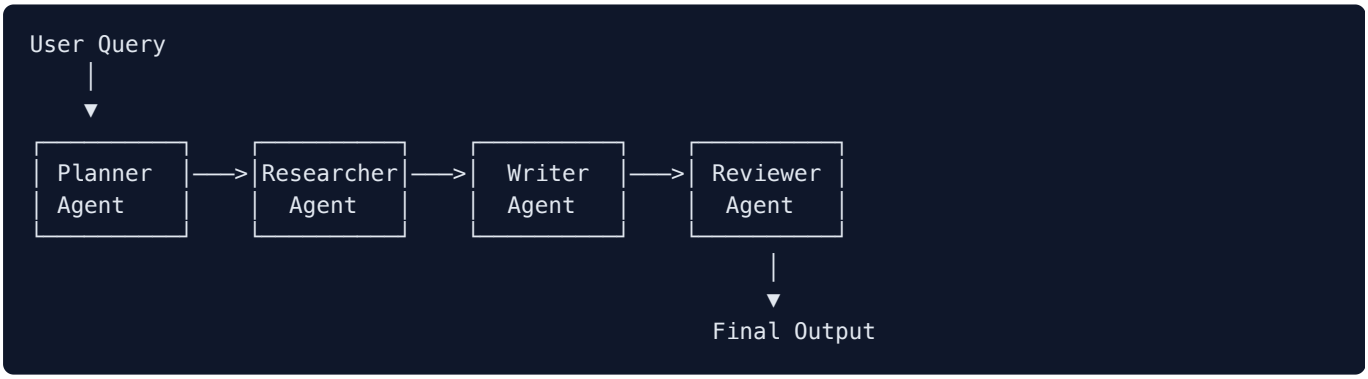
Multi-Agent Patterns Overview

Multi-agent systems let you decompose complex tasks across specialized agents, but every pattern adds coordination overhead. Choose the simplest topology that handles your task -- most problems do not need more than two or three agents.

Pattern	Agents	Communication	Best For
Single agent	1	N/A	Simple tasks with few tools
Sequential pipeline	2-5	Output -> Input	Multi-step processing
Parallel fan-out	2-10	Same input, merge outputs	Independent subtasks
Router/dispatcher	1 + N specialists	Router selects specialist	Domain-specific handling
Supervisor-worker	1 + N workers	Supervisor delegates and reviews	Complex task decomposition
Hierarchical	N layers	Multi-level delegation	Enterprise workflows
Debate/consensus	2-5 peers	Argue until agreement	High-stakes decisions
Swarm	N peers	Dynamic handoffs	Flexible, exploratory tasks

Sequential Pipeline

The sequential pipeline is the most intuitive multi-agent pattern -- each agent completes its stage before handing off to the next. Use it when your task has clear, ordered phases that require different expertise.



When to Use

- Tasks have clear sequential stages
- Each stage needs different skills/tools
- Output of one stage is input to the next

Implementation

```
async def sequential_pipeline(query: str):
    # Stage 1: Plan
    plan = await planner.run(f"Create a research plan for: {query}")
```

```
    # Stage 2: Research
    research = await researcher.run(
        f"Execute this research plan:\n{plan}\n\nGather relevant information."
    )

    # Stage 3: Write
    draft = await writer.run(
        f"Write a report based on this research:\n{research}"
    )

    # Stage 4: Review
    final = await reviewer.run(
        f"Review and improve this draft:\n{draft}\n\n"
        f"Original query: {query}"
    )

    return final
```

Parallel Fan-Out

Fan-out runs multiple agents simultaneously on the same input, then merges their results. This is ideal when you need multiple independent perspectives or can decompose a task into non-overlapping subtasks.

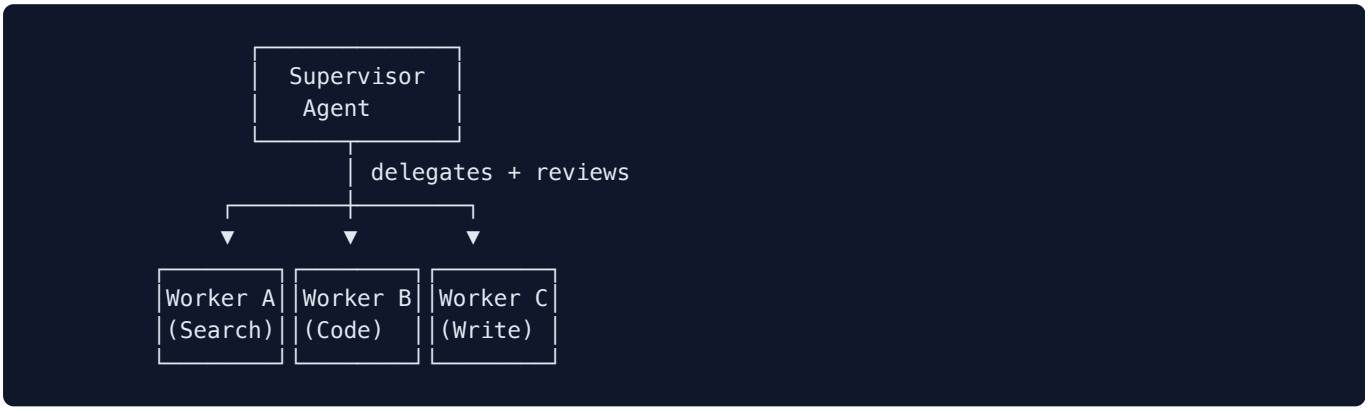
Implementation

```
import asyncio

async def fan_out(query: str, agents: list[Agent]) -> str:
    # Run all agents in parallel
    tasks = [agent.run(query) for agent in agents]
    results = await asyncio.gather(*tasks, return_exceptions=True)

    # Filter out failures
    successes = [r for r in results if not isinstance(r, Exception)]

    # Aggregate results
    combined = "\n\n---\n\n".join(successes)
    final = await aggregator.run(
        f"Synthesize these perspectives into a single answer:\n{combined}"
```

```
    )
    return final
```

Supervisor-Worker

The supervisor-worker pattern gives one agent the authority to plan, delegate, and review -- making it the most powerful pattern for complex, multi-step tasks that require coordination and quality control.



Supervisor Prompt

```
You are a project supervisor managing a team of specialist workers:
- researcher: Searches documents, web, and databases for information
- coder: Writes and debugs code
- writer: Creates and edits text content
- analyst: Performs data analysis and creates charts

For the given task:
1. Break it into subtasks
2. Assign each subtask to the best worker
3. Review each worker's output
4. Request revisions if quality is insufficient
5. Combine results into the final deliverable

Respond with a JSON plan:
{
  "subtasks": [
    {"worker": "researcher", "task": "...", "depends_on": []},
    {"worker": "coder", "task": "...", "depends_on": [0]}
  ]
}
```

Router/Dispatcher Pattern

A router classifies incoming requests and sends each one to the most appropriate specialist agent. This pattern is essential for customer-facing applications where queries span multiple domains.

```
SPECIALISTS = {
    "billing": Agent(system="You are a billing specialist...", tools=[...]),
    "technical": Agent(system="You are a technical support engineer...", tools=[...]),
```

```
    "sales": Agent(system="You are a sales representative...", tools=[...]),
    "general": Agent(system="You are a helpful assistant...", tools=[...]),
}

async def route_request(query: str):
    # Lightweight classification
    classification = await classifier.run(
        f"Classify this request into one of: billing, technical, sales, general\n"
        f"Request: {query}\n"
        f"Category:"
    )
    category = classification.strip().lower()
    agent = SPECIALISTS.get(category, SPECIALISTS["general"])
    return await agent.run(query)
```

Debate/Consensus Pattern

The debate pattern forces agents to challenge each other's reasoning, producing more robust answers for high-stakes decisions. A judge agent synthesizes the strongest arguments from both sides into a final answer.

```
Round 1:  Agent A argues position -> Agent B critiques
Round 2:  Agent B argues position -> Agent A critiques
Round 3:  Judge agent synthesizes best answer
```

Implementation

```
async def debate(question: str, rounds: int = 2):
    agent_a_history = []
    agent_b_history = []

    for round_num in range(rounds):
        # Agent A argues
        a_input = f"Question: {question}\n"
        if agent_b_history:
            a_input += f"Opponent's last argument:\n{agent_b_history[-1]}\n"
        a_input += "Present your argument:"

        a_response = await agent_a.run(a_input)
        agent_a_history.append(a_response)

        # Agent B argues
        b_input = f"Question: {question}\n"
        b_input += f"Opponent's argument:\n{a_response}\n"
        b_input += "Present your counter-argument:"

        b_response = await agent_b.run(b_input)
        agent_b_history.append(b_response)

    # Judge synthesizes
    judge_input = (
        f"Question: {question}\n\n"
        f"Arguments from side A:\n{chr(10).join(agent_a_history)}\n\n"
        f"Arguments from side B:\n{chr(10).join(agent_b_history)}\n\n"
        f"Synthesize the best answer, incorporating the strongest points from both sides."
```

```
    )
    return await judge.run(judge_input)
```

Shared Memory

Without shared memory, each agent operates in isolation and cannot benefit from what other agents have already discovered. The right shared state architecture is what turns a collection of agents into a cohesive team.

Memory Architecture

Memory Type	Scope	Persistence	Use Case
Task context	Single task	In-memory	Passing data between agents
Shared scratchpad	All agents in workflow	Session-scoped	Collaborative work
Long-term memory	Cross-session	Database	Learning, preferences
Knowledge graph	Cross-session	Database	Structured facts

Shared State Implementation

```
from dataclasses import dataclass, field

@dataclass
class SharedState:
    """Shared memory accessible by all agents in the workflow."""
    task: str = ""
    plan: list[str] = field(default_factory=list)
    findings: dict[str, str] = field(default_factory=dict)
    artifacts: dict[str, str] = field(default_factory=dict)
    messages: list[dict] = field(default_factory=list)
    status: str = "in_progress"

    def add_finding(self, agent: str, key: str, value: str):
        self.findings[key] = value
        self.messages.append({
            "agent": agent, "action": "finding",
            "key": key, "summary": value[:200]
        })

    def get_context_for_agent(self, agent_name: str) -> str:
        """Generate a context summary relevant to this agent."""
        recent = self.messages[-10:]  # Last 10 messages
        return (
            f"Task: {self.task}\n"
            f"Plan: {self.plan}\n"
            f"Recent activity:\n" +
            "\n".join(f"- [{m['agent']}] {m['action']}: {m['summary']}"
                      for m in recent)
        )
```

Quality Gates

Without quality gates, a bad output from one agent cascades through the entire pipeline and corrupts the final result. Gates at each stage catch errors early when they are cheapest to fix.

Gate	When	Check	Action on Fail
Input validation	Before processing	Format, length, scope	Reject with message
Plan review	After planning	Completeness, feasibility	Re-plan
Subtask output	After each worker	Quality score, relevance	Retry or reassign
Aggregation check	After combining	Consistency, completeness	Revise
Final review	Before delivery	All criteria	Revise or escalate

Quality Gate Implementation

```
async def quality_gate(output: str, criteria: str, threshold: float = 0.7):
    """Evaluate output quality. Returns (passed, score, feedback)."""
    evaluation = await evaluator.run(
        f"Rate this output on a scale of 0-1 for: {criteria}\n\n"
        f"Output:\n{output}\n\n"
        f"Respond as JSON: {{\"score\": 0.X, \"feedback\": \"...\"}}"
    )
    result = json.loads(evaluation)
    passed = result["score"] >= threshold
    return passed, result["score"], result["feedback"]

# Usage in pipeline
draft = await writer.run(task)
passed, score, feedback = await quality_gate(draft, "accuracy and completeness")
if not passed:
    draft = await writer.run(f"Revise based on feedback: {feedback}\n\nOriginal:\n{draft}")
```

Human-in-the-Loop

Fully autonomous multi-agent systems will eventually encounter tasks they cannot handle or decisions they should not make alone. Designing explicit human touchpoints prevents agents from taking irreversible bad actions.

Interaction Points

Point	Trigger	Human Action
Approval gate	Before destructive action	Approve / reject / modify
Escalation	Agent confidence below threshold	Take over or guide
Review	After draft/plan generation	Edit, approve, or request revision
Disambiguation	Ambiguous user request	Clarify intent

Point	Trigger	Human Action
Exception handling	Agent encounters unknown scenario	Provide instructions

Escalation Pattern

```
async def agent_with_escalation(query: str, confidence_threshold: float = 0.6):
    response = await agent.run(query)

    # Self-assessed confidence
    confidence = await evaluator.run(
        f"Rate your confidence in this response (0-1):\n{response}"
    )

    if float(confidence) < confidence_threshold:
        # Escalate to human
        human_input = await request_human_review(
            query=query,
            agent_response=response,
            confidence=float(confidence),
            reason="Low confidence - please review or provide guidance"
        )

        if human_input.action == "approve":
            return response
        elif human_input.action == "override":
            return human_input.response
        elif human_input.action == "guide":
            return await agent.run(
                f"Original query: {query}\n"
                f"Human guidance: {human_input.guidance}\n"
                f"Please revise your response."
            )

    return response
```

Framework Comparison for Multi-Agent

Not every framework supports every pattern. Choose based on the specific patterns you need, your language preference, and whether built-in state management and human-in-the-loop support matter to you.

Framework	Pattern Support	State Management	Human-in-Loop
LangGraph	All (graph-based)	Built-in checkpointing	Yes
CrewAI	Sequential, Hierarchical	Shared memory	Yes
AutoGen	Conversation, Debate	Chat history	Yes
Agents SDK (OpenAI)	Handoffs, Router	Thread-based	Via tool
Mastra	Workflow-based	Built-in	Yes

When to Use Multi-Agent

The biggest mistake in multi-agent design is reaching for it when a single agent would suffice. Multi-agent systems add latency, cost, and debugging complexity -- only use them when the task genuinely demands it.

Use Multi-Agent When

- Task requires genuinely different expertise areas
- Subtasks can be parallelized for speed
- You need checks and balances (reviewer, critic)
- Single agent context window is insufficient
- Different stages need different tools or models

Do NOT Use Multi-Agent When

- A single well-prompted agent can handle the task
- The overhead of coordination exceeds the benefit
- Tasks are tightly coupled and hard to decompose
- Latency is critical (agent-to-agent adds latency)
- Debugging simplicity is more important than capability

Common Pitfalls

Multi-agent systems multiply the failure modes of single agents by the number of agents and their interactions. These pitfalls are responsible for most multi-agent project failures.

Pitfall	Problem	Fix
Over-engineering	Simple task + 5 agents = slow and expensive	Start with single agent, add more only when needed
No quality gates	Bad output from one agent cascades	Add review steps between agents
Shared state conflicts	Agents overwrite each other	Use structured state with atomic updates
No max iteration limit	Infinite revision loops	Set hard limits on retries (2-3)
Missing context	Agent lacks info from prior stages	Pass relevant shared state, not just last output
Ignoring cost	Multi-agent multiplies LLM calls	Track total cost, use cheaper models for simple tasks
No observability	Cannot debug agent interactions	Log every agent call, state transition
No fallback	Multi-agent system fails completely	Graceful degradation to simpler agent or human