# AI Agents

## Technical Guide

Prepared by

**Kadharmoideen Fadurudeen**

# In this guide

2 articles · ~25 min total reading

AI Agents June 8, 2026 11 min read

# Building a Multi-Agent System from Scratch: Architecture Patterns

Orchestrator-worker, blackboard, and pipeline patterns for coordinating multiple AI agents — with the failure modes that only show up once you go past a single-agent demo.

A single well-tuned agent is easy to demo. Give it a clear goal, a handful of tools, and a generous system prompt, and it'll look impressive in a five-minute walkthrough. I wrote about exactly that kind of agent in an earlier post — a single agent that turned natural-language restaurant queries into structured search filters. It worked well because the problem had one shape: understand the query, call some tools, format the answer.

Multi-agent systems are a different animal. The moment you have more than one agent, you've introduced a coordination problem on top of the reasoning problem, and coordination is where almost all of the real engineering effort goes. Nobody warns you about this in the demos, because demos are built to succeed once, not to run in production for six months against inputs nobody anticipated.

This post is about the three architecture patterns I keep coming back to when splitting work across multiple agents, and — more importantly — the failure modes that only show up once you're past a single-agent demo and into something that has to run unattended.

### Why split into multiple agents at all?

Mostly for the same reason you split code into functions: a single agent juggling retrieval, domain reasoning, validation, and formatting in one prompt tends to do all four things a little worse than four agents each doing one thing well. Narrower responsibility means a shorter, more focused system prompt, which means more predictable behavior and easier evaluation.

## Pattern 1: Orchestrator-Worker

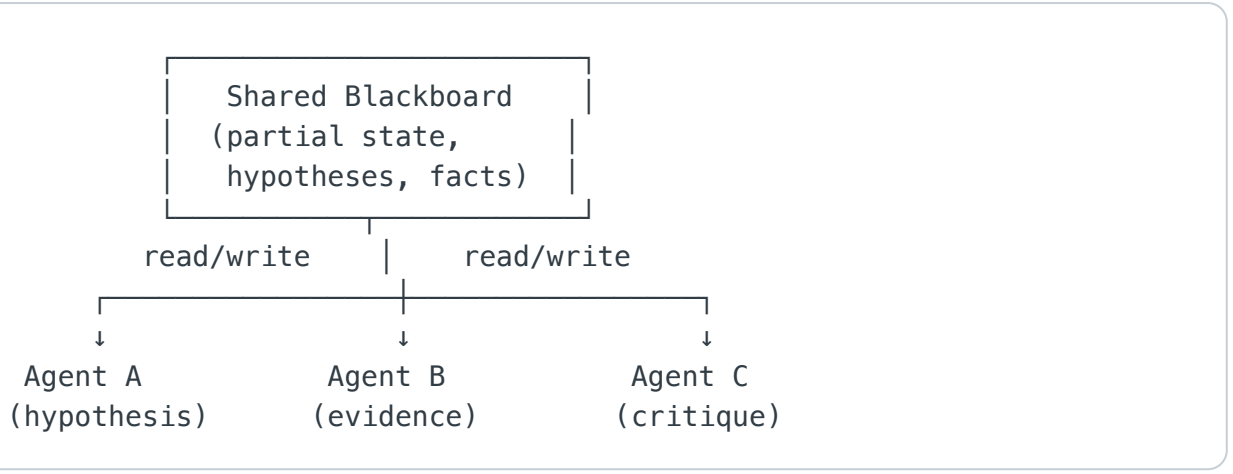
This is the pattern most people reach for first, and for good reason — it maps cleanly onto how a human manager delegates work. One orchestrator agent receives the task, breaks it into subtasks, dispatches each subtask to a specialized worker agent, and aggregates the results into a final answer. Workers don't talk to each other; they only talk to the orchestrator.

```
Task → Orchestrator
              ├→ Worker A (research)
              ├→ Worker B (code generation)
              └→ Worker C (validation)
                     ↓
            Orchestrator aggregates → Final answer
```

This pattern fits when the task decomposition is clear upfront — you know in advance that answering the question requires "look something up," "generate something," and "check it," even if you don't know the specifics until runtime. The orchestrator's only job is planning and aggregation, which keeps its prompt small and its behavior predictable. The tradeoff is that the orchestrator becomes a bottleneck: every piece of context has to flow through it, and if it misjudges the decomposition, every worker downstream inherits that mistake.

## Pattern 2: Blackboard

The blackboard pattern drops the strict hierarchy. Instead of a central orchestrator directing traffic, agents read from and write to a shared state store — the "blackboard" — and each agent decides for itself when it has something useful to contribute. There's usually a lightweight scheduler that decides which agent gets to act next based on what's currently on the blackboard, but there's no single agent that owns the plan.




I reach for this pattern when the solution path genuinely isn't known ahead of time — debugging a subtle production issue, for instance, where one agent proposes a hypothesis, another gathers evidence for or against it, and a third critiques the reasoning. Nobody could have written that as a fixed sequence of steps beforehand, because the next useful action depends entirely on what's already been discovered.

The cost of that flexibility is that blackboard systems are much harder to reason about and to test. There is no single place where "the plan" lives, so debugging a bad outcome means reconstructing the entire sequence of reads and writes after the fact — closer to debugging a race condition than debugging a function call stack.

## Pattern 3: Pipeline

The pipeline pattern is the simplest of the three, and it's the one I default to whenever the workflow is well-understood and repeatable. A fixed sequence of specialized agents, each handing its output to the next: retrieve, extract, validate, format. No delegation logic, no shared scratch space — just a chain.

```
Input → Retrieve Agent → Extract Agent → Validate Agent → Format Agent
```

Pipelines are the easiest of the three patterns to evaluate, because each stage has a well-defined input and output contract, which means you can test and version each agent independently. The obvious limitation is rigidity: if a task doesn't decompose into a fixed sequence — if step three sometimes needs to loop back and redo step one — a pipeline will either fail outright or you'll end up bolting ad hoc branching logic onto it until it quietly turns into an orchestrator with extra steps.

**Picking a pattern**

If you can write the decomposition down as a checklist before you start, use a pipeline. If the decomposition is knowable but the specifics depend on the input, use orchestrator-worker. If you can't know the solution path until agents start finding things out, use a blackboard — and budget more time for evaluation.

## ⚠ Failure Modes You Won't See in a Demo

Every one of these showed up for me only after moving past a controlled demo into something running against real, messy inputs.

### Infinite delegation loops

Two agents that can each decide "this isn't my job, pass it back" will occasionally pass a task back and forth indefinitely. An orchestrator delegates to a worker; the worker decides it needs clarification and delegates back to the orchestrator; the orchestrator, seeing an unclear request, delegates it right back out. Neither agent is wrong in isolation — the loop only exists at the system level, which is exactly why it's invisible when you're testing agents individually.

### Runaway cost from re-invocation

Every agent-to-agent call is a model call, and model calls cost money and time. A blackboard system with three agents each re-checking the board after every write can spiral into dozens of model calls for a task that a human would resolve in three steps. This is easy to miss in development, where you're running one task at a time and eyeballing the transcript. It's very much not easy to miss on the invoice.

### Conflicting concurrent writes

When two agents can act on shared state at the same time — two workers both trying to update the same record, two blackboard agents both trying to "resolve" the same open question — you get the same class of bug as an unguarded race condition in concurrent code, except the two writers are language models instead of threads, which makes it much harder to reproduce reliably.

### Error propagation between agents

The most insidious one. If Agent A hallucinates a fact and passes its output to Agent B, Agent B has no way to distinguish that hallucination from a verified fact — it just sees text in its context window and reasons from it as if it were true. In a single-agent system, a hallucination is a wrong answer. In a multi-agent system, a hallucination is an input that gets compounded by every downstream agent that trusts it.

## Guardrails That Actually Work

None of the failure modes above are solved by "better prompting." They're solved by hard limits enforced in code, outside the agents' control.

#### Hard caps on delegation depth

Track how many times a task has been re-delegated and refuse further delegation past a fixed limit — typically 3 to 5 hops in my experience. When the cap is hit, escalate rather than loop.

#### A token or cost budget per task

Set a hard ceiling on total tokens or dollars spent per task, tracked centrally, not per agent. When a task hits the ceiling, it stops — it does not get "one more try."

#### A defined human-escalation path

When confidence is low or a cap is hit, the system needs a real exit that isn't "let the agents keep trying" — surface the partial state to a human, don't let the system quietly retry into a low-quality answer it presents as confident.

## The Honest Summary

Multi-agent architecture buys you task decomposition — the ability to break a large, fuzzy problem into smaller, more tractable pieces that are each easier to prompt, test, and reason about. It does not, on its own, buy you correctness. A system of five agents that each hallucinate 5% of the time doesn't average out to more reliable behavior; it compounds. The pattern you choose determines how the work is split. The

evaluation harness and the guardrails you build around it determine whether the system is safe to actually put in front of users. Spend your engineering time accordingly.

AI Agents October 13, 2025 14 min read

# Building Agentic Workflows for Restaurant Discovery

Learn how to create intelligent AI agents that help users discover nearby restaurants based on specific queries like lunch buffets, continental breakfast, and more using modern AI frameworks.

Discover how to build intelligent AI agents that help users find the perfect restaurant based on specific queries like "lunch buffet near me" or "continental breakfast downtown" using modern AI frameworks and location-based services.

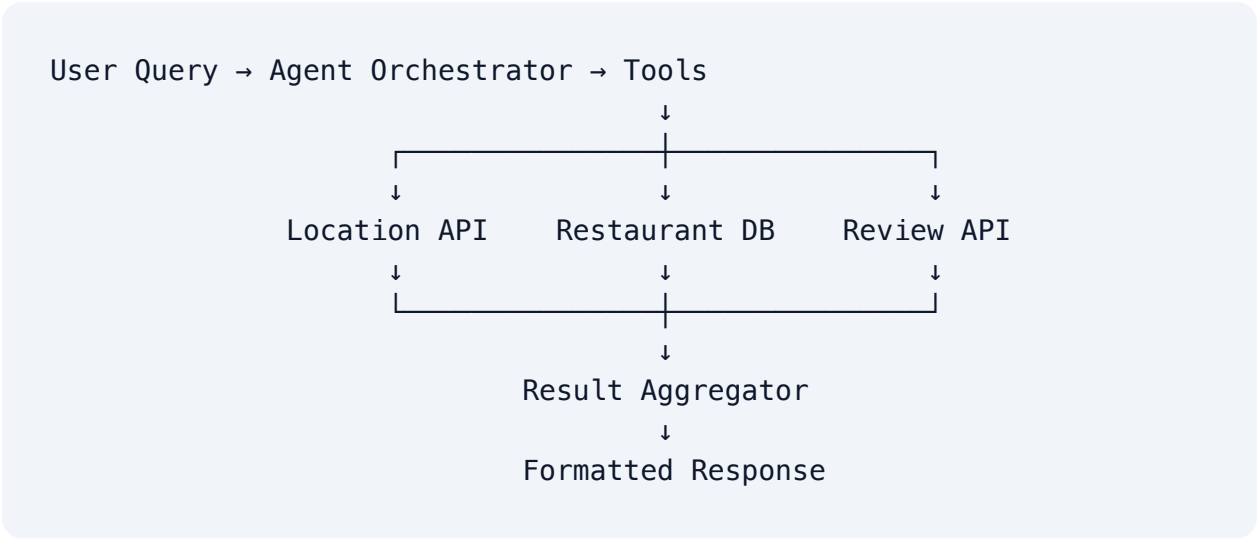
### What Are Agentic Workflows?

Agentic workflows are AI-powered systems that can autonomously plan, execute, and adapt their actions to achieve specific goals. Unlike traditional chatbots, agents can break down complex queries, use multiple tools, and make decisions based on context.

## The Restaurant Discovery Challenge

Finding the right restaurant involves multiple factors: location, cuisine type, meal timing, price range, dietary restrictions, and specific amenities like buffets or outdoor seating. Traditional search requires users to manually filter through multiple criteria. An agentic workflow can understand natural language queries and orchestrate multiple API calls to deliver precise results.

## Architecture Overview




## Example Query Scenarios

### Continental Breakfast

"Find hotels or cafes serving continental breakfast within 2 miles"

- Filter: Breakfast hours (6 AM - 11 AM)
- Cuisine: Continental, European
- Amenities: Coffee, pastries, buffet

### Lunch Buffet

"Show me Indian restaurants with lunch buffet under $20"

- Filter: Lunch hours (11 AM - 3 PM)
- Cuisine: Indian
- Price: $ - $$
- Feature: All-you-can-eat buffet

## Building the Agent

We'll use the AI SDK with tool calling capabilities to create an agent that can understand queries and use multiple tools to find restaurants.

### Step 1: Define the Tools

```
import { tool } from 'ai';
import { z } from 'zod';

const getLocationTool = tool({
  description: 'Get user location coordinates',
  parameters: z.object({
    address: z.string().describe('Address or "current location"'),
  }),
  execute: async ({ address }) => {
    // Use geocoding API
    const coords = await geocode(address);
    return { lat: coords.lat, lng: coords.lng };
  },
});

const searchRestaurantsTool = tool({
  description: 'Search restaurants by criteria',
  parameters: z.object({
    location: z.object({
      lat: z.number(),
      lng: z.number(),
    }),
    cuisine: z.string().optional(),
    mealType: z.enum(['breakfast', 'lunch', 'dinner']).optional(),
    features: z.array(z.string()).optional(),
    priceRange: z.enum(['$', '$$', '$$$', '$$$$']).optional(),
    radius: z.number().default(5), // miles
  }),
  execute: async (params) => {
    // Query restaurant database/API
    const results = await queryRestaurants(params);
    return results;
  },
});
```

### Step 2: Create the Agent

```
import { generateText } from 'ai';

async function findRestaurants(query: string) {
  const { text, toolCalls } = await generateText({
    model: 'gpt-4',
    prompt: `You are a restaurant discovery assistant.
    Help the user find restaurants based on their query: "${query}"

    Use the available tools to:
    1. Get the user's location
    2. Search for restaurants matching their criteria
    3. Provide detailed recommendations`,
    tools: {
      getLocation: getLocationTool,
      searchRestaurants: searchRestaurantsTool,
    },
    maxSteps: 5, // Allow multi-step reasoning
  });

  return { recommendations: text, toolCalls };
}
```

### Step 3: Handle Complex Queries

```
// Example: "Find Indian lunch buffets under $20 near downtown"

const result = await findRestaurants(
  "Find Indian lunch buffets under $20 near downtown Seattle"
);

// Agent reasoning:
// 1. Extract location: "downtown Seattle"
// 2. Call getLocation({ address: "downtown Seattle" })
// 3. Extract criteria: cuisine=Indian, mealType=lunch,
//    features=["buffet"], priceRange="$-$$"
// 4. Call searchRestaurants with all parameters
// 5. Format and return top recommendations
```

## Integrating Location Services

Location is crucial for restaurant discovery. Here's how to integrate Google Places API or similar services:

```
import { Client } from '@googlemaps/google-maps-services-js';

const client = new Client({});

async function searchNearbyRestaurants(params) {
  const response = await client.placesNearby({
    params: {
      location: { lat: params.lat, lng: params.lng },
      radius: params.radius * 1609.34, // miles to meters
      type: 'restaurant',
      keyword: params.features?.join(' '),
      key: process.env.GOOGLE_MAPS_API_KEY,
    },
  });

  // Filter by meal type and price
  const filtered = response.data.results.filter(place => {
    const matchesMealType = checkMealType(place, params.mealType);
    const matchesPrice = place.price_level <= getPriceLevel(params.priceR
    return matchesMealType && matchesPrice;
  });

  return filtered.map(place => ({
    name: place.name,
    address: place.vicinity,
    rating: place.rating,
    priceLevel: '$'.repeat(place.price_level || 1),
    isOpen: place.opening_hours?.open_now,
  }));
}
```

## Advanced Features

### Time-Aware Filtering

Automatically filter restaurants based on current time and meal periods:

- Breakfast: 6 AM - 11 AM
- Lunch: 11 AM - 3 PM
- Dinner: 5 PM - 10 PM
- Late night: 10 PM - 2 AM

### ☆ Review Integration

Integrate with Yelp or Google Reviews to provide sentiment analysis and highlight specific features mentioned in reviews (e.g., "great buffet selection", "authentic continental breakfast").

### $ Dynamic Pricing

Track special offers, happy hour deals, and lunch specials to provide cost-effective recommendations.

## Example User Interaction

User:

"I want a lunch buffet with Indian food, preferably under $15"

Agent:

I found 3 Indian restaurants with lunch buffets near you:

**Taj Palace**
0.8 miles • $$ • ☆ 4.5 (230 reviews)
Lunch buffet: $12.99 • Open until 3 PM • Vegetarian options

**Spice Garden**
1.2 miles • $ • ☆ 4.3 (180 reviews)
Lunch buffet: $10.99 • Open until 2:30 PM • Extensive buffet selection

**Curry House**
1.5 miles • $$ • ☆ 4.4 (195 reviews)
Lunch buffet: $13.99 • Open until 3 PM • Popular for tandoori dishes

## Best Practices

1. **Cache Location Data**
   Store user location preferences to avoid repeated geocoding API calls.

2 **Handle Ambiguity**

When queries are unclear, ask clarifying questions before making API calls.

3 **Optimize API Usage**

Batch requests and implement rate limiting to stay within API quotas.

4 **Provide Fallbacks**

If no exact matches are found, suggest nearby alternatives or broaden the search criteria.

## Conclusion

Agentic workflows transform restaurant discovery from a manual, multi-step process into an intelligent, conversational experience. By combining natural language understanding, tool orchestration, and location-based services, you can build systems that truly understand user intent and deliver personalized recommendations. The key is designing agents that can reason about complex queries, use multiple data sources, and adapt to user preferences over time.