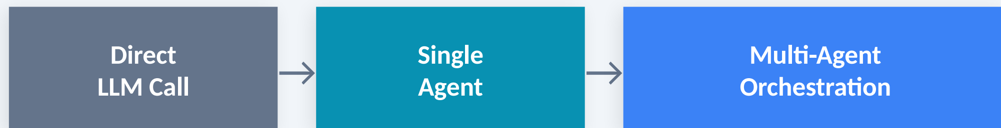
04

Multi-Agent Orchestration

Five patterns for coordinating multiple agents

5 Orchestration Patterns

Start simple. Add complexity only when needed.



Sequential

A → B → C
Pipeline

Concurrent

A,B,C parallel
→ Aggregator

Group Chat

Shared thread
Debate & reflect

Handoff

Triage → Route
to specialist

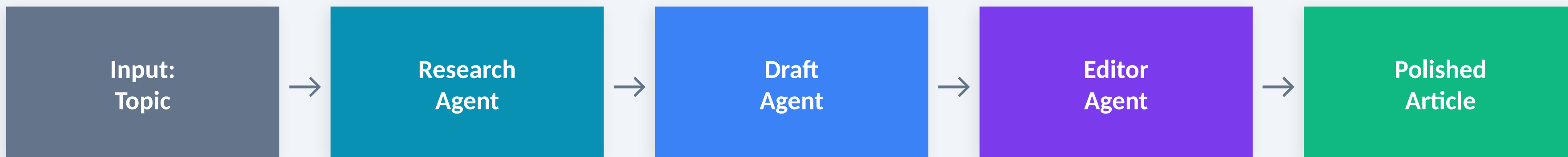
Magentic

Manager + Ledger
+ Workers

The biggest practical challenge: how conversation context flows between agents. Each pattern handles this differently — fresh context, shared thread, structured objects, or task-specific context.

Sequential: Content Pipeline

Agents work in a chain — each agent's output becomes the next agent's input.



When to Use

- Clear, ordered stages
- Each stage refines the previous output
- Different persona per stage

Context Strategy: Fresh per Stage

Each agent gets a clean conversation with ONLY the previous agent's output. No 'context pollution' from earlier stages' internal reasoning. Token-efficient since each call is small.

Sequential — Decision Guide

✓ When to Use

Every step depends on the outcome of the previous one

The workflow has a single, linear path

You need tight control and clear ordering

Progressive refinement: draft → review → polish

★ Why Choose It

Predictable, debuggable flow

Great for compliance or risk check processes

Ensures no step is skipped

Token-efficient: fresh context per stage

✗ When NOT to Use

Only a few stages a single agent can handle

Agents need to collaborate or see each other's reasoning

Dynamic routing based on intermediate results needed

Steps can run independently (use Concurrent instead)

➤ Example

Loan Approval Pipeline

Verify applicant info → Evaluate credit → Approve or reject

Exercise: Content Pipeline Code

`exercises/04_sequential/01_content_pipeline.py`

3 Agents, Each with a Focused Prompt

```
research_agent = Agent(
    name="Research Agent",
    system_prompt=RESEARCH_PROMPT, model=model)
draft_agent = Agent(
    name="Draft Writer",
    system_prompt=DRAFT_PROMPT, model=model)
editor_agent = Agent(
```

Fresh Context at Each Stage

```
# Stage 1: Research (own messages)
research_msgs = [{"role": "user",
  "content": f"Research: {TOPIC}"}]
research_out = run(research_agent,
    research_msgs, client)
# Stage 2: Draft (FRESH messages!)
draft_msgs = [{"role": "user",
```

Context Flow: Only Output Passes Forward

Research Agent
Gets: topic string
Produces: research notes

Draft Writer
Gets: research notes only
No research agent reasoning

Editor Agent
Gets: draft article only
No research or draft reasoning

log_context_pass() makes the handoff visible in console output.

Concurrent — Decision Guide

✓ When to Use

Multiple agents can work on a problem simultaneously
Tasks do not depend on one another
Need multiple perspectives (creative, legal, quality)
Speed matters — you want faster throughput

★ Why Choose It

Faster throughput via parallel execution
Each agent provides independent evaluations
Easy to scale horizontally
Ideal for high-volume or time-sensitive workflows

✗ When NOT to Use

Agents need to build on each other's work
Tasks require a specific order or are deterministic
Resource constraints (e.g., model quota limits)
Results require sequential refinement

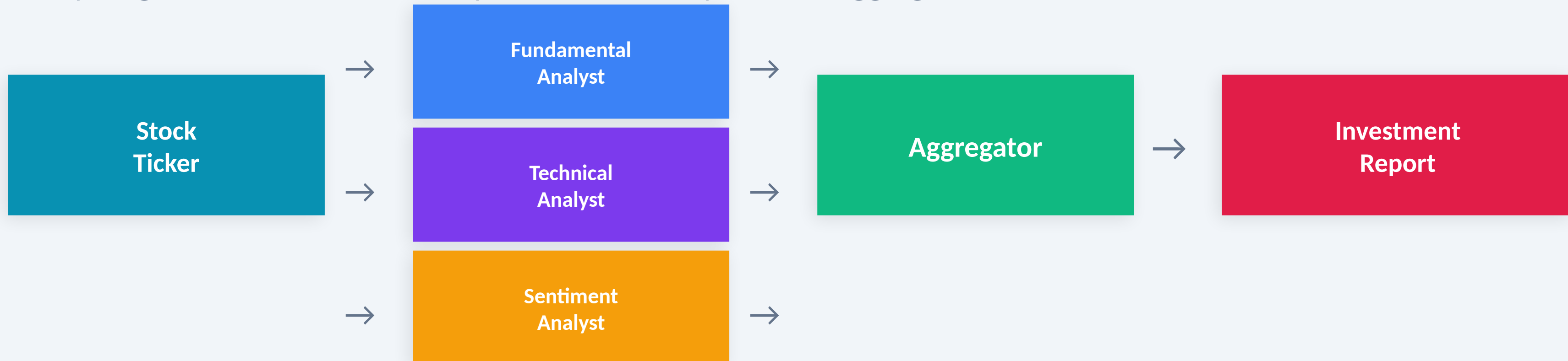
➤ Example

Marketing Campaign Review

Branding + Legal + Compliance + Quality — all review in parallel

Concurrent: Stock Analysis

Multiple agents work on the same input simultaneously, then an aggregator combines results.



When to Use

- Multiple independent analyses of same data
- Each analysis benefits from a different perspective
- Analyses can run without depending on each other

Context Strategy: Isolated per Thread

Each agent gets the SAME initial input independently. No shared state between parallel agents. The aggregator receives ALL outputs combined. Uses ThreadPoolExecutor for real parallelism.

Exercise: Concurrent Stock Analysis Code

`exercises/05_concurrent/01_stock_analysis.py`

Fan-Out: ThreadPoolExecutor

```
def run_analyst(name, prompt, query, model):
    """Each thread gets its own client!"""
    agent = Agent(name=name,
        system_prompt=prompt, model=model)
    messages = [{"role": "user",
                 "content": query}]
    with get_client() as client:
        output = run(agent, messages, client)
    return name, output
```

Fan-In: Aggregate Results

```
# Combine all outputs into one context
combined = "\n\n".join(
    f"=== {name} ===\n{output}"
    for name, output in results.items()
)

aggregator = Agent(
    name="Aggregator",
    system_prompt=AGGREGATOR_PROMPT,
    model=model)
```

Key Implementation Details

Each thread creates its own get_client() — OpenAI clients are not thread-safe.
as_completed() processes results as they arrive, not in submission order. No shared state between analysts.

Group Chat — Decision Guide

✓ When to Use

Multiple agents need to reason together
Complex problems require back-and-forth refinement
No single agent has all the context or expertise
Final answer emerges from collaboration, not strict order

★ Why Choose It

Higher-quality insights through discussion
Great for ambiguous or exploratory tasks
Agents can challenge and validate each other's ideas
Flexible — not locked into sequential or parallel

✗ When NOT to Use

You need strict, step-by-step processing
Problem requires complex dynamic planning
Discussion could go off the rails without clarity
Token budget is tight (shared history grows fast)

➤ Example

Product Strategy Planning

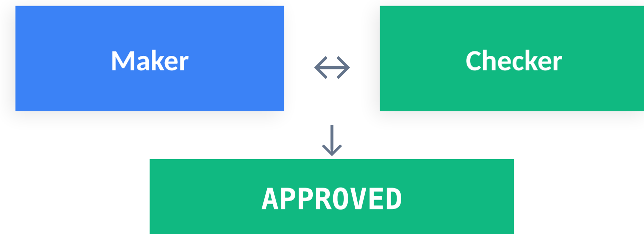
Engineering + Design + Marketing + Business → Collaborative Alignment

Group Chat: Brainstorm & Maker-Checker

Agents share a conversation and take turns contributing, building on each other's messages.

Brainstorm (Multi-Agent)

Maker-Checker (Two-Agent)



Context Strategy: Shared Messages List

All agents share the SAME conversation thread. Later agents see the full history. Token usage grows with every turn. Maker-checker variant loops until 'APPROVED' keyword appears.

Sequential — Decision Guide

✓ When to Use

Every step depends on the outcome of the previous one

The workflow has a single, linear path

You need tight control and clear ordering

Progressive refinement: draft → review → polish

★ Why Choose It

Predictable, debuggable flow

Great for compliance or risk check processes

Ensures no step is skipped

Token-efficient: fresh context per stage

✗ When NOT to Use

Only a few stages a single agent can handle

Agents need to collaborate or see each other's reasoning

Dynamic routing based on intermediate results needed

Steps can run independently (use Concurrent instead)

➤ Example

Loan Approval Pipeline

Verify applicant info → Evaluate credit → Approve or reject

Concurrent — Decision Guide

✓ When to Use

Multiple agents can work on a problem simultaneously
Tasks do not depend on one another
Need multiple perspectives (creative, legal, quality)
Speed matters — you want faster throughput

★ Why Choose It

Faster throughput via parallel execution
Each agent provides independent evaluations
Easy to scale horizontally
Ideal for high-volume or time-sensitive workflows

✗ When NOT to Use

Agents need to build on each other's work
Tasks require a specific order or are deterministic
Resource constraints (e.g., model quota limits)
Results require sequential refinement

➤ Example

Marketing Campaign Review

Branding + Legal + Compliance + Quality — all review in parallel

Group Chat — Decision Guide

✓ When to Use

Multiple agents need to reason together

Complex problems require back-and-forth refinement

No single agent has all the context or expertise

Final answer emerges from collaboration, not strict order

★ Why Choose It

Higher-quality insights through discussion

Great for ambiguous or exploratory tasks

Agents can challenge and validate each other's ideas

Flexible — not locked into sequential or parallel

✗ When NOT to Use

You need strict, step-by-step processing

Problem requires complex dynamic planning

Discussion could go off the rails without clarity

Token budget is tight (shared history grows fast)

➤ Example

Product Strategy Planning

Engineering + Design + Marketing + Business → Collaborative Alignment