edge("from_node", "to_node")`
Conditional Edge	Routes execution based on a boolean or value-based condition	`graph.add_conditional_edges("node", {"True": "A", "False": "B"})`
Entry Point	Designates where the graph starts execution	`graph.set_entry_point("start_node")` or `graph.add_edge(START, "to_first_node")`
End Point	Designates a terminal node where execution finishes	`graph.set_finish_point("end_node")` or `graph.add_edge("last_node", END)`
Parallel Branch	Runs multiple nodes concurrently, gathers all results	`graph.add_parallel("parent", ["node1", "node2"])`
State	Stores and passes shared context across the graph	Passed automatically between nodes via input/output dicts

- First initialize the `StateGraph` and pass in the `State` variable that will be tracked across the workflow.
- Then add the nodes to the graph giving them labels.
- Then add the edges to the graph connectin the nodes in your desired order.
- Lastly compile the workflow and invoke it with an input.

```
from langgraph.graph import StateGraph, END, START
# initialize the graph
builder = StateGraph(State)
# add the nodes
builder.add_node("worker", worker)
# add the edges
builder.add_edge(START, "worker")
builder.add_edge("worker", END)
# compile the workflow into an executable
workflow = builder.compile()
# invoke the workflow with an example input
workflow.invoke({"input": "Example input"})
```

2.5 Workflow Visualization

Visualizing LangGraph graphs is easy in a JupyterLab environment. It helps you quickly understand the flow of logic between nodes—especially in more complex graphs involving loops, branching, or multiple agents.

If you're working in a notebook, you can generate and display the graph using Mermaid syntax or as an image.

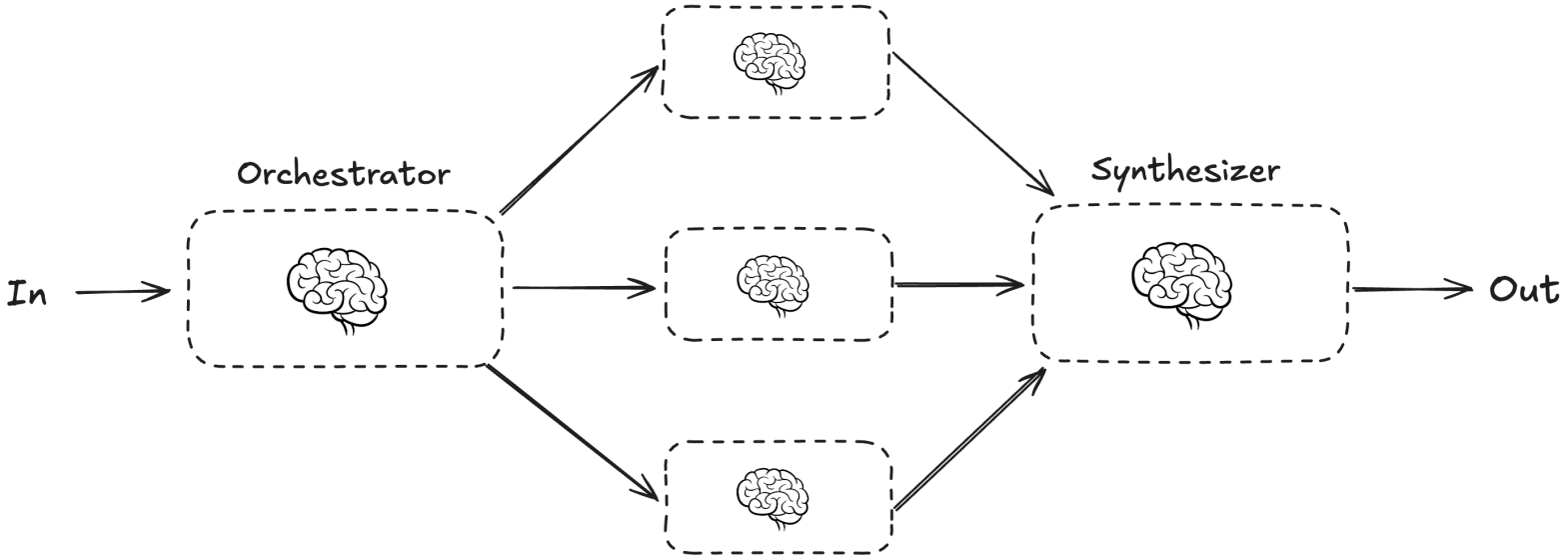
Mermaid Chart Visualization (Jupyter-friendly)

If your `StateGraph` object supports it (ex. `workflow.get_graph()`), you can render it using Mermaid diagrams:

```
from IPython.display import Image, display
# display the orchestrator workflow as a Mermaid-rendered PNG
display(Image(orchestrator_worker.get_graph().draw_mermaid_png()))
```

3. Orchestrator–Worker Pattern

An industry-tested coordination pattern used to break down complex workflows into modular, repeatable tasks. An orchestrator delegates responsibilities to specialized worker agents and aggregates their outputs to form a cohesive result. This pattern improves maintainability, allows for dynamic control flow, and supports multi-stage pipelines with clear structure and logic.

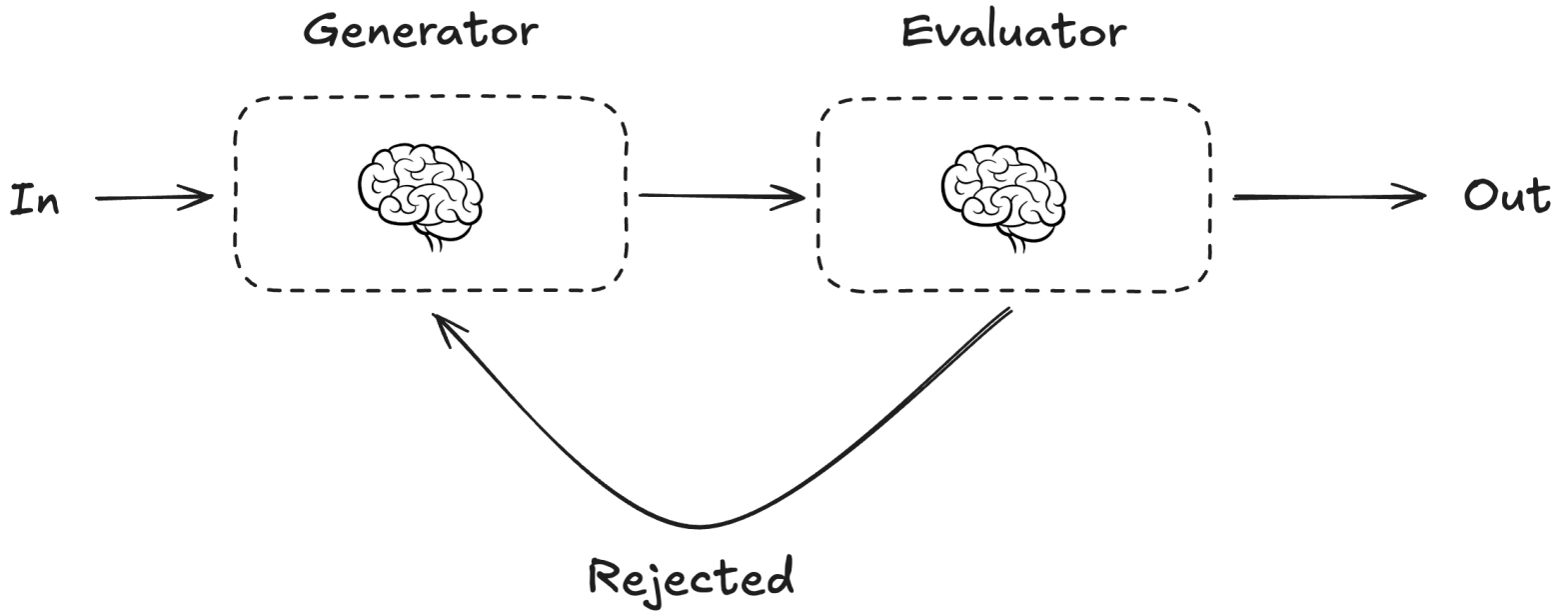


Component	Purpose
Orchestrator Node	Central control unit that routes data, manages logic, and delegates tasks
Worker Node	Executes a single, focused subtask within the larger workflow
Routing Logic	Decides which worker(s) to activate based on input or intermediate state
Dynamic Edges	Connect orchestrator to multiple workers depending on routing conditions

---

4. Reflection Pattern

A self-improving feedback pattern that allows agents to evaluate, critique, and revise their outputs iteratively. By looping results through an evaluator node and back into the generator, this pattern enables internal quality control, self-correction, and optimization, making it ideal for high-stakes tasks requiring accuracy, consistency, or refinement over time.



Component	Purpose
Generator Node	Generates initial output and generates further outputs when feedback is available
Evaluator Node	Scores agent outputs (correctness, style, etc.)
Routing Node	Re-routes based on evaluator feedback, is added as a conditional edge because it might lead to END or loop back to the Generator Node
Testing Node	Unit-tests chains with known in/out examples

Summary

This cheat sheet gives you a practical foundation for applying agentic design patterns using **LangGraph**. By mastering these core patterns and components, you'll be equipped to structure **complex LLM workflows**, enhance reliability through **state management**, and confidently prototype or scale intelligent systems. Use it as a quick **reference** when designing, debugging, or extending your own multi-agent applications.

Author

Joshua Zhou