Multi-Agent Coordination Patterns (MCP)

1. Quick definition

Multi-Agent Coordination Patterns (MCP) are reusable architectural patterns describing how multiple autonomous agents communicate, divide work, coordinate decisions, and reach a joint outcome. MCPs capture proven ways to organize agent interactions so the whole system behaves reliably, efficiently, and predictably as it scales. Good patterns help you pick the right trade-offs between decentralization, latency, robustness, explainability, and cost.

---

2. Core coordination dimensions (how to choose a pattern)

When choosing or designing an MCP, ask:

- **Control model:** centralized (one orchestrator) vs decentralized (peer-to-peer).
- **Communication style:** synchronous RPC, asynchronous messaging, or pub/sub.
- **Coordination tightness:** do agents need tightly synchronized steps or loose eventual coordination?
- **Task allocation:** static assignment, negotiated (auction), or emergent (local rules).
- **Fault model & trust:** can the system tolerate agent failure or malicious agents?
- **Latency & scale:** real-time control (low latency) vs batch/analytical workflows.

---

3. Common Multi-Agent Coordination Patterns

A. Centralized Orchestrator (Leader–Worker)

What: One central orchestrator agent plans, assigns tasks to worker agents, collects results, and makes global decisions.
When to use: When global optimization or consistent policy is required (one source of truth).
Pros: simpler global reasoning, easier to audit and enforce policies, easier to debug.
Cons: single point of failure, potential scaling bottleneck, higher latency if orchestrator overloaded.

Example — Hospital surgery scheduling: A central Scheduling Agent receives incoming surgery requests and assigns OR slots and staff (worker agents). The orchestrator can run global optimization (minimize staff idle time, respect constraints). If the scheduler fails, the whole pipeline stalls — so redundancy or failover is needed.

B. Decentralized Negotiation / Market (Auction & Bidding)

What: Agents bid for tasks; a marketplace mechanism (auctions) allocates tasks to winners.
When to use: When tasks are naturally divisible and agents have private utility functions (e.g., fleets with varying costs).
Pros: scalable, economic fairness, adaptation to local costs.
Cons: requires good bid protocols, can be suboptimal versus global optimization, vulnerable to strategic behavior.

Example — Delivery fleet routing: Each delivery vehicle (agent) bids on delivery tasks based on remaining capacity and location; a dispatcher clears auctions to allocate parcels efficiently.

C. Blackboard (Shared Workspace)

What: A shared memory (blackboard) where agents post partial results, hypotheses, or observations. Other agents read and augment the shared state. Coordination emerges via the shared workspace.
When to use: Multi-stage reasoning or situations where different specialists contribute complementary knowledge (e.g., diagnostics).
Pros: flexible collaboration, supports asynchronous contributors, good for iterative refinement.
Cons: contention or write conflicts, potential staleness, needs governance for consistency.

Example — Clinical diagnosis pipeline: An Imaging Agent posts candidate lesions to the blackboard; a Lab Agent adds test correlations; a Treatment Agent reads combined evidence to propose plans. This supports human clinicians reading the blackboard too. (Blackboard-style coordination is used in complex reasoning systems.) (Medium)

D. Choreography / Peer-to-Peer (Emergent Coordination)

What: Agents follow local rules and react to local observations; global coordination emerges without a central controller.
When to use: Large distributed systems where central control is impractical and local autonomy is desired (e.g., swarms).
Pros: highly resilient, scalable, and robust to single failures.
Cons: harder to reason about emergent behavior, may require extensive simulation to validate.

Example — Traffic microsimulation: Each vehicle agent follows local policies (speed, gap acceptance). Coordinated traffic patterns (waves, platoons) emerge; local V2V messages can improve safety.

E. Hierarchical / Layered Coordination

What: A layered approach where high-level agents make strategic decisions and delegate lower-level tactical tasks to subordinate agents.
When to use: Systems needing both strategic planning and fast reactive control.
Pros: balance between global oversight and local responsiveness, easier to scale via boundaries.
Cons: complexity in interface contracts; delay when top-level decisions change.

Example — Autonomous fleet management: A Fleet Manager agent sets strategic goals (coverage targets), regional Dispatch agents allocate tasks to vehicle controllers.

F. Pipeline / Linear Workflow (Producer–Consumer)

What: Agents are organized in a linear processing pipeline—producer → processor → consumer. Each stage is an agent with a clear contract.
When to use: Structured data processing, ETL, or chained transformations.

Pros: clear responsibilities, easy to scale by scaling stages.
Cons: rigid flow, bottlenecks at slow stages.

Example — Document ingestion: OCR Agent → NLP Extraction Agent → Summarizer Agent → Storage Agent.

MCP examples in domain contexts

Healthcare examples

- **Blackboard:** Multi-modal diagnosis — imaging agent, genomics agent, and EHR-reader agent post evidence to a blackboard; a Treatment Planner reads consolidated evidence and proposes options. Best when different specialists contribute partial knowledge. (Medium)

- **Centralized Orchestrator:** Hospital bed and staff management — one Orchestration Agent enforces global constraints (beds, staff schedules) and assigns tasks to department agents. Good when legal/compliance constraints must be globally enforced.

- **Hybrid:** Telemedicine triage — an initial triage agent collects symptoms (single agent), then a dispatcher markets cases to specialist agents (auction or priority), and a follow-up monitoring agent tracks progress.

Design tip: clinical systems need human-in-the-loop checkpoints and strict logging/audit trails. Use blackboard for reasoning but require approvals for any high-risk actions.

Mobility examples

- **Choreography / Swarm:** Drone swarm area mapping — each drone follows local rules and shares short telemetry to avoid collisions. Use local P2P messages for collision avoidance.

- **Market/Auction:** Last-mile delivery with independent drivers — drivers bid on jobs based on current route and load; the system balances cost and timeliness.

- **Hierarchical + Orchestrator:** City traffic management — local traffic lights operate autonomously for immediate responsiveness; a higher-level Traffic Manager optimizes corridors during major events.

Design tip: mobility systems often need real-time guarantees — place latency-sensitive control at the edge (on-vehicle), and use cloud for coordination and planning.

Customer service examples

- **Pipeline:** Incoming ticket → Intent classification agent → Knowledge retrieval agent → Response generation agent → Escalation agent.

- **Centralized Orchestrator:** High-volume contact centers use a central orchestrator to route messages across bots and human agents, enforce escalation policies, and manage SLAs.

- **Blackboard/Shared memory:** Case management where multiple specialist agents attach notes and partial solutions to a shared case record; a coordinator compiles final replies.

Design tip: keep sensitive identity data segregated and only allow agents with proper authorization to access PII; log every action for compliance.

Azure AI Foundry: Agent as a Service

Agent as a Service (Foundry Agent Service) is a managed runtime within Azure AI Foundry that lets you design, deploy, and operate AI agents with production features out of the box. It ties together models, tool connectors, identity, content safety, observability, and runtime orchestration so developers can focus on agent logic instead of infra plumbing. The agent service manages threads, orchestrates tool calls, enforces content safety, and integrates with Azure identity and monitoring systems.

Key capabilities called out in official docs and product pages include:

- **Model catalog & model composition:** use a range of hosted models (Foundry model catalog) and combine them with tools.

- **Tool & connector ecosystem:** integrate knowledge sources (Bing, Microsoft 365, SharePoint, Azure AI Search) and 1,400+ connectors (Logic Apps) to call actions.

- **Runtime orchestration:** the service orchestrates LLM calls, tool invocations and multi-step plans.

- **Security, identity & compliance:** integrates with Azure AD, network controls, and enterprise observability.

- **Extensibility:** call Azure Functions and serverless workflows as tools from agents.

How Foundry Agent Service supports MCPs

Azure AI Foundry's Agent Service is designed to make the common MCP building blocks simpler:

- **Centralized orchestration:** you can implement a single orchestrator agent in Foundry that calls tactics (functions, connectors) and keeps state — Foundry provides the runtime, scaling, and safety controls.

- **Tool integrations & connectors:** marketplaces and connectors let agents access data sources (SharePoint, Fabric, search) so agents become tool-enabled workers in marketplace/auction patterns.

- **Event-driven coordination:** Foundry integrates with serverless and eventing (Azure Functions, Logic Apps), enabling event-driven choreography and pipeline patterns.

- **Observability & governance:** for MAS deployments, centralized telemetry and audit logs are essential — Foundry integrates with Azure monitoring to track agent decisions and tool calls.

Example architectures (conceptual) using Azure AI Foundry

Below are three compact examples that map MCPs to Foundry components.

Example 1 — Hospital Triage (Blackboard + Orchestrator hybrid)

Goal: Triage incoming patients and route them to the right clinician, escalate emergencies, and manage OR bookings.

Components:

- **Intake Agent (Foundry)**: collects initial data via chat and forms; posts an initial case to a shared case store (blackboard in Azure Storage or a DB).

- **Imaging Agent (Foundry model + tool)**: when imaging arrives, it writes findings to the shared store.

- **Triage Orchestrator Agent (Foundry Agent Service)**: reads shared state, computes priority, assigns specialists (calls hospital scheduling connector via Logic Apps).

- **Human-in-the-loop UI**: clinicians view the blackboard and approve major interventions.

Why Foundry fits: connectors (EHR/SharePoint), model composition (imaging model + LLM for summaries), observability and identity for auditability.

Example 2 — Fleet Coordination for Last-Mile Delivery (Market pattern)

Goal: Assign delivery tasks to drivers minimizing travel time.

Components:

- **Task Broker Agent (Foundry orchestrator)**: posts parcels to a task queue and runs auctions by calling driver agents' endpoints.

- **Driver Agents (edge or API endpoints)**: report status/capacity and compute bids (local costs).

- **Settlement & Routing Agent**: after auction, instructs selected drivers and tracks status; updates dashboards (monitoring via Azure Monitor).

Why Foundry fits: serverless connectors and agent runtime for auctions; integrates telematics and mapping via connectors.

Example 3 — Customer Service Orchestration (Pipeline + Blackboard)

Goal: Automate first-line support, escalate complex issues to human teams with full context.

Components:

- **Frontline Chat Agent (Foundry)**: intent detection and immediate answers via knowledge connectors (SharePoint, Fabric).

- **Knowledge Agent (Foundry)**: retrieves docs and posts suggested answers to a case blackboard.

- **Escalation Orchestrator**: if confidence low, packages context and forwards to human agent with logs and suggested steps.

Why Foundry fits: integrated connectors, model catalog, and tool invocation (e.g., CRM updates via Logic Apps).

Best practices when using Foundry Agent Service

1. **Design for safety & approval:** use human-in-the-loop for high-risk steps (healthcare, finance). Foundry provides safety gates and content filters — use them.

2. **Prefer hybrid architecture:** put latency-sensitive control near the edge (vehicles, devices) and use Foundry for higher-level coordination and planning.

3. **Use standard connectors:** rely on Foundry's connectors for search and enterprise data sources rather than custom scraping; this improves grounding and traceability.

4. **Observability & audits:** log every tool call, decision rationale, and the data version used to derive outputs. Foundry integrates with Azure monitoring and logging stacks to help.

5. **Simulate agent interactions:** before production, run multi-agent simulations (chaos testing) to catch emergent failure modes — industry guidance emphasizes simulation as key for MAS validation.

Limitations & risks

- **Emergent/Unpredictable behavior:** MAS can produce unexpected global states. Simulate and stage rollouts.
- **Security & data governance:** Foundry integrates identity and network controls, but you must design least-privilege access for agents to sensitive data (EHRs, PII).

- **Cost & complexity:** MAS at scale brings operational cost (many agents, telemetry, connectors). Use observability to find hot spots.

- **Model updates & drift:** coordinate model deployment and versioning across many agents; Foundry's model catalog and SDK aim to help with governance.