Chapter 8. From One Agent to Many

Most use cases start with one agent, but as the number of tools increases, and the range of problems you want your agent to solve increases, introducing a multiagent pattern can improve the overall performance and reliability. Just as we saw that it's probably not a good idea to put all of your code in a single file, or bundle all of your backend servers into a single monolith, many of the lessons we learned about the principles of software architecture and service design still apply when building systems with AI and foundation models. As you continue to add functionality and capabilities into your agentic system, you'll soon find the need to break up your system into smaller agents that can be independently validated, tested, integrated, and reused. In this chapter, we'll discuss how and when to add an agent to your system, and how to organize and manage them.

How Many Agents Do I Need?

Begin with a simple approach, and only add complexity as needed to improve performance. The appropriate number and organization of agents will vary enormously based on the difficulty of the tasks, the number of tools, and the complexity of the environment.

Single-Agent Scenarios

We'll begin with single-agent systems, which are suitable for tasks that are of modest difficulty, a limited number of tools, and lower-complexity environments. They are also often better when latency is critical, as multiagent systems typically require multiple exchanges between agents, which increases the latency for the user. As a result, it is typically best practice to begin with a single-agent system, as it is often faster and cheaper than extending to multiagent systems. In this approach, a single agent is responsible for invoking tools, if available, up to a limit before responding to the user. In this, the agent performs tasks and chooses when to invoke tools or submit the answer. The primary benefits include:

Simplicity

Easier implementation and management

Lower resource requirements

Less computational overhead

Latency

Quicker response for users

Single-agent systems offer a strong starting point for building agentic applications. Their simplicity, lower cost, and reduced latency make them well suited for many practical scenarios—especially when the task scope is limited and performance requirements are tight. While they may not scale well to highly complex or multifaceted tasks, starting with a single-agent architecture enables teams to validate core functionality quickly and iterate efficiently. Only when the complexity, toolset, or task coordination needs outgrow the capacity of a single agent should developers consider transitioning to more sophisticated multiagent systems.

To illustrate, consider a single-agent system for supply chain logistics management. This agent handles a broad set of tools for inventory, shipping, and supplier tasks in one unified prompt and graph. While effective for basic queries, performance can degrade with too many tools, as the agent must select from a large set. Here's how we set up a single agent with 16 tools:

```
from __future__ import annotations
"""
supply_chain_logistics_agent.py
LangGraph workflow for a Supply Chain & Logistics Manag
handling inventory management, shipping operations, sup
and warehouse optimization.
"""
import os
import json
import operator
import builtins
from typing import Annotated, Sequence, TypedDict, Opti

from langchain_openai.chat_models import ChatOpenAI
from langchain.schema import AIMessage, BaseMessage, Hu
from langchain_core.messages.tool import ToolMessage
from langchain.callbacks.streaming_stdout import Stream

from langchain.tools import tool
from langgraph.graph import StateGraph, END

from traceloop.sdk import Traceloop
from src.common.observability.loki_logger import log_tc

os.environ["OTEL_EXPORTER_OTLP_ENDPOINT"] = "http://loc
os.environ["OTEL_EXPORTER_OTLP_INSECURE"] = "true"

@tool
def manage_inventory(sku: str = None, **kwargs) -> str:
   """Manage inventory levels, stock replenishment, auc
   and optimization strategies."""
   print(f"[TOOL] manage_inventory(sku={sku}, kwargs={k
   log_to_loki("tool.manage_inventory", f"sku={sku}")
   return "inventory_management_initiated"

@tool
def track_shipments(origin: str = None, **kwargs) -> st
   """Track shipment status, delays, and coordinate del
   print(f"[TOOL] track_shipments(origin={origin}, kwar
   log_to_loki("tool.track_shipments", f"origin={origir
   return "shipment_tracking_updated"

@tool
def evaluate_suppliers(supplier_name: str = None, **kwa
   """Evaluate supplier performance, conduct audits,
   and manage supplier relationships."""
   print(f"[TOOL] evaluate_suppliers(supplier_name={sup
         kwargs={kwargs})")
   log_to_loki("tool.evaluate_suppliers", f"supplier_na
   return "supplier_evaluation_complete"

@tool
def optimize_warehouse(operation_type: str = None, **kw
   """Optimize warehouse operations, layout, capacity,
   print(f"[TOOL] optimize_warehouse(operation_type={op
         kwargs={kwargs})")
   log_to_loki("tool.optimize_warehouse", f"operation_t
   return "warehouse_optimization_initiated"

@tool
def forecast_demand(season: str = None, **kwargs) -> st
   """Analyze demand patterns, seasonal trends, and cre
```

```
    print(f"[TOOL] forecast_demand(season={season}, kwar
    log_to_loki("tool.forecast_demand", f"season={seasor
    return "demand_forecast_generated"

@tool
def manage_quality(supplier: str = None, **kwargs) -> s
    """Manage quality control, defect tracking, and supp
    print(f"[TOOL] manage_quality(supplier={supplier}, k
    log_to_loki("tool.manage_quality", f"supplier={suppl
    return "quality_management_initiated"

@tool
def arrange_shipping(shipping_type: str = None, **kwarg
    """Arrange shipping methods, expedited delivery,
    and multi-modal transportation."""
    print(f"[TOOL] arrange_shipping(shipping_type={shipp
          kwargs={kwargs})")
    log_to_loki("tool.arrange_shipping", f"shipping_type
    return "shipping_arranged"

@tool
def coordinate_operations(operation_type: str = None, *
    """Coordinate complex operations like cross-docking,
    and transfers."""
    print(f"[TOOL] coordinate_operations(operation_type=
          kwargs={kwargs})")
    log_to_loki("tool.coordinate_operations", f"operatic
    return "operations_coordinated"

@tool
def manage_special_handling(product_type: str = None, *
    """Handle special requirements for hazmat, cold chai
    sensitive products."""
    print(f"[TOOL] manage_special_handling(product_type=
          kwargs={kwargs})")
    log_to_loki("tool.manage_special_handling", f"produc
    return "special_handling_managed"

@tool
def handle_compliance(compliance_type: str = None, **kw
    """Manage regulatory compliance, customs, documentat
    and certifications."""
    print(f"[TOOL] handle_compliance(compliance_type={cc
          kwargs={kwargs})")
    log_to_loki("tool.handle_compliance", f"compliance_t
    return "compliance_handled"
```

```
@tool
def process_returns(returned_quantity: str = None, **kw
    """Process returns, reverse logistics, and product c
    print(f"[TOOL] process_returns(returned_quantity={re
          kwargs={kwargs})")
    log_to_loki("tool.process_returns", f"returned_quant
    return "returns_processed"

@tool
def scale_operations(scaling_type: str = None, **kwargs
    """Scale operations for peak seasons, capacity plann
    and workforce management."""
    print(f"[TOOL] scale_operations(scaling_type={scalin
          kwargs={kwargs})")
    log_to_loki("tool.scale_operations", f"scaling_type=
    return "operations_scaled"

@tool
def optimize_costs(cost_type: str = None, **kwargs) ->
    """Analyze and optimize transportation, storage, and
    print(f"[TOOL] optimize_costs(cost_type={cost_type},
    log_to_loki("tool.optimize_costs", f"cost_type={cost
    return "cost_optimization_initiated"

@tool
def optimize_delivery(delivery_type: str = None, **kwar
    """Optimize delivery routes, last-mile logistics,
    and sustainability initiatives."""
    print(f"[TOOL] optimize_delivery(delivery_type={deli
          kwargs={kwargs})")
    log_to_loki("tool.optimize_delivery", f"delivery_typ
    return "delivery_optimization_complete"

@tool
def manage_disruption(disruption_type: str = None, **kw
    """Manage supply chain disruptions, contingency plan
    and risk mitigation."""
    print(f"[TOOL] manage_disruption(disruption_type={di
          kwargs={kwargs})")
    log_to_loki("tool.manage_disruption", f"disruption_t
    return "disruption_managed"

@tool
def send_logistics_response(operation_id: str = None, m
    """Send logistics updates, recommendations, or statu
    to stakeholders."""
    print(f"[TOOL] send_logistics_response → {message}")
    log_to_loki("tool.send_logistics_response", f"operat
                message={message}")
    return "logistics_response_sent"

TOOLS = [
    manage_inventory, track_shipments, evaluate_supplier
    forecast_demand, manage_quality, arrange_shipping, c
    manage_special_handling, handle_compliance, process_
    optimize_costs, optimize_delivery, manage_disruptior
]
```

These tools encompass the core functions of a supply chain agent, from tracking shipments to forecasting demand and managing disruptions. By defining them with the `@tool` decorator in LangChain, we enable the agent to call them dynamically based on the user's query. This setup is straightforward, requiring no complex coordination—the agent simply analyzes the prompt and selects the appropriate tool. For example, a basic agent might handle inventory shortages by invoking `manage_inventory` and `forecast_demand` in sequence, as we'll see in the execution flow.

However, as the toolset expands—here, to 16—the agent's system prompt must describe all possibilities, potentially leading to confusion or suboptimal choices. This is where the single-agent model's limitations begin to show, paving the way for multiagent decomposition. Now, let's complete the agent setup with the foundation model binding, state definition, and graph construction:

```
Traceloop.init(disable_batch=True, app_name="supply_cha
llm = ChatOpenAI(model="gpt-5", temperature=0.0,
                 callbacks=[StreamingStdOutCallbackHandl
                 verbose=True).bind_tools(TOOLS)

class AgentState(TypedDict):
    operation: Optional[dict]  # Supply chain operation
    messages: Annotated[Sequence[BaseMessage], operator.

def call_model(state: AgentState):
    history = state["messages"]

    # Handle missing or incomplete operation data gracef
    operation = state.get("operation", {})
```

```
if not operation:
    operation = {"operation_id": "UNKNOWN", "type":
        "priority": "medium", "status": "active"}

operation_json = json.dumps(operation, ensure_ascii=
system_prompt = (
    "You are an experienced Supply Chain & Logistics
    "Your expertise covers:\n"
    "- Inventory management and demand forecasting\r
    "- Transportation and shipping optimization\n"
    "- Supplier relationship management and evaluati
    "- Warehouse operations and capacity planning\n'
    "- Quality control and compliance management\n"
    "- Cost optimization and operational efficiency\
    "- Risk management and disruption response\n"
    "- Sustainability and green logistics initiative
    "\n"
    "When managing supply chain operations:\n"
    "  1) Analyze the logistics challenge or opportu
    "  2) Call the appropriate supply chain manageme
    "  3) Follow up with send_logistics_response to
    "  4) Consider cost, efficiency, quality, and su
    "  5) Prioritize customer satisfaction and busir
    "\n"
    "Always balance cost with quality and risk mitig
    f"OPERATION: {operation_json}"
)

full = [SystemMessage(content=system_prompt)] + hist

first: ToolMessage | BaseMessage = llm.invoke(full)
messages = [first]

if getattr(first, "tool_calls", None):
    for tc in first.tool_calls:
        print(first)
        print(tc['name'])
        fn = next(t for t in TOOLS if t.name == tc['
        out = fn.invoke(tc["args"])
        messages.append(ToolMessage(content=str(out)

    second = llm.invoke(full + messages)
    messages.append(second)

return {"messages": messages}
```

```
def construct_graph():
    g = StateGraph(AgentState)
    g.add_node("assistant", call_model)
    g.set_entry_point("assistant")
    return g.compile()

graph = construct_graph()

if __name__ == "__main__":
    example = {"operation_id": "OP-12345", "type": "inve
               "priority": "high", "location": "Warehous
    convo = [HumanMessage(content="We're running critica
    Current stock is 50 units but we have 200 units on b
    reorder strategy?")]
    result = graph.invoke({"operation": example, "messag
    for m in result["messages"]:
        print(f"{m.type}: {m.content}")
```

With the agent fully assembled, we see the elegance of a single-node LangGraph: the state holds operation details and messages, the model call analyzes queries and invokes tools, and the graph is minimal—just one “assistant” node. This structure minimizes overhead, ensuring low latency as there’s no inter-agent communication. In practice, as demonstrated in 2025 LangGraph tutorials for supply chain agents, such setups can process queries in under a second on standard hardware, making them ideal for operational dashboards or real-time alerts.

For most use cases, though, the key bottleneck arises when the number of tools and responsibilities increases. When an agent is expected to choose the correct tool from a set, performance degrades as the potential number of tools increases. Before jumping to multiagents, consider scaling within the single-agent framework: for instance, encapsulate multiple tools into larger groupings (e.g., via hierarchical tool selection), or use semantic tool selection using a vector database as described in Chapter 5 on orchestration. If these approaches still fall short, decomposing tools into distinct agents with appropriate responsibilities can then improve reliability and performance, though it introduces coordination overhead.

Multiagent Scenarios

In multiagent systems, multiple agents collaborate to achieve shared goals, an approach that is especially advantageous when tasks are complex and require varied toolsets, parallel processing, or adaptability to dynamic environments. A key benefit of multiagent systems is specialization: each agent can be assigned specific roles or areas of expertise, allowing the system to leverage each agent's strengths effectively. This division of labor enables agents to focus on defined aspects of a task, which improves efficiency and ensures that specialized tools are applied where they are most needed. By distributing tools and responsibilities across agents, multiagent systems address the limitations faced by single-agent systems, especially when tasks require expertise across different domains or when the number of tools required exceeds what a single agent can manage reliably.

Building on the single-agent supply chain example from the previous subsection, let's evolve it into a multiagent system. Here, we decompose the 16 tools into three specialized agents: one for inventory and warehouse management, one for transportation and logistics, and one for supplier relations and compliance. A supervisor agent routes queries to the appropriate specialist, embodying manager coordination (detailed in "Manager Coordination"). This setup demonstrates specialization by narrowing each agent's toolset and prompt, reducing selection errors and improving reliability. The code begins with imports and a shared response tool, ensuring all specialists can communicate outcomes uniformly. This shared tool minimizes duplication while allowing decentralized execution:

```
import os
import json
import operator
from typing import Annotated, Sequence, TypedDict, Opti

from langchain_openai.chat_models import ChatOpenAI
from langchain.schema import AIMessage, BaseMessage, Hu
from langchain_core.messages.tool import ToolMessage
from langchain.callbacks.streaming_stdout import Stream

from langchain.tools import tool
from langgraph.graph import StateGraph, END

from traceloop.sdk import Traceloop
from src.common.observability.loki_logger import log_tc

os.environ["OTEL_EXPORTER_OTLP_ENDPOINT"] = "http://loc
os.environ["OTEL_EXPORTER_OTLP_INSECURE"] = "true"
```

```python
# Shared tool for all specialists
@tool
def send_logistics_response(operation_id = None, messag
    """Send logistics updates, recommendations, or statu
    stakeholders."""
    print(f"[TOOL] send_logistics_response → {message}")
    log_to_loki("tool.send_logistics_response",
                f"operation_id={operation_id}, message={
    return "logistics_response_sent"

# Inventory & Warehouse Specialist Tools
@tool
def manage_inventory(sku: str = None, **kwargs) -> str:
    """Manage inventory levels, stock replenishment, auc
    and optimization strategies."""
    print(f"[TOOL] manage_inventory(sku={sku}, kwargs={k
    log_to_loki("tool.manage_inventory", f"sku={sku}")
    return "inventory_management_initiated"

@tool
def optimize_warehouse(operation_type: str = None, **kw
    """Optimize warehouse operations, layout, capacity,
    print(f"[TOOL] optimize_warehouse(operation_type={op
         kwargs={kwargs})")
    log_to_loki("tool.optimize_warehouse", f"operation_t
    return "warehouse_optimization_initiated"

@tool
def forecast_demand(season: str = None, **kwargs) -> st
    """Analyze demand patterns, seasonal trends, and cre
    print(f"[TOOL] forecast_demand(season={season}, kwar
    log_to_loki("tool.forecast_demand", f"season={seasor
    return "demand_forecast_generated"

@tool
def manage_quality(supplier: str = None, **kwargs) -> s
    """Manage quality control, defect tracking, and supp
    print(f"[TOOL] manage_quality(supplier={supplier}, k
    log_to_loki("tool.manage_quality", f"supplier={suppl
    return "quality_management_initiated"

@tool
def scale_operations(scaling_type: str = None, **kwargs
    """Scale operations for peak seasons, capacity planr
    workforce management."""
    print(f"[TOOL] scale_operations(scaling_type={scalir
        kwargs={kwargs})")
    log_to_loki("tool.scale_operations", f"scaling_type=
    return "operations_scaled"

@tool
def optimize_costs(cost_type: str = None, **kwargs) ->
    """Analyze and optimize transportation, storage, anc
    print(f"[TOOL] optimize_costs(cost_type={cost_type},
    log_to_loki("tool.optimize_costs", f"cost_type={cost
    return "cost_optimization_initiated"

INVENTORY_TOOLS = [manage_inventory, optimize_warehouse
manage_quality, scale_operations, optimize_costs, send_

# Transportation & Logistics Specialist Tools
@tool
def track_shipments(origin: str = None, **kwargs) -> st
    """Track shipment status, delays, and coordinate del
    print(f"[TOOL] track_shipments(origin={origin}, kwar
    log_to_loki("tool.track_shipments", f"origin={origir
    return "shipment_tracking_updated"

@tool
def arrange_shipping(shipping_type: str = None, **kwarg
    """Arrange shipping methods, expedited delivery,
    and multi-modal transportation."""
    print(f"[TOOL] arrange_shipping(shipping_type={shipp
        kwargs={kwargs})")
    log_to_loki("tool.arrange_shipping", f"shipping_type
    return "shipping_arranged"

@tool
def coordinate_operations(operation_type: str = None, *
    """Coordinate complex operations like cross-docking,
    and transfers."""
    print(f"[TOOL] coordinate_operations(operation_type=
        kwargs={kwargs})")
    log_to_loki("tool.coordinate_operations", f"operatic
    return "operations_coordinated"

@tool
def manage_special_handling(product_type: str = None, *
    """Handle special requirements for hazmat, cold chai
    and sensitive products."""
    print(f"[TOOL] manage_special_handling(product_type=
        kwargs={kwargs})")
```

```python
    log_to_loki("tool.manage_special_handling", f"produc
    return "special_handling_managed"

@tool
def process_returns(returned_quantity: str = None, **kw
    """Process returns, reverse logistics, and product c
    print(f"[TOOL] process_returns(returned_quantity={re
         kwargs={kwargs})")
    log_to_loki("tool.process_returns", f"returned_quant
    return "returns_processed"

@tool
def optimize_delivery(delivery_type: str = None, **kwar
    """Optimize delivery routes, last-mile logistics,
    and sustainability initiatives."""
    print(f"[TOOL] optimize_delivery(delivery_type={deli
         kwargs={kwargs})")
    log_to_loki("tool.optimize_delivery", f"delivery_typ
    return "delivery_optimization_complete"

@tool
def manage_disruption(disruption_type: str = None, **kw
    """Manage supply chain disruptions, contingency plar
    and risk mitigation."""
    print(f"[TOOL] manage_disruption(disruption_type={di
         kwargs={kwargs})")
    log_to_loki("tool.manage_disruption", f"disruption_t
    return "disruption_managed"

TRANSPORTATION_TOOLS = [track_shipments, arrange_shippi
    manage_special_handling, process_returns, optimize_
    manage_disruption, send_logistics_response]

# Supplier & Compliance Specialist Tools
@tool
def evaluate_suppliers(supplier_name: str = None, **kwa
    """Evaluate supplier performance, conduct audits,
    and manage supplier relationships."""
    print(f"[TOOL] evaluate_suppliers(supplier_name={sup
         kwargs={kwargs})")
    log_to_loki("tool.evaluate_suppliers", f"supplier_na
    return "supplier_evaluation_complete"

@tool
def handle_compliance(compliance_type: str = None, **kw
    """Manage regulatory compliance, customs, documentat
    and certifications."""
    print(f"[TOOL] handle_compliance(compliance_type={cc
         kwargs={kwargs})")
    log_to_loki("tool.handle_compliance", f"compliance_t
    return "compliance_handled"

SUPPLIER_TOOLS = [evaluate_suppliers, handle_compliance

Traceloop.init(disable_batch=True, app_name="supply_cha
llm = ChatOpenAI(model="gpt-4o", temperature=0.0,
    callbacks=[StreamingStdOutCallbackHandler()], verbc

# Bind tools to specialized LLMs
inventory_llm = llm.bind_tools(INVENTORY_TOOLS)
transportation_llm = llm.bind_tools(TRANSPORTATION_TOOL
supplier_llm = llm.bind_tools(SUPPLIER_TOOLS)
```

With tools grouped, we bind them to separate language model instances for each specialist. This allows for tailored prompts and reduces context size per agent, enhancing focus and efficiency. Multiagent architectures like this enable parallel processing (e.g., one agent optimizing delivery while another evaluates suppliers) cutting response times in high-volume logistics. The shared state ensures seamless handoffs.

The supervisor node acts as a central coordinator, analyzing queries and routing to specialists—exemplifying streamlined decision making without full consensus overhead. Specialist nodes then process independently, invoking tools and responding. This structure mitigates conflicts through clear role boundaries and enables parallelism if edges are expanded to concurrent calls:

```
class AgentState(TypedDict):
    operation: Optional[dict]  # Supply chain operation
    messages: Annotated[Sequence[BaseMessage], operator.

# Supervisor (Manager) Node: Routes to the appropriate
def supervisor_node(state: AgentState):
    history = state["messages"]
    operation = state.get("operation", {})
    operation_json = json.dumps(operation, ensure_ascii=

    supervisor_prompt = (
        "You are a supervisor coordinating a team of sup
        "Team members:\n"
```

```
            "- inventory: Handles inventory levels, forecast
            "quality, warehouse optimization, scaling, and c
            "- transportation: Handles shipping tracking,\n'
            "arrangements, operations coordination,\n"
             specialhandling, returns, delivery optimizatior
            "- supplier: Handles supplier evaluation and com
            "\n"
            "Based on the user query, select ONE team member
            "Output ONLY the selected member's name\n"
            "(inventory, transportation, or supplier), nothi
            f"OPERATION: {operation_json}"
        )

        full = [SystemMessage(content=supervisor_prompt)] +
        response = llm.invoke(full)
        return {"messages": [response]}

    # Specialist Node Template
    def specialist_node(state: AgentState, specialist_llm,
        history = state["messages"]
        operation = state.get("operation", {})
        if not operation:
            operation = {"operation_id": "UNKNOWN", "type":
                "priority": "medium", "status": "active"}
        operation_json = json.dumps(operation, ensure_ascii=
        full_prompt = system_prompt + f"\n\nOPERATION: {oper

        full = [SystemMessage(content=full_prompt)] + histor

        first: ToolMessage | BaseMessage = specialist_llm.ir
        messages = [first]

        if getattr(first, "tool_calls", None):
            for tc in first.tool_calls:
                print(first)
                print(tc['name'])
                # Find the tool (assuming tools are unique t
                all_tools = INVENTORY_TOOLS + TRANSPORTATION
                fn = next(t for t in all_tools if t.name ==
                out = fn.invoke(tc["args"])
                messages.append(ToolMessage(content=str(out)

            second = specialist_llm.invoke(full + messages)
            messages.append(second)

        return {"messages": messages}
    # Inventory Specialist Node
    def inventory_node(state: AgentState):
        inventory_prompt = (
            "You are an inventory and warehouse management s
            "When managing:\n"
            "  1) Analyze the inventory/warehouse challenge\
            "  2) Call the appropriate tool\n"
            "  3) Follow up with send_logistics_response\n"
            "Consider cost, efficiency, and scalability."
        )
        return specialist_node(state, inventory_llm, invento

    # Transportation Specialist Node
    def transportation_node(state: AgentState):
        transportation_prompt = (
            "You are a transportation and logistics speciali
            "When managing:\n"
            "  1) Analyze the shipping/delivery challenge\n'
            "  2) Call the appropriate tool\n"
            "  3) Follow up with send_logistics_response\n"
            "Consider efficiency, sustainability, and risk m
        )
        return specialist_node(state, transportation_llm, tr

    # Supplier Specialist Node
    def supplier_node(state: AgentState):
        supplier_prompt = (
            "You are a supplier relations and compliance spe
            "When managing:\n"
            "  1) Analyze the supplier/compliance issue\n"
            "  2) Call the appropriate tool\n"
            "  3) Follow up with send_logistics_response\n"
            "Consider performance, regulations, and relatior
        )
        return specialist_node(state, supplier_llm, supplier
```

Finally, the graph assembles the system with conditional edges for routing, enabling adaptability as the supervisor dynamically selects based on query content. In execution, this enables efficient handling of diverse tasks without a single point of overload. While coordination adds some latency, the benefits in scalability and reliability far outweigh it for complex environments:

```python
# Routing function for conditional edges
def route_to_specialist(state: AgentState):
   last_message = state["messages"][-1]
   agent_name = last_message.content.strip().lower()
   if agent_name == "inventory":
       return "inventory"
   elif agent_name == "transportation":
       return "transportation"
   elif agent_name == "supplier":
       return "supplier"
   else:
       # Fallback if no match
       return END

def construct_graph():
   g = StateGraph(AgentState)
   g.add_node("supervisor", supervisor_node)
   g.add_node("inventory", inventory_node)
   g.add_node("transportation", transportation_node)
   g.add_node("supplier", supplier_node)

   g.set_entry_point("supervisor")
   g.add_conditional_edges("supervisor", route_to_speci
    {"inventory": "inventory", "transportation":
    "transportation", "supplier": "supplier"})

   g.add_edge("inventory", END)
   g.add_edge("transportation", END)
   g.add_edge("supplier", END)

   return g.compile()

graph = construct_graph()

if __name__ == "__main__":
   example = {"operation_id": "OP-12345", "type": "inve
              "priority": "high", "location": "Warehous
   convo = [HumanMessage(content='''We're running criti
     low on SKU-12345. Current stock is 50 units
     but we have 200 units on backorder. What's our rec
     strategy?''')]
   result = graph.invoke({"operation": example, "messag
   for m in result["messages"]:
        print(f"{m.type}: {m.content}")
```

This multiagent framework exemplifies the power of adaptability in action. For instance, if a query involves a sudden supply disruption during peak season, the supervisor could route it to the transportation specialist for immediate containment, while the inventory specialist concurrently scales warehouse operations. This type of dynamic rerouting has become commonplace, enabling systems to pivot in response to real-time data like weather events or market shifts, thereby minimizing downtime and optimizing resource allocation. In our code, the conditional edges facilitate this flexibility, as the supervisor's output determines the flow, enabling the system to handle evolving conditions without rigid predefined paths. This not only boosts throughput through potential parallelism—such as forking to multiple specialists if extended—but also enhances resilience, as failures in one agent (e.g., due to API downtime) don't halt the entire process.

Adaptability is another core advantage, as multiagent systems can respond dynamically to changing conditions. By coordinating their actions, agents can reallocate roles and responsibilities as needed, adapting to new information or environmental changes in real time. This adaptability enables the system to remain efficient and effective in complex and unpredictable scenarios, where static, single-agent approaches may struggle to keep up.

However, multiagent systems are not without challenges. With multiple agents interacting, the complexity of coordination increases, requiring sophisticated communication and synchronization mechanisms to ensure agents work harmoniously. Communication overhead is another challenge, as agents must frequently exchange information to stay aligned and avoid duplicating efforts. This need for communication can slow down the system and introduce additional resource demands, especially in large-scale applications. Additionally, conflicts between agents may arise if they pursue overlapping goals or fail to prioritize effectively, necessitating protocols for conflict resolution and resource allocation.

In sum, while multiagent systems offer powerful advantages in handling complex, multifaceted tasks, they also require careful planning to manage the additional complexity and coordination requirements they introduce. By assigning agents distinct roles, enabling parallel processing, and incorporating adaptability and redundancy, multiagent systems can achieve high levels of performance, reliability, and flexibility, particularly in scenarios where a single-agent approach would fall short.

Swarms

Swarms represent a distinctive approach to agentic system design, inspired by decentralized systems in nature—such as flocks of birds, schools of fish, or colonies of ants. In swarm-based systems, large numbers of simple agents operate with minimal individual intelligence but collectively give rise to intelligent, emergent behavior through local interactions and simple rules.

Unlike traditional multiagent systems, which often rely on explicit role assignment and centralized coordination, swarm systems emphasize decentralization and self-organization. Each agent follows its own set of local policies or behaviors, typically without a global view of the system. Yet, through repeated, local interactions—such as broadcasting small updates, reacting to neighbors, or adapting based on shared signals—the swarm can adapt to changing conditions, solve complex problems, and exhibit robust group-level behavior. Key advantages of swarm-based systems include:

Scalability

> Because swarm agents are loosely coupled and locally driven, the system can scale to hundreds or thousands of agents with minimal coordination overhead.

Robustness

> There is no single point of failure. If individual agents fail, others can continue operating without significant degradation in performance.

Flexibility

> Swarms can adapt in real time to changing goals or environments, making them well suited to dynamic or unpredictable scenarios.

Distributed problem-solving

> Tasks such as exploration, monitoring, consensus formation, or distributed search can be tackled effectively through swarm dynamics.

Swarms are particularly effective in environments where centralized control is impractical or undesirable. For example, they are useful in large-scale data discovery, researching across multiple sources, or distributed decision making. In these scenarios, agents can operate semi-independently, contribute small insights or actions, and let global behavior emerge from the accumulation of local actions.

However, designing swarm systems comes with unique challenges, especially around predictability, observability, and efficiency. Despite these limitations, swarm-based systems offer a powerful and elegant solution for problems that benefit from decentralization, parallelism, and resilience. While not suitable for every problem domain, swarms shine in distributed environments and are increasingly relevant in fields like edge computing, sensor networks, and real-time collaborative systems—especially where flexibility and robustness matter more than precision or central control.

Principles for Adding Agents

When expanding a system by adding more agents, a strategic approach is essential to ensure the system remains efficient, manageable, and effective. The following principles serve as guidelines for optimizing agent-based design and functionality:

Task decomposition

> Task decomposition is a foundational principle, emphasizing the importance of breaking down complex tasks into smaller, manageable subtasks. By decomposing tasks, each agent can focus on a specific aspect of the workload, simplifying its responsibilities and improving efficiency. Clear task boundaries reduce overlap and redundancy, ensuring that each agent's contribution is valuable and that no effort is wasted. This decomposition not only enhances individual agent performance but also makes the system easier to coordinate and scale.

Specialization

> Specialization enables agents to be assigned roles that match their strengths, thereby maximizing the system's collective capabilities. When each agent is tasked with activities that align with its specific functions, the system operates with greater precision and effectiveness. Specialized agents are more adept at handling particular types of work, which translates to improved performance and faster task execution

overall. By designing agents with distinct responsibilities, the system can leverage diverse expertise to address complex or multidisciplinary tasks.

Parsimony

Parsimony is a guiding principle that encourages adding only the minimal number of agents necessary to achieve the desired functionality and performance. This principle emphasizes simplicity and efficiency, reminding developers that each agent added to the system introduces additional communication overhead, coordination complexity, and resource demands. By adhering to parsimony, developers avoid unnecessary agent proliferation, which can lead to increased maintenance burdens and potential performance bottlenecks. Parsimony requires careful assessment of each agent’s role and a disciplined approach to agent allocation, ensuring that each addition provides clear value to the system. Before adding an agent, developers should consider whether its responsibilities could be fulfilled by existing agents or by enhancing current capabilities. This focus on simplicity results in a streamlined, more manageable system that performs effectively without excessive redundancy. Ultimately, parsimony promotes an efficient, lean multiagent system that maximizes functionality while minimizing the risks and costs associated with complexity.

Coordination

Coordination is critical for the harmonious operation of multiagent systems. To maintain alignment among agents, robust communication protocols must be established, facilitating efficient information sharing and reducing the risk of conflicts. Coordination mechanisms should also include protocols for conflict resolution, particularly when agents have overlapping tasks or resource requirements. When agents can exchange information seamlessly and resolve issues autonomously, the system is more resilient and adaptable, capable of responding efficiently to dynamic scenarios.

Robustness

Robustness is essential for enhancing fault tolerance and resilience. Redundancy involves adding agents that can take over if others fail, providing backup support that ensures uninterrupted operation. In high-stakes environments, redundancy is invaluable for maintaining system stability and reliability. Robustness also encompasses designing agents and workflows that can withstand unexpected disruptions, such as network failures or agent downtime. By embedding redundancy and robustness into the system, developers can ensure that it remains functional even in adverse conditions.

Efficiency

Efficiency helps in assessing the trade-offs between adding agents and the potential complexity or resource demands that come with them. Each additional agent increases computational requirements and coordination overhead, so it is crucial to weigh the advantages of expanded functionality against these costs. By carefully evaluating the costs and benefits of each agent addition, developers can make informed decisions that balance system performance, resource efficiency, and scalability.

By following these principles, developers can determine the optimal number and configuration of agents required to achieve the desired balance of performance, efficiency, and complexity. This thoughtful approach enables the creation of multiagent systems that are both capable and sustainable, maximizing the benefits of additional agents while minimizing potential downsides.

Multiagent Coordination

Effective coordination among agents is critical for the success of multiagent systems. Various coordination strategies can be employed, each with its advantages and challenges. This section explores several of the leading coordination strategies, but we may see new approaches emerge.

Democratic Coordination

In democratic coordination, each agent within the system is given equal decision-making power, with the goal of reaching consensus on actions and solutions. This approach is characterized by decentralized control, where no single agent is designated as the leader. Instead, agents collaborate and share information equally, contributing their unique perspectives to collectively arrive at a decision. The key strength of democratic coordination is its robustness; because no agent holds a dominant role, the system has no single point of failure. This means that even if one or more agents experience failures, the overall system can continue functioning effectively. Another advantage is flexibility: when agents collaborate openly, they can quickly adapt to changes in their environment by updating their collective input. This adaptability is essential in dynamic settings where responsiveness to new information is crucial.

Moreover, democratic coordination promotes equity among agents, ensuring that all participants have an equal voice, which can lead to fairer outcomes.

However, democratic coordination comes with its own set of challenges. The process of reaching a consensus often requires extensive communication between agents, leading to significant communication overhead. As each agent must contribute and negotiate their perspective, the decision-making process can also be slow, potentially causing delays in environments where quick responses are necessary. Furthermore, implementing a democratic coordination protocol is often complex, as it requires well-defined communication and conflict-resolution mechanisms to facilitate consensus building. Despite these challenges, democratic coordination is particularly well suited for applications that prioritize fairness and robustness, such as distributed sensor networks or collaborative robotics, where each agent's contribution is valuable and consensus is essential for system success.

Manager Coordination

Manager coordination adopts a more centralized approach, where one or more agents are designated as managers that are responsible for overseeing and directing the actions of subordinate agents. In this model, managers take on a supervisory role, making decisions, distributing tasks, and resolving conflicts among agents under their guidance. One of the primary advantages of manager coordination is its streamlined decision making. Because managers have the authority to make decisions on behalf of the group, the system can operate more efficiently, bypassing the lengthy negotiation process required in democratic systems. This centralization also enables managers to clearly assign tasks and responsibilities, ensuring that agents focus on specific objectives without duplicating efforts or causing conflicts. Additionally, manager coordination simplifies communication pathways, as subordinate agents primarily communicate with their designated manager rather than with every other agent, reducing coordination complexity.

However, the reliance on managers introduces certain vulnerabilities. A single point of failure exists because if a manager agent fails or is compromised, the entire system may experience disruptions. Additionally, scalability becomes a concern as the system grows; managers can become bottlenecks if they cannot handle the increased volume of tasks or interactions required in larger networks. Finally, the centralized nature of decision making in manager coordination can reduce adaptability, as managers may not always be able to make the most informed decisions based on real-time changes within each subordinate's environment. This type of coordination is particularly effective in structured, hierarchical settings like manufacturing systems or customer support centers, where centralized control allows for optimized workflows and quicker conflict resolution.

Hierarchical Coordination

Hierarchical coordination takes a multitiered approach to organization, combining elements of both centralized and decentralized control through a structured hierarchy. In this system, agents are organized into multiple levels, with higher-level agents overseeing and directing those below them while affording subordinate agents a degree of autonomy. This approach provides significant scalability benefits, as the hierarchical structure enables coordination responsibilities to be distributed across multiple levels. By doing so, the system can manage a large number of agents more efficiently than a fully centralized model. The layered design also introduces redundancy, as tasks can be managed at different levels, improving fault tolerance. Clear lines of authority within the hierarchy streamline operations, with higher-level agents handling strategic decisions and lower-level agents focusing on tactical execution.

Despite these advantages, hierarchical coordination presents its own challenges. The complexity of designing a hierarchical system can be substantial, as each level must be carefully structured to ensure smooth coordination between layers. Communication delays can arise due to the need for information to propagate through multiple levels before reaching all agents, which can slow down responsiveness to urgent changes. Additionally, decision making at higher levels may introduce latency, as lower-level agents may need to wait for instructions before acting. Despite these challenges, hierarchical coordination is well suited for large, complex systems such as supply chain management or military operations, where different levels of coordination can handle both high-level planning and on-the-ground execution.

Actor-Critic Approaches

The actor-critic pattern in agentic systems is a lightweight form of evaluation-driven iteration. In this setup, the actor is responsible for generating candidate outputs—such as answers, plans, or actions—while the critic serves as a quality gate, accepting or rejecting outputs based on a predefined rubric.

The process is simple: the actor keeps producing candidates until the critic determines the output meets a desired quality threshold. This can be seen as a form of *test-time compute*, where additional inference cycles are used to improve reliability and performance. The trade-off is increased computational cost, but often with significantly better outcomes. This approach is especially effective in the following circumstances:

- There's a clear evaluation rubric or checklist (e.g., correctness, completeness, tone).
- The cost of generating additional outputs is acceptable relative to the benefit of higher quality.
- The task is fuzzy or generative in nature, where a single attempt often underperforms a reranked or filtered approach.

In the supply chain example, an "actor" agent generates reorder plans and a "critic" evaluates for feasibility (e.g., cost, risk), which is repeated until approval. This subsequent code adds an actor-critic loop after the supervisor:

```
# Actor Node: Generates candidate plans
def actor_node(state: AgentState):
   history = state["messages"]
   actor_prompt = '''Generate 3 candidate supply chain
     as JSON list: [{'plan': 'description', 'tools': [.
   response = llm.invoke([SystemMessage(content=actor_p
   state["candidates"] = json.loads(response.content)
   return state

# Critic Node: Evaluates and selects/iterates
```

```
def critic_node(state: AgentState):
    candidates = state["candidates"]
    history = state["messages"]
    critic_prompt = f'''Score candidates {candidates} on
     1-10 for feasibility, cost, risk. Select the best i
     8, else request regeneration.'''
    response = llm.invoke([SystemMessage(content=critic_
    eval = json.loads(response.content)
    if eval['best_score'] > 8:
        winning_plan = eval['selected']
        # Execute winning plan's tools (similar to speci
        messages = []
        for tool_info in winning_plan['tools']:
            tc = {'name': tool_info['tool'], 'args': toc
                  'id': 'dummy'}
            fn = next(t for t in all_tools if t.name ==
            out = fn.invoke(tc["args"])
            messages.append(ToolMessage(content=str(out)
        # Send response
        send_fn.invoke({"message": winning_plan['plan']]
        return {"messages": history + messages}
    else:
        # Iterate: Add feedback to history for actor
        return {"messages": history +
                [AIMessage(content="Regenerate with impr
                 eval['feedback'])]}

def construct_actor_critic_graph():
    g = StateGraph(AgentState)
    g.add_node("actor", actor_node)
    g.add_node("critic", critic_node)

    g.set_entry_point("actor")
    g.add_edge("actor", "critic")
    # Loop back if not approved (conditional)
    g.add_conditional_edges("critic", lambda s: "actor"
     if "regenerate" in s["messages"][-1].content.lower(
     END)

    return g.compile()
```

Actor-critic setups are particularly useful when evaluation is easier than generation. If you can reliably say "This is a good output," but can't easily produce it on the first try, then a simple actor-critic loop can be a powerful tool—no learning required. As an easy strategy to implement, it is often worth trying when a performance boost is worth the additional computational cost.

Automated Design of Agent Systems

Automated Design of Agentic Systems (ADAS) represents a transformative approach to agent development, shifting away from handcrafted architectures and toward systems that can design, evaluate, and iteratively improve themselves. As articulated by Shengran Hu, Cong Lu, and Jeff Clune in their 2024 original paper,[1] the central idea of ADAS is that, rather than manually constructing each component of an agent, we can enable a higher-level Meta Agent Search (MAS) algorithm to automatically create, assess, and refine agentic systems. This approach opens up a new research frontier—one that could enable agents to adapt to complex, shifting environments and continually improve their own capabilities without direct human intervention. As Figure 8-1 shows, ADAS builds on the idea that, historically, hand-designed solutions in machine learning (ML) have often been replaced by learned or automated alternatives, suggesting that agentic systems, too, may benefit from this transition.

In ADAS, foundation models serve as flexible, general-purpose modules within an agent's architecture. These models, which already power strategies such as chain-of-thought reasoning, self-reflection, and Toolformer-based agents, form a base upon which more specialized or task-specific capabilities can be layered. However, ADAS seeks to advance beyond these traditional approaches by enabling agents to invent entirely new structures and modules autonomously. The versatility of foundation models provides an ideal starting point, but ADAS leverages automated processes to push beyond predefined capabilities, enabling agents to evolve novel prompts, control flows, and tool use. These building blocks are not static; rather, they are generated dynamically by the meta-agent, which can continuously experiment with new designs in response to changing requirements or opportunities for improvement.

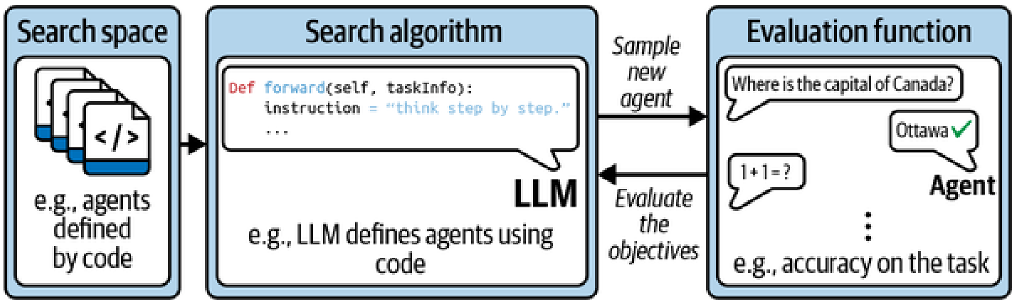
Search space
e.g., agents defined by code
Search algorithm
Def forward(self, taskInfo):
instruction = "think step by step."
...
LLM
e.g., LLM defines agents using code
Sample new agent
Evaluate the objectives
Evaluation function
Where is the capital of Canada?
Ottawa ✓
1+1=?
Agent
e.g., accuracy on the task

Figure 8-1. Core components of the ADAS framework. The search space outlines the scope of representable agentic architectures. The search algorithm dictates the exploration strategy within this space. The evaluation function quantifies candidate agents' effectiveness against objectives like performance, robustness, and efficiency. From the original paper.

The backbone of ADAS is the concept of defining agents through code. By utilizing programming languages that are Turing-complete, this framework theoretically allows agents to invent any conceivable structure or behavior. This includes complex workflows, creative tool integrations, and innovative decision-making processes that a human designer may not have foreseen. The power of ADAS lies in this code-based approach, which treats agents not as static entities but as flexible constructs that can be redefined, modified, and optimized over time. The potential of this approach is vast: in principle, a meta-agent could develop an endless variety of agents, continually refining and combining elements in pursuit of higher performance across diverse tasks.

Central to ADAS is the MAS algorithm, a specific method that demonstrates how a meta-agent can autonomously generate and refine agent systems. In MAS, the meta-agent acts as a designer, writing code to define new agents and testing these agents against an array of tasks. Each successful design is archived, forming a continuously growing knowledge base that informs the creation of future agents. MAS operates through an iterative cycle: the meta-agent, conditioned on an archive of prior agents, generates a high-level design description, implements it in code (defining a "forward" function for the agent), and refines via two self-reflection steps for novelty and correctness. The new agent is evaluated on validation data; errors trigger up to five debugging refinements. Successful agents are archived with performance metrics (e.g., accuracy or F1 score), informing future iterations. This mirrors evolutionary processes, balancing exploration of novel designs with exploitation of high performers. The meta-agent is thus both a creator and a curator, balancing exploration of new designs with exploitation of successful patterns. This process mirrors the evolution of biological systems, where successful traits are preserved and iteratively modified to adapt to new challenges.

To illustrate how MAS operationalizes these ideas, consider a generic Python implementation inspired by the open source ADAS. This framework uses a foundation model (e.g., GPT-5) as the meta-agent to generate and refine agent code. Key components include a foundation model agent base for prompting, a search loop for iterative evolution, and an evaluation function for fitness scoring. These elements enable the meta-agent to dynamically invent agents for tasks like grid puzzles (ARC [Abstraction and Reasoning Corpus]) or multiple-choice reasoning (MMLU), archiving high performers for future use:

```
class LLMAgentBase:
    def __init__(self, output_fields: list, agent_name:
     role='helpful assistant', model='gpt-4o-2024-05-13
     temperature=0.5):
        self.output_fields = output_fields
        self.agent_name = agent_name
        self.role = role
        self.model = model
        self.temperature = temperature
        self.id = random_id()  # Unique ID for agent ir

    def generate_prompt(self, input_infos, instruction,
        # Builds system prompt with role and JSON forma
        system_prompt = f"You are a {self.role}.\n\n" +
                        FORMAT_INST(output_descriptic
        # Constructs user prompt from inputs and instru
        prompt = ''  # (Build input text from infos) +
        return system_prompt, prompt

    def query(self, input_infos: list, instruction, out
              iteration_idx=-1):
        system_prompt, prompt = self.generate_prompt(ir
                                                     ir
                                                     ou
        response_json = get_json_response_from_gpt(pron
            self.model, system_prompt, self.temperature
        # Handle errors, parse JSON
        output_infos = [Info(key, self.__repr__(), valu
            iteration_idx) for key, value in response_j
        return output_infos
```

The `LLMAgentBase` class forms the core of the meta-agent, wrapping interactions with a foundation model to generate structured responses (e.g., thoughts, code). It enforces JSON outputs for parseability and handles errors gracefully, allowing the meta-agent to query for new agent designs based on archived priors. This modular design ensures flexibility: the role (e.g., “helpful assistant”) and temperature (for creativity) can be tuned, while output descriptions guide task-specific behaviors, such as returning only a single-letter answer for MMLU.

At the heart of MAS is the search function, which iterates over generations to evolve agents. Starting from an initial archive (e.g., basic prompt-based agents), it conditions the meta-agent on past successes, generates new code, applies Reflexion for refinement, evaluates on validation data, and archives fitness-scored solutions. This loop balances exploration (novel designs) with exploitation (building on high performers), often running for 25–30 generations:

```
def search(args, task):
    archive = task.get_init_archive()  # Or load existi
    for n in range(args.n_generation):
        # Generate prompt from archive
        msg_list = [{"role": "system", "content": syste
            {"role": "user", "content": prompt}]
        next_solution = get_json_response_from_gpt_refl
            msg_list, args.model)
        # Initial generation
        # Reflexion: Two steps to refine
        next_solution = reflect_and_refine(msg_list,
            task.get_reflexion_prompt())
        # Pseudocode for reflections
        # Evaluate and debug
        acc_list = evaluate_forward_fn(args, next_solut
        next_solution['fitness'] = bootstrap_confidence
        archive.append(next_solution)

def evaluate_forward_fn(args, forward_str, task):
    # Dynamically load agent code as function
    exec(forward_str, globals(), namespace)
    func = namespace['forward']  # Assume single functi
    data = task.load_data(SEARCHING_MODE)  # Val or tes
    task_queue = task.prepare_task_queue(data)
    # Parallel evaluate
    with ThreadPoolExecutor() as executor:
        acc_list = list(executor.map(process_item, task
        # process_item: run func, score vs truth
```

```
            return acc_list
```

The evaluation function dynamically loads the generated agent's code (via exec) as a callable forward function, applies it to task data in parallel (using multithreading for efficiency), and computes accuracy via task-specific scoring. This modular setup enables easy adaptation to new problems by subclassing a BaseTask abstract class, which defines methods for data loading, formatting, and prediction parsing. For example, in MMLU, it maps letter choices (A–D) to indices for exact-match scoring, while in ARC, it evaluates grid transformations for pixel-perfect accuracy. Such implementations demonstrate the generality of ADAS, leading to the strong empirical results observed.

The results of MAS reveal an intriguing property of agents designed through ADAS: they tend to maintain high levels of performance even when applied to new domains and models. For instance, on the ARC challenge (grid-transformation puzzles), MAS-discovered agents outperformed hand-designed baselines like Chain-of-Thought (CoT), Self-Refine, and LLM-Debate. On reasoning benchmarks, MAS achieved F1 scores of 79.4 ± 0.8 on DROP (reading comprehension, +13.6 over Role Assignment baseline), 53.4% ± 3.5 accuracy on MGSM (math, +14.4% over LLM-Debate), 69.6% ± 3.2 on MMLU (multitask, +2% over OPRO prompt optimization), and 34.6% ± 3.2 on GPQA (science, +1.7% over OPRO). Cross-domain transfer was robust (e.g., ARC agents applied to MMLU), and performance held when switching models (e.g., from GPT-3.5 to GPT-4).

This robustness across domains suggests that agents created through MAS are not merely optimized for one-off tasks; rather, they embody more general principles and adaptive structures that enable them to excel even when the specifics of the environment change. This cross-domain transferability reflects a fundamental advantage of automated design: by generating agents that are inherently flexible, MAS produces solutions that can generalize more effectively than those designed for narrow, specialized contexts.

ADAS holds significant promise, yet its development requires careful consideration of both ethical and technical dimensions. The potential to automate the design of ever-more-powerful agents introduces questions about safety, reliability, and alignment with human values. While MAS offers a structured and exploratory approach, it is crucial to ensure that the evolving agents adhere to ethical standards and do not develop unforeseen behaviors that could be misaligned with human intentions. Ensuring that these systems are beneficial necessitates a balance between autonomy and constraint, giving agents the freedom to innovate while guiding them to operate within safe and predictable bounds.

The trajectory of ADAS suggests a future where agentic systems can autonomously adapt, improve, and tackle an expanding range of tasks with minimal human intervention. As ADAS advances, the ability of agents to develop more sophisticated designs will likely become a cornerstone of AI research, providing tools that can address increasingly complex, evolving challenges. In this way, ADAS offers a glimpse into a future of intelligent systems capable of self-improvement and innovation, embodying a shift from static, predesigned agents to adaptive, autonomous systems that grow alongside our expanding needs.

Communication Techniques

As agentic systems grow from single-agent prototypes into multiagent, distributed systems, the choice of communication architecture becomes increasingly critical. What starts as simple in-memory message passing or function calls quickly becomes untenable as systems grow in scope, number of agents, geographic distribution, or deployment complexity. This section explores the core techniques and technologies available for managing communication, coordination, and task flow across agents—especially as systems transition from single-device experiments to production-grade distributed deployments. The reader will notice there are many valid approaches, all with different trade-offs in development effort, latency, scalability, reliability, and cost.

Local Versus Distributed Communication

At a small scale—such as a single-device or single-process setup—agents often communicate through direct function calls, shared memory, or in-memory message queues. While simple and efficient, these methods don’t scale well. As soon as agents are distributed across services, containers, or nodes, communication must be made explicit, asynchronous, and fault-tolerant.

In local deployments, frameworks like AutoGen often use in-memory routers to orchestrate agent message passing and tool invocation. These setups can work well for research and prototyping, especially with single-threaded or single-agent configurations. But for production use, communication and state management must evolve.

Agent-to-Agent Protocol

The Agent-to-Agent (A2A) Protocol, introduced by Google, is an ambitious and promising step toward enabling autonomous agents to work together toward more complex goals. It offers a standardized, cross-platform mechanism for agents to discover each other, negotiate collaboration, and exchange structured requests—without revealing internal logic or implementation details. By enabling heterogeneous agents to interoperate over HTTP-based transports, A2A creates a shared language that could, in time, make multiagent coordination as routine as API calls between microservices.

At the core of A2A is the *Agent Card*, a machine-readable JSON descriptor that each agent publishes to advertise its identity, capabilities, endpoints, and supported authentication methods. These cards enable agents to find peers, evaluate their functions, and negotiate secure communication channels. Capabilities are defined explicitly—such as `generateReport`, `summarizeLegalDocument`—along with schemas for inputs and outputs, enabling structured composition of agent workflows. Endpoint information and supported authentication methods (e.g., OAuth 2, API key) ensure that communication can be established securely and programmatically. Optional metadata like versioning and media support further enrich agent discovery and compatibility. To illustrate, here’s a simple Python dictionary representing an Agent Card for a summarization agent:

```python
agent_card = {
    "identity": "SummarizerAgent",
    "capabilities": ["summarizeText"],
    "schemas": {
        "summarizeText": {
            "input": {"text": "string"},
            "output": {"summary": "string"}
        }
    },
    "endpoint": "http://localhost:8000/api",
    "auth_methods": ["none"],  # In production: OAuth2,
    "version": "1.0"
}
```

This JSON can be served at a well-known endpoint like */.well-known/agent.json* for discovery. A2A uses JSON-RPC 2.0 over HTTPS as its reference implementation, but the protocol is designed to be transport-agnostic. This opens the door to integration over gRPC, WebSocket, or other streaming and multiplexed protocols as infrastructure demands evolve. JSON-RPC ensures consistent handling of requests, responses, and errors, creating a shared semantic model even across agents built in different languages or frameworks.

In practical use, agents locate one another via a registry—centralized or distributed—that stores Agent Cards. Once a peer is identified, an initiating agent performs a handshake, exchanging Agent Cards and negotiating session parameters like protocol version, timeout expectations, or payload limits. For example, a client agent might discover and negotiate compatibility like this (using Python's requests library):

```
import requests
import json

# Discover Agent Card (mocked as direct access; in proc
card_url = 'http://localhost:8000/.well-known/agent.jsc
response = requests.get(card_url)
if response.status_code != 200:
    raise ValueError("Failed to retrieve Agent Card")

agent_card = response.json()
print("Discovered Agent Card:", json.dumps(agent_card,

# Handshake: Check compatibility
if agent_card['version'] != '1.0':
    raise ValueError("Incompatible protocol version")
if "summarizeText" not in agent_card['capabilities']:
    raise ValueError("Required capability not supportec
print("Handshake successful: Agent is compatible.")
```

Once validated, the agents can begin coordinating work: Agent A may issue a `requestSummarize` call to Agent B, who then processes the request and returns a structured response or an error, as needed. Continuing the example, here's how the client issues a JSON-RPC request:

```
# Issue JSON-RPC request
rpc_url = agent_card['endpoint']
rpc_request = {
    "jsonrpc": "2.0",
    "method": "summarizeText",
    "params": {"text": '''This is a long example text t
        It discusses multiagent systems and communicati
    "id": 123  # Unique request ID
}

response = requests.post(rpc_url, json=rpc_request)
if response.status_code == 200:
    rpc_response = response.json()
    print("RPC Response:", json.dumps(rpc_response, inc
else:
    print("Error:", response.status_code, response.text
```

On the server side, handling this request might look like this (using Python's `http.server` for simplicity):

```
# Excerpt from server handler (in do_POST method)
import os
from openai import OpenAI

content_length = int(self.headers['Content-Length'])
post_data = self.rfile.read(content_length)
rpc_request = json.loads(post_data)

# Handle JSON-RPC request (core of A2A)
if rpc_request.get('jsonrpc') == '2.0'
    and rpc_request['method'] == 'summarizeText':
    text = rpc_request['params']['text']
    # Real LLM summarization using OpenAI API
    client = OpenAI(api_key=os.getenv("OPENAI_API_KEY")
    try:
        llm_response = client.chat.completions.create(
            model="gpt-4o",
            messages=[
                {"role": "system", "content": '''You ar
                    provides concise summaries.'''},
                {"role": "user", "content": f"""Summari
                {text}"""}
            ],
            max_tokens=150,
            temperature=0.7
        )
        summary = llm_response.choices[0].message.conte
    except Exception as e:
        summary = f"Error in summarization: {str(e)}"

    response = {
        "jsonrpc": "2.0",
        "result": {"summary": summary},
        "id": rpc_request['id']
    }
    # Send response
    self.send_response(200)
    self.send_header('Content-type', 'application/json'
    self.end_headers()
    self.wfile.write(json.dumps(response).encode())
else:
    # Error response
    error_response = {
        "jsonrpc": "2.0",
        "error": {"code": -32601, "message": "Method nc
        "id": rpc_request.get('id')
    }
    self.send_response(400)
    self.send_header('Content-type', 'application/json'
    self.end_headers()
    self.wfile.write(json.dumps(error_response).encode(
```

While A2A presents an exciting direction for multiagent systems—offering a modular, runtime-agnostic approach to delegation and coordination—it is still in its infancy. Significant open questions remain, particularly around security. Authentication is currently supported via pluggable mechanisms, but robust authorization, rate-limiting, trust establishment, and abuse resistance are far from solved. As with any early protocol, it should be approached with both enthusiasm and caution. Early adopters should expect vulnerabilities, implementation gaps, and evolving specifications.

Still, A2A points to a future where agents don't operate in isolation but as part of dynamic, loosely coupled ecosystems capable of tackling broader and more sophisticated problems. Much like HTTP enabled the composability of the web, A2A aspires to do the same for AI agents. It’s too early to say whether it will become *the* standard—but it’s a promising beginning in the quest to make agent cooperation seamless, scalable, and secure.

Message Brokers and Event Buses

As agent-based systems scale, point-to-point communication becomes brittle and inflexible. A common alternative is to adopt message brokers or event buses, which decouple senders from receivers and enable agents to interact asynchronously through a shared communication fabric. This pattern establishes scalable, fault-tolerant, and observable workflows, especially in loosely coupled multiagent architectures.

To see the utility of this approach, consider integrating a message broker into a supply chain multiagent system from earlier in this chapter. In the original synchronous setup, the supervisor directly routes to a specialist via graph edges, creating tight coupling. By using a broker, the supervisor can publish tasks to a shared topic (e.g., “supply-chain-tasks”), and specialists subscribe asynchronously—processing only relevant messages. This decouples agents, enabling independent scaling (e.g., replaying inventory instances), fault tolerance (e.g., replay missed messages), and easier addition of new agents without rewriting the graph. Key options include:

Apache Kafka

> This is a high-throughput, distributed event streaming platform ideal for agent systems where agents need to publish and consume structured events. Kafka supports strong durability, topic partitioning for parallelism, and consumer groups for coordination. It is especially effective for building log-based communication architectures where every interaction is preserved and replayable.

Redis Stream and RabbitMQ

> These are lightweight alternatives for lower-throughput or simpler use cases, with tighter latency and easier deployment. Redis Stream in particular offers fast, memory-based communication, though durability is more limited.

Neural Autonomic Transport System (NATS)

A lightweight, cloud-native messaging system designed for low-latency, high-throughput communication. NATS is ideal for real-time agent coordination in microservice or edge environments. It supports publish/subscribe, request/reply, and—with JetStream—durable message streams and replay. NATS emphasizes simplicity, speed, and scalability, making it well suited for distributed agentic systems that require fast, resilient communication with minimal overhead.

For the supply chain agent system, Redis Stream provides quick, low-latency decoupling ideal for prototyping. The supervisor adds tasks to a stream, and specialists read/consume them in separate processes. Assume Redis is running (e.g., via Docker: `docker run -p 6379:6379 redis`) and use redis-py (`pip install redis`). The supervisor determines the specialist and publishes the task:

```
import redis
import json
import uuid

# Helper to serialize messages
def serialize_messages(messages):
    return [m.dict() for m in messages]

def supervisor_publish(operation: dict, messages):
    # ... (existing supervisor prompt and LLM logic to
    r = redis.Redis(host='localhost', port=6379)
    task_id = str(uuid.uuid4())
    task_message = {
        'task_id': task_id,
        'agent': agent_name,
        'operation': operation,
        'messages': serialize_messages(messages)
    }
    r.xadd('supply-chain-tasks', {'data': json.dumps(ta
    return task_id
```

Specialists (e.g., inventory) consume in a loop, process with their node logic, and publish responses:

```
import redis
import json
```

```
# Helper to deserialize messages
def deserialize_messages(serialized):
    # Rehydrate based on type (HumanMessage, AIMessage,
    return [...]  # Implementation as in full code

def inventory_consumer():
    r = redis.Redis(host='localhost', port=6379)
    last_id = '0'
    # ... (inventory_prompt)
    while True:
        msgs = r.xread({'supply-chain-tasks': last_id},
        if msgs:
            stream, entries = msgs[0]
            for entry_id, entry_data in entries:
                task = json.loads(entry_data[b'data'])
                if task['agent'] == 'inventory':
                    state = {
                        'operation': task['operation'],
                        'messages': deserialize_message
                    }
                    result = specialist_node(state, inv
                                             inventory_
                    response = {
                        'task_id': task['task_id'],
                        'from': 'inventory',
                        'result': {'messages': serializ
                                   result['messages'])}
                    }
                    r.xadd('supply-chain-responses', {'
                           json.dumps(response)})
                last_id = entry_id
```

We then set up similar consumer loops to run for transportation and supplier specialists. To wait for a response:

```
import time

def wait_for_response(task_id, timeout=60):
    r = redis.Redis(host='localhost', port=6379)
    last_id = '0'
    start = time.time()
    while time.time() - start < timeout:
        msgs = r.xread({'supply-chain-responses': last_
        if msgs:
```

```
                    stream, entries = msgs[0]
                    for entry_id, entry_data in entries:
                        resp = json.loads(entry_data[b'data'])
                        if resp['task_id'] == task_id:
                            return resp
                        last_id = entry_id
        raise TimeoutError("No response")
```

In general, it's wise to run specialists in separate processes (e.g., via multiprocessing). This enables fast async coordination—e.g., the supplier agent can process compliance tasks without blocking others—while keeping setup simple for lower-scale systems.

Message buses support loose coupling between agents, allowing for flexible scaling, observability via logging pipelines, and replay of failed or missed messages. However, they also introduce challenges around eventual consistency and the need for more complex error handling.

Actor Frameworks: Ray, Orleans, and Akka

While message buses primarily decouple communication by routing events asynchronously between components—focusing on data flow without dictating execution—actor frameworks integrate both messaging and computation into a unified model. Here, actors (representing agents) not only exchange messages but also encapsulate their own state and behavior, ensuring sequential processing to eliminate race conditions and shared-state bugs common in traditional threaded systems. This contrasts sharply with the standard monolithic approach many developers initially take: deploying a single-container agent service that handles all logic centrally, often relying on synchronous foundation model calls and in-memory orchestration. While simple for prototypes, such setups become bottlenecks at scale—prone to single points of failure, inefficient resource use during idle periods, and challenges in parallelizing diverse agent roles without custom concurrency hacks.

Actor frameworks shine in scenarios requiring fine-grained distribution, resilience, and dynamic scaling, such as multiagent simulations with persistent per-agent memory (e.g., tracking conversation history or learned behaviors), high-concurrency environments like real-time bidding or IoT coordination, or systems integrating heterogeneous agents across clusters. They enable "location-transparent" invocation—where actors can migrate or replicate without changing code—and built-in supervision for automatic recovery from failures, reducing operational overhead compared with manually managing queues or containers.

The investment in infrastructure (e.g., setting up clusters, monitoring actor lifecycles) pays off when systems exceed a few agents or handle variable workloads: for instance, in production agent swarms where downtime costs are high, or when evolving from local prototypes to cloud native deployments. For smaller, low-traffic setups, the added complexity may not justify it—stick to buses or monolithic services—but as the agent count grows beyond 10–20 or latency demands tighten, actors provide unmatched elasticity and fault tolerance. Three leading frameworks in this space are Ray, Orleans, and Akka, each offering distinct advantages depending on the environment and language ecosystem:

Ray

> Ray is a Python-native distributed computing framework that supports an actor model for stateful, scalable computations. Actors in Ray are defined using the `@ray.remote` decorator, enabling asynchronous method invocations that process messages while preserving internal state across invocations. Ray manages distribution automatically, with resource-aware scheduling, fault tolerance via optional restarts and retries, and support for clustering to handle large-scale deployments. It pairs naturally with tools like AutoGen or LangGraph for agentic systems, offering a lightweight alternative in Python environments where ease of use and rapid prototyping are prioritized over JVM-specific (Java Virtual Machine) performance tuning.

Orleans

> Orleans offers a *virtual actor model*, where actors (or agents) are logically addressable and automatically instantiated, suspended, or recovered based on demand. Orleans handles state persistence, concurrency, and lifecycle management with minimal boilerplate. It abstracts away much of the complexity of distributed systems while enabling developers to scale agent-like components naturally across a

cluster. When paired with AutoGen, Orleans can power agent systems that treat each agent as a service, dynamically scaling with system needs while retaining internal state and identity.

Akka

Akka is a well-established actor framework in the JVM ecosystem, supporting both Java and Scala. Akka's classic actor model is highly performant and suitable for building fault-tolerant, distributed systems with fine-grained control over actor behavior. With Akka Cluster, actors can be distributed across multiple nodes, supporting advanced features like sharding, persistence, supervision, and adaptive load balancing. Akka is particularly well suited for high-throughput, low-latency applications requiring tight control over concurrency, and it has been used in production environments ranging from telecom systems to trading platforms.

This actor-style design aligns naturally with multiagent coordination, where each agent maintains its own identity, role, and internal state. Actor systems enable these agents to be invoked dynamically, react to messages or events, and manage complex workflows through message passing rather than shared state or global control.

Because this book emphasizes Python-based implementations for multiagent systems (e.g., using LangChain and related libraries), we'll illustrate the actor model with a Ray example integrated into the supply chain system. Similar principles apply to Orleans (primarily .NET-based, ideal for Windows ecosystems or enterprise integrations) and Akka (JVM-focused, suited for high-performance Java/Scala apps), but their code would require language-specific adaptations beyond our Python-centric scope.

In the context of the supply chain multiagent system, the specialist agents (e.g., inventory, transportation) are implemented as Ray actors with per-session isolation. Each session (identified by `operation_id`) gets its own actor instance per specialist type, ensuring clean state management—isolated history or caches per session—while guaranteeing sequential execution within each actor for tasks in that session. This avoids cross-session contamination and enables parallel processing across sessions in a cluster. A session manager actor tracks and creates these on demand. Here's the core Ray actor class for a specialist, which processes tasks sequentially and maintains isolated session state:

```
@ray.remote
class SpecialistActor:
    def __init__(self, name: str, specialist_llm, tools
                 system_prompt: str):
        self.name = name
        self.llm = specialist_llm
        self.tools = {t.name: t for t in tools}
        self.prompt = system_prompt
        self.internal_state = {}

    def process_task(self, operation: dict, messages: S
        if not operation:
            operation = {"operation_id": "UNKNOWN", "ty
                         "priority": "medium", "status'
        operation_json = json.dumps(operation, ensure_a
        full_prompt = self.prompt + f"\n\nOPERATION: {c

        full = [SystemMessage(content=full_prompt)] + m

        first = self.llm.invoke(full)
        result_messages = [first]

        if hasattr(first, "tool_calls"):
            for tc in first.tool_calls:
                print(first)
                print(tc['name'])
                fn = self.tools.get(tc['name'])
                if fn:
                    out = fn.invoke(tc["args"])
                    result_messages.append(ToolMessage(
                                           tool_call_ic

            second = self.llm.invoke(full + result_mess
            result_messages.append(second)

        # Update internal state (example: track process
        step_key = str(len(self.internal_state) + 1)
        # Or use a more specific key
        self.internal_state[step_key] = {"status": "prc
                                         "timestamp": t

        return {"messages": result_messages}

    def get_state(self):
        return self.internal_state  # Return entire ses
```

This actor encapsulates the foundation model and tool logic, processing messages (tasks) serially via `process_task` —Ray queues concurrent calls to the same actor and executes them one by one, preserving order and state integrity. The `internal_state` dict is session-isolated because each actor is created per session, enabling per-session persistence (e.g., step tracking) without shared memory risks. A session manager actor handles dynamic creation for isolation:

```
@ray.remote
class SessionManager:
    def __init__(self):
        self.sessions: Dict[str, Dict[str, ray.actor.Ac

    def get_or_create_actor(self, session_id: str, ager
        llm, tools: list, prompt: str):
        if session_id not in self.sessions:
            self.sessions[session_id] = {}
        if agent_name not in self.sessions[session_id]:
            actor = SpecialistActor.remote(agent_name,
            self.sessions[session_id][agent_name] = act
        return self.sessions[session_id][agent_name]

    def get_session_state(self, session_id: str, agent_
        if session_id in self.sessions and
              agent_name in self.sessions[session_id]:
            actor = self.sessions[session_id][agent_nam
            return actor.get_state.remote()  # Returns
        return None
```

The manager uses a dict to track actors by `session_id` and `agent_name`, creating them lazily. This enables scalability: Ray distributes actors across cluster nodes, and querying state (e.g., `ray.get(manager.get_session_state.remote(session_id, agent_name))`) retrieves session-specific data without global sharing.

For developers building agentic systems, actor frameworks like Orleans and Akka offer a proven, scalable foundation for representing each agent as an autonomous, self-contained unit—capable of handling asynchronous workflows, maintaining persistent memory, and integrating cleanly into distributed infrastructures.

Orchestration and Workflow Engines

Even with robust messaging and agent execution models, real-world systems need orchestration—the logic that sequences tasks, handles retries, tracks dependencies, and manages failure across agents. This is especially important for long-running or multistep interactions that span time and components. Workflow orchestration tools provide a higher-level abstraction, ensuring durability and recoverability in complex agentic systems.

Workflow orchestration tools are particularly useful when processes involve unreliable external dependencies (e.g., APIs, foundation models, or human approvals), potential failures, or extended durations—such as supply chain workflows that may take days due to asynchronous agent actions or real-world delays. By persisting state and automating recovery, these engines prevent data loss and redundant work, making them essential for production-grade reliability where simple in-memory coordination falls short. Use them when scaling from prototypes to resilient deployments, especially in scenarios with high stakes like financial transactions, compliance-heavy operations, or distributed AI agents; for quick, low-risk experiments, basic scripting may suffice.

Temporal provides durable, stateful workflows with long-running tasks, retries, and failure recovery. It's ideal for managing multiagent systems where each agent may perform asynchronous, multistep actions. Temporal workflows offer a clean abstraction for encapsulating business logic that spans multiple services or agents over long durations.

To illustrate Temporal's durable execution in the supply chain multiagent system, consider a workflow that sequences agent steps (e.g., inventory management, then transportation arrangement, followed by supplier compliance)—with automatic retries on failures and persistent state for recovery. Temporal ensures the workflow resumes from the last successful step even after crashes, making it suitable for production agent coordination. Assume Temporal is set up (e.g., via `pip install temporalio`), with activities defined for each specialist (wrapping their foundation model/tool logic). Here's a simplified workflow definition:

```
from datetime import timedelta
from temporalio import workflow
from temporalio.common import RetryPolicy

# Assume activities are defined elsewhere, e.g., invent
# transportation_activity, supplier_activity
# Each takes operation dict and messages, returns resul

@workflow.defn
class SupplyChainWorkflow:
    @workflow.run
    async def run(self, operation: dict, initial_messag
        # Step 1: Inventory management with retry
        inventory_result = await workflow.execute_activ
            "inventory_activity",
            {"operation": operation, "messages": initia
            start_to_close_timeout=timedelta(seconds=3
            retry_policy=RetryPolicy(maximum_attempts=
        )

        # Update state and proceed to transportation
        updated_messages = initial_messages + inventory
        transportation_result = await workflow.execute_
            "transportation_activity",
            {"operation": operation, "messages": update
            start_to_close_timeout=timedelta(seconds=3
            retry_policy=RetryPolicy(maximum_attempts=
        )

        # Final step: Supplier compliance
        final_messages = updated_messages + transportat
        supplier_result = await workflow.execute_activi
            "supplier_activity",
            {"operation": operation, "messages": final_
            start_to_close_timeout=timedelta(seconds=3
            retry_policy=RetryPolicy(maximum_attempts=
        )

        # Compile and return results
        return {
            "inventory": inventory_result,
            "transportation": transportation_result,
            "supplier": supplier_result
        }
```

This workflow durably sequences the agents. Each activity (agent step) runs with retries, and Temporal persists progress—e.g., if transportation fails, it retries without rerunning inventory. For long-running processes, add signals for user input or pauses, similar to the full example's confirmation handling.

Apache Airflow is widely used for data pipelines but can also coordinate agent flows via DAGs (directed acyclic graphs). While powerful, Airflow is best suited to batch or time-triggered workflows. Airflow remains a staple for scheduled, tool-agnostic orchestration in data engineering and business operations, such as ETL (extract, transform, load) jobs or ML model training. Opt for Airflow when dealing with periodic, dependency-heavy pipelines that benefit from its mature ecosystem and visualization tools, but not for real-time or highly dynamic agent interactions.

For developers preferring to prototype and run orchestration locally before scaling to distributed environments, tools like Dagger can be particularly useful, enabling workflows to be composed as code using containers, foundation models, and other resources with automatic caching and type safety. This ensures consistency across local development, CI/CD pipelines, and production, and even supports agentic integrations such as automation enabled by foundation models, making it a flexible option depending on your stack. Workflow engines offer a higher layer of abstraction—separating coordination logic from communication mechanics. They help ensure idempotency, recoverability, and durable state—features that become essential when agents fail, stall, or must respond to changing environments.

Managing State and Persistence

Communication alone is not enough—multiagent systems must also manage shared state, agent memory, and task metadata that often span multiple executions, workflows, or system restarts. This introduces significant complexity in terms of data durability, consistency, and access patterns, particularly as the system scales.

As you can see in Table 8-1, traditional solutions rely on stateful databases like PostgreSQL, Redis, or vector stores to persist task outcomes, interaction logs, and agent memories. These offer fine-grained control and can be tailored to the needs of each agent, but they also require developers to explicitly manage schema design, read/write consistency, caching, and recovery logic—adding engineering overhead and opportunities for subtle bugs.

For unstructured or large-scale outputs (e.g., plans, tool traces, JSON blobs), object storage options like Amazon S3 or Azure Blob Storage provide durable, low-cost storage with high availability. This is ideal for immutable artifacts, but it comes with trade-offs in access latency and the need for separate indexing or tracking systems to relate artifacts back to agent tasks or states.

Table 8-1. Durable storage option overview

Approach	Pros	Cons	Best for
Relational databases (e.g., PostgreSQL/Redis)	Flexible, queryable, cost-effective	Manual management, potential inconsistency	Custom, high-query systems
Vector stores (e.g., Pinecone)	Semantic search, scalable embeddings	Higher cost, specialized setup	Knowledge-intensive agents
Object storage (e.g., S3)	Cheap, durable for large data	Slow access, no native indexing	Archival outputs
Stateful orchestration frameworks	Automated recovery, low boilerplate	Framework lock-in	Resilient, long-running workflows

Frameworks like Temporal and Orleans offer a different approach: they abstract away much of the complexity of persistence by tightly integrating state management into the agent or workflow lifecycle. Temporal automatically checkpoints workflow progress, supports deterministic replay, and handles failures transparently. Orleans enables each actor (agent) to maintain a durable, event-driven state with minimal boilerplate. These abstractions reduce development effort and improve resilience, but they also impose framework-specific constraints—such as serialization formats, execution models, or language bindings—that may not suit every architecture.

The right choice depends on the nature of the memory and coordination required:

- Episodic memory (short-lived, task-specific state) may only need in-memory or transient storage with minimal durability.
- Semantic memory (long-term knowledge across interactions) typically requires durable storage with search or vector indexing capabilities.
- Workflow durability (resilience to mid-process failure) benefits most from integrated engines like Temporal or Orleans that automatically checkpoint progress and state.

Ultimately, persistence decisions reflect trade-offs between developer effort, performance, durability, and flexibility. Systems with tight service-level agreements, cross-agent dependencies, or real-time coordination requirements will often benefit from workflow-native persistence layers, while more modular or research-oriented systems may prefer explicit, database-driven state management that offers more control and visibility.

Conclusion

The transition from single-agent to multiagent systems offers significant advantages in addressing complex tasks, enhancing adaptability, and increasing efficiency. Yet, as we've explored in this chapter, the scalability that comes with adding more agents brings challenges that demand careful planning. Deciding on the optimal number of agents requires a nuanced understanding of task complexity, potential task decomposition, and the cost-benefit balance of multiagent collaboration.

Coordination is critical to success in multiagent systems, and a variety of coordination strategies—such as democratic, manager-based, hierarchical, actor-critic approaches, and automated design with ADAS—provide different trade-offs between robustness, efficiency, and complexity. Each coordination strategy offers unique advantages and limitations, suited to particular scenarios, and careful selection can significantly enhance a system's effectiveness and reliability.

Equally critical is the choice of communication infrastructure. As systems scale, so too does the need for reliable, low-latency, and durable message passing between agents. While in-memory queues may suffice in simple settings, production-grade systems often rely on message brokers (e.g., Kafka, NATS, RabbitMQ), actor frameworks (e.g., Orleans, Akka), and workflow engines (e.g., Temporal, Conductor) to manage not only communication but also state, retries, and execution durability. Designing for effective communication is not just an implementation detail—it is a first-class concern that shapes how agents perceive, respond to, and collaborate within their environment. To help developers navigate these options, Table 8-2 summarizes the key communication and execution approaches for multiagent systems, comparing their concepts, trade-offs, and ideal use cases in the context of our supply chain example.

Table 8-2. Agent coordination techniques

Approach	Key concepts	Benefits	Challenges	Use cases a
Single-container deployment	Monolithic agent/service in one container; synchronous calls, in-memory state/orchestration	Simple setup, low latency, easy prototyping	Single failure point, poor scalability, concurrency issues	Basic supp prototypes; limited age agent to ha inquiries)
A2A Protocol	Standardized discovery via Agent Cards, negotiation, JSON-RPC for structured requests; transport-agnostic (HTTP/gRPC)	Interoperable across heterogeneous agents, modular, secure channels	Early-stage (security gaps, evolving specs), discovery overhead	Agent colla ecosystems requesting another in s
Message brokers	Decoupled async messaging via publish/subscribe (Kafka for durability, Redis Stream for low-latency, NATS for real time)	Loose coupling, scalability, fault-tolerant replays	Eventual consistency, complex error handling, potential latency	Distributed chain (e.g., to a stream subscribing
Actor frameworks	Stateful actors processing messages sequentially (Ray for Python/distributed, Orleans for virtual actors, Akka for JVM/performance)	Integrated state/behavior, resilience (auto-recovery), location-transparent scaling	Infrastructure investment, framework lock-in, per actor sequential limits	Per-session supply cha creation fo in inventor

By understanding these factors and applying them thoughtfully, developers can create multiagent systems that are not only robust and capable but also prepared to meet the demands of increasingly complex, dynamic tasks in real-world applications. This strategic approach enables multiagent systems to evolve as powerful solutions that drive meaningful advancements across various domains.

[1] Shengran Hu et al., "Automated Design of Agentic Systems", paper presented at the International Conference on Learning Representations, Singapore, April 2025.