Multi Agent Design Patterns Cheat Sheet End to End

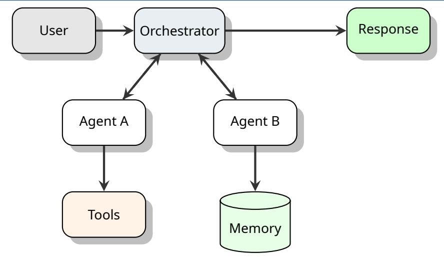
Design, Coordinate, Evaluate, and Deploy Agent Systems

1. DEFINITION

Multi-Agent System (MAS): A system composed of multiple autonomous agents that interact, communicate, and collaborate to solve complex tasks that are difficult or impossible for a single monolithic agent.

- **Why Used:** Decomposes complex problems into manageable subtasks. Allows specialization (e.g., one agent codes, one tests). Improves reliability through redundancy and review loops.

2. HIGH LEVEL ARCHITECTURE FLOW



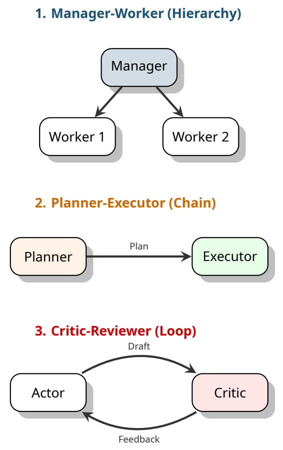
- **Flow:** User Request → Orchestrator (Router) → Specialized Agents → Tools/APIs → Shared Memory → Aggregation → Response.

3. CORE COMPONENTS

- **Orchestrator (Controller):** The "brain" that receives requests, breaks them down, assigns tasks to agents, and synthesizes results.
- **Specialized Agents:** Agents tuned for specific tasks (e.g., Coder, Writer, Reviewer) with distinct prompts and tools.
- **Communication Layer:** The protocol for agents to talk (Message Bus, JSON, Chat History).
- **Memory Layer:** Short-term (context window) and Long-term (Vector DB) storage.
- **Tool & API Layer:** External functions agents can call (Search, Calculator, Database).
- **State Management:** Tracking the overall progress of the workflow.
- **Aggregation Layer:** Combining outputs from multiple agents into a final answer.

4. COMMON MULTI AGENT DESIGN PATTERNS

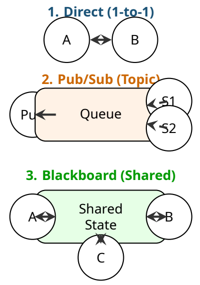
Pattern Visuals


1. Manager-Worker (Hierarchy)
Manager
Worker 1
Worker 2
2. Planner-Executor (Chain)
Planner
Plan
Executor
3. Critic-Reviewer (Loop)
Draft
Actor
Critic
Feedback

- **Coordinator-Worker:** A central coordinator breaks down a task and assigns sub-tasks to workers. Parallelizable.
- **Manager-Worker (Hierarchical):** Recursive structure. A manager assigns high-level goals; workers execute sub-tasks.
- **Planner-Executor:** One agent creates a step-by-step plan; another agent executes it step-by-step.
- **Debate / Voting:** Multiple agents generate answers, critique each other, and vote on the best one.
- **Critic-Reviewer:** Generator agent creates content; Critic agent checks for errors/safety. Loop continues until criteria met.
- **Reflection Loop:** An agent critiques its own output in a loop to self-correct before final response.
- **Swarm / Mesh:** Decentralized. Agents subscribe to tasks or topics without a central leader.

5. AGENT COMMUNICATION MODELS

Communication Topologies



- **Direct Messaging:** Point-to-point. Agent A explicitly calls Agent B. Simple but tightly coupled.
- **Publish-Subscribe:** Agents listen to topics. Agent A posts a "Result"; Agents B and C listen. Decoupled and scalable.
- **Blackboard Model:** Shared global state (the "Blackboard"). Agents read/write to it. Good for complex, non-linear problem solving.

6. STATE AND MEMORY MANAGEMENT

Defining State (LangGraph):

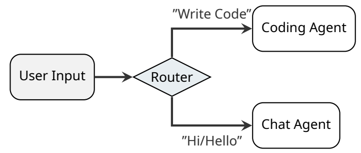
```python
from typing import TypedDict, List, Annotated
import operator

class AgentState(TypedDict):
    # Shared message history
    messages: Annotated[List[str], operator.add]
    # Task tracking
    next_step: str
    errors: List[str]
```

- **Shared Memory:** All agents see the same conversation history. Good for context, bad for cost/token limits.
- **Local Memory:** Agents keep their own private context. Reduces noise but limits global awareness.
- **Long-term Memory:** Using Vector DBs (RAG) to persist knowledge across sessions.
- **Consistency Handling:** Using a state machine (e.g., LangGraph) to ensure agents don't overwrite or lose state during transitions.

7. TASK DECOMPOSITION AND ROUTING

Routing Flow



Router Implementation:

```python
def route_request(state: AgentState):
    # Intent Classification
    intent = llm.classify(state["input"])

    if intent == "write_code":
        return "coder_agent"
    elif intent == "search_web":
        return "research_agent"
    return "general_agent"
```

- **Intent Classification:** First step. Classify user query (e.g., "Coding", "General", "Math") to select the right agent.
- **Task Splitting:** Breaking "Build a website" into "Write HTML", "Write CSS", "Write JS".
- **Agent Selection:** Dynamic routing based on agent capability descriptions.
- **Load Balancing:** If multiple agents can do the task, distribute to the

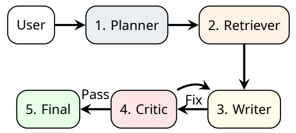
least busy one.

8. IMPLEMENTATION FRAMEWORKS

- **LangGraph:** Best for stateful, cyclic graphs and production control flows.
- **AutoGen (Microsoft):** Best for conversational multi-agent patterns and code execution.
- **CrewAI:** High-level wrapper for role-playing agents. Easy to start.
- **Semantic Kernel:** Good for enterprise integration and rigorous function calling.
- **Custom Orchestrators:** Use when you need exact control over the loop and state (Python/FastAPI).

9. END TO END EXAMPLE FLOW

Scenario: ”Research the latest EV battery tech.”



1. **Planner:** Breaks ”EV Tech” into ”Solid State”, ”LFP”, ”Charging”.
2. **Retriever:** Runs 3 parallel searches.
3. **Writer:** Summarizes search results into sections.
4. **Critic:** Checks for citation accuracy. Rejects ”Solid State” section (too vague).
5. **Writer:** Fixes ”Solid State” section.
6. **Aggregator:** Compiles sections into final report.

10. PERFORMANCE METRICS

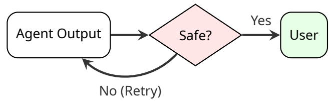
- **Task Success Rate:** % of user requests fully satisfied without error.
- **Completion Time:** Total latency from Request to Response.
- **Agent Utilization:** How busy each agent is (helps optimization).
- **Error Rate:** Frequency of agent crashes or invalid outputs (JSON errors).
- **Retry Rate:** How often the loop has to self-correct.
- **Cost per Task:** Token usage $\times$ Price per token.
- **User Satisfaction:** Thumbs up/down on final result.

11. SCALING AND OPTIMIZATION

- **Parallel Execution:** Run independent sub-tasks (e.g., search queries) at the same time using ‘async‘.
- **Caching:** Cache tool outputs (Search Results) to save time/money on identical queries.
- **Agent Pooling:** Maintain a pool of initialized agents to avoid cold-start latency.
- **Timeouts:** Set strict limits on agent execution time to prevent hanging.

12. RELIABILITY AND SAFETY

Guardrail Pattern



Loop Detection Logic:

```
def loop_guard(state):
    # Prevent infinite cycles
    if len(state["messages"]) > 20:
        return "error_agent"

    # Check for repeating feedback
    if state["last_critique"] == state["current_critique"]:
        return "human_handoff"

    return "continue"
```

- **Loop Detection:** Orchestrator must count iterations and kill infinite loops (e.g., Critic rejecting forever).
- **Deadlock Prevention:** Ensure agents aren’t waiting on each other circularly.
- **Tool Misuse Protection:** Read-only permissions for sensitive tools (e.g., ”Delete File” disabled).
- **Fallback Strategies:** If ”Complex Planner” fails, fall back to ”Simple LLM” to give a basic answer.

13. COST OPTIMIZATION

- **Smaller Agents:** Use GPT-4o-mini or Llama 3 for simple tasks (routing, summarizing); reserve GPT-4 for reasoning.
- **Limiting Context:** Don’t pass full history to every agent. Pass only relevant summaries.
- **Early Stopping:** Stop the chain if confidence is high or budget is exceeded.

14. COMMON FAILURE POINTS

- **Infinite Loops:** Planner $\leftrightarrow$ Critic arguing indefinitely. Fix: Max turns.
- **Hallucination Amplification:** One agent makes up a fact; next agent assumes it’s true and builds on it. Fix: Grounding/Search.
- **Coordination Failures:** Agents passing data in wrong formats (JSON vs Text). Fix: Pydantic Validation.
- **State Corruption:** Global memory becoming messy/contradictory. Fix: Scoped memory.

15. SUMMARY

Goal: To solve tasks too big for one model by emulating a human team.

- **Core:** Decomposition + Routing + Iteration.
- **Key:** Orchestration is as important as intelligence.
- **Outcome:** Robust systems that can code, research, and reason autonomously.