Cheat Sheet: Multi-Agent Systems and Agentic RAG with LangGraph

Why multi-agent systems?

Challenge faced by single LLM agents	Multi-agent system solution
Context overload	Splits tasks among agents to reduce burden
Role confusion	Agents specialize in distinct cognitive roles
Debugging difficulty	Modular agents ease error tracing
Quality dilution	Each agent excels at a focused subtask

Typical multi-agent communication patterns

Pattern	Description	Example
Sequential (Pipeline)	Agents work one after another, passing results	Research → Analysis → Writing → Review
Parallel with aggregation	Multiple agents run concurrently, results combined	SEO analysis, fact-checking, writing run in parallel
Interactive dialogue	Agents exchange messages to clarify or refine	Requirements agent queries data agent before finalizing

---

Real-world multi-agent use cases

Use case	Agents & workflow	Benefit
Automated market report	Research → Data analysis → Writing → Critique → Editing	Faster, accurate, well-rounded reports
Customer support	Intent detection → Knowledge retrieval → Response → Escalation	Dynamic, personalized, scalable responses
Legal contract review	Clause extraction → Compliance check → Risk analysis → Summary	Thorough, accurate, actionable legal reviews

---

Communication protocols

- **Model Context Protocol (MCP):** JSON-RPC-based interface for LLMs to interact with external tools/services, enabling modular, real-time collaboration.
- **IBM Agent Communication Protocol (ACP):** Standardizes message exchange among autonomous agents for secure, scalable enterprise workflows.

Frameworks supporting multi-agent LLM systems

Framework	Focus/Features
LangGraph	Graph-based orchestration, shared state, dynamic routing
AutoGen	Agent self-organization, negotiation, adaptive collaboration
CrewAI	Structured workflows, strict typed interfaces (Pydantic), high-fidelity data passing
BeeAI	Enterprise-grade modular orchestration, uses IBM ACP

LangGraph multi-agent workflow essentials

Core concepts

- **Directed graph nodes:** represent agents/tasks
- **Edges:** control flow between agents
- **Shared state:** a TypedDict or similar, passed and updated by all agents
- **Routing logic:** dynamically determines the next agent based on the state

---

Example of shared state definition

```
from typing import TypedDict, Optional, List
class SalesReportState(TypedDict):
    request: str
    raw_data: Optional[dict]
    processed_data: Optional[dict]
    chart_config: Optional[dict]
    report: Optional[str]
    errors: List[str]
    next_action: str
```

Example of agent node skeleton

```
def data_collector_agent(state: SalesReportState) -> SalesReportState:
    # Collect raw data based on state['request']
    state['raw_data'] = {...}
    state['next_action'] = 'process'
    return state
```

Repeat for other agents: `data_processor_agent`, `chart_generator_agent`, `report_generator_agent`, `error_handler_agent`.

Example of routing function

```
def route_next_step(state: SalesReportState) -> str:
    routing = {
        "collect": "data_collector",
        "process": "data_processor",
        "visualize": "chart_generator",
        "report": "report_generator",
        "error": "error_handler",
        "complete": "END"
    }
    return routing.get(state.get("next_action", "collect"), "END")
```

Building the workflow graph

```
from langgraph.graph import StateGraph, END
def create_workflow():
    workflow = StateGraph(SalesReportState)
    workflow.add_node("data_collector", data_collector_agent)
    workflow.add_node("data_processor", data_processor_agent)
    workflow.add_node("chart_generator", chart_generator_agent)
    workflow.add_node("report_generator", report_generator_agent)
    workflow.add_node("error_handler", error_handler_agent)
    # Define conditional edges based on routing decisions
    workflow.add_conditional_edges("data_collector", route_next_step, {...})
    # Repeat for other nodes...
    workflow.set_entry_point("data_collector")
    return workflow.compile()
```

Running the workflow

```
def run_workflow():
    app = create_workflow()
    initial_state = SalesReportState(
        request="Q1-Q2 Sales Report",
        raw_data=None,
```

```
        processed_data=None,
        chart_config=None,
        report=None,
        errors=[],
        next_action="collect"
    )
    final_state = app.invoke(initial_state)
    return final_state
```

Agentic RAG systems

- Combine **Retrieval**, **Reasoning**, and **Verification** using specialized agents
- Retrieval agent fetches relevant knowledge/data
- Reasoning agent performs inference and decision-making
- Verification agent checks results for accuracy and consistency
- Multi-agent design improves reliability and trustworthiness

---

Best practices & challenges

Challenge	Strategy
Context management	Share only relevant info, avoid overload
Granularity	Balance agent count — not too few or too many
Communication cost	Optimize message size and frequency
Error handling	Implement fallback, retries, and error agents

Author

[Karan Goswami]

Skills Network