From Retrievers to Agents

The Problem

You know three retrieval patterns:

- **Vector**: Semantic content search
- **Vector Cypher**: Content + relationships
- **Text2Cypher**: Precise facts

But users don't know about retriever types.

They just ask questions:

- "What is Apple's strategy?"
- "How many companies are in the database?"
- "Which asset managers own companies facing cyber risks?"

The Solution: Agents

What is an Agent?

In AI terms, an agent has **four components**:

Component	What It Does
Perception	Receives input (questions, history, tool descriptions)
Reasoning	Analyzes the question and decides what to do
Action	Executes the selected tool(s)
Response	Returns output in natural language

Tools: How Agents Take Action

Action involves calling tools.

Tools are capabilities the agent can use—functions it can call to get information or perform tasks.

- During **Perception**, the agent sees what tools are available
- During **Reasoning**, it decides which tool fits the question
- During **Action**, it executes the tool

How Agents Choose Tools

The agent matches questions to tool descriptions:

Question: "How many companies are there?"

Tool descriptions:

- `get_graph_schema` : "Get database structure..."
- `search_content` : "Search for content about topics..."
- `query_database` : "Get answers to factual questions, counts..."

Agent reasons: "How many" → count → `query_database`

Retrievers as Tools

Your retrievers become tools:

Tool	Based On	When Agent Uses It
Schema Tool	Graph introspection	"What data exists?"
Semantic Search	Vector Retriever	"What is...", "Tell me about..."
Database Query	Text2Cypher	"How many...", "List all..."

Each tool has a description that tells the agent when to use it.

The ReAct Pattern

Agents follow **ReAct** (Reasoning + Acting):

```
1. Receive question: "How many risk factors does Apple face?"
2. Reason: "This asks for a count"
3. Act: Call Database Query Tool
4. Observe: Result = 45
5. Respond: "Apple faces 45 risk factors."
```

For complex questions, the agent may loop through multiple cycles.

Multi-Tool Example

Question: "What are Apple's main risks and which investors are affected?"

Agent process:

1. **Reason:** Need risk content AND investor relationships
2. **Act:** Call Semantic Search for Apple's risks
3. **Observe:** Risk descriptions
4. **Reason:** Now need investors
5. **Act:** Call Database Query for Apple's investors
6. **Observe:** Investor list
7. **Respond:** Combine both into comprehensive answer

Why Agents Matter

Without agents:

- Build separate interfaces for each retriever
- Force users to choose which retriever to use
- Complex user experience

With agents:

- Users ask natural questions
- System figures out how to answer
- Conversational, intuitive experience

An Example: The GraphRAG Agent

An example of an agent is a GraphRAG system that:

1. **Receives** a user question
2. **Analyzes** what kind of question it is
3. **Selects** and executes the appropriate tool(s)
4. **Synthesizes** results into a coherent answer

Your retrievers become **tools** the agent can use.

Summary

In this lesson, you learned:

- **Agents** have four components: Perception, Reasoning, Action, Response
- **Tools** are capabilities agents use to take action
- **Selection** happens through semantic matching to tool descriptions
- **ReAct pattern**: Reason → Act → Observe → Respond
- **Result**: Users ask naturally; agents figure out how to answer

Next: Learn about the Microsoft Agent Framework.

The Microsoft Agent Framework

Framework Components

The Microsoft Agent Framework provides infrastructure for building agents:

Component	Purpose
AzureAIClient	Connects to Microsoft Foundry services
ChatAgent	Pre-built agent with tool selection
Tools	Python functions with docstrings
Threads	Conversation history for multi-turn

Creating an Agent

```
async with client.create_agent(
    name="graphrag-agent",
    instructions="You are a helpful assistant that answers questions
                  about a knowledge graph.",
    tools=[get_graph_schema, search_content, query_database],
) as agent:
    async for update in agent.run_stream(query):
        if update.text:
            print(update.text, end="", flush=True)
```

Key elements:

- `instructions` : Agent's purpose and behavior
- `tools` : List of callable functions
- `run_stream` : Execute and stream responses

How Tool Selection Works

Tools are Python functions with descriptive docstrings:

```
def get_graph_schema() -> str:
    """Get the schema of the graph database including node labels,
    relationships, and properties."""
    return get_schema(driver)
```

The framework:

1. Reads function name and docstring
2. Registers tool with agent
3. Matches questions to tool descriptions
4. Calls the best-matching tool

Why Docstrings Matter

Vague docstring (poor selection):

```
def search(query: str) -> str:
    """Search the database."""
```

Specific docstring (accurate selection):

```
def search_content(query: str) -> str:
    """Search for content about topics and concepts using semantic search.
    Use for questions like 'What is...', 'Tell me about...', 'Explain...'"""
```

The agent reads docstrings to decide when to use each tool.

The ReAct Loop

The framework implements ReAct automatically:

```
User: "How many companies are in the database?"
    ↓
Agent thinks: "This asks for a count—use database query tool"
    ↓
Agent calls: query_database("How many companies...")
    ↓
Agent observes: Result = 523
    ↓
Agent responds: "There are 523 companies in the database."
```

This loop can iterate multiple times for complex questions.

Agent Instructions

Instructions guide agent behavior:

Generic (less effective):

```
"You are a helpful assistant."
```

Specific (more effective):

```
"You are a helpful assistant that answers questions about a knowledge
graph containing SEC filings data. You have three tools:

1. get_graph_schema - Use for questions about data structure
2. search_content - Use for semantic questions about topics
3. query_database - Use for specific facts, counts, and lookups

Choose the appropriate tool based on the question type."
```

Thread Management

For multi-turn conversations:

```
# Create a thread for conversation context
thread = agent.get_new_thread()

# First message
result1 = await agent.run("What companies are in the database?", thread=thread)

# Follow-up maintains context
result2 = await agent.run("Tell me more about the first one", thread=thread)
```

The thread preserves conversation history.

Streaming Responses

For real-time output:

```
async for update in agent.run_stream(query):
    if update.text:
        print(update.text, end="", flush=True)
```

Benefits:

- Users see responses as they're generated
- Better experience (no waiting)
- Can observe tool selection in progress

Summary

The Microsoft Agent Framework provides:

- **AzureAIClient** for Microsoft Foundry integration
- **ChatAgent** for pre-built agent functionality
- **Automatic tool selection** based on docstrings
- **ReAct pattern** for reasoning and acting
- **Thread management** for conversations
- **Streaming** for real-time responses

Next: Build your agent progressively—starting with one tool.

Building Your Agent

Progressive Enhancement

Build agents progressively:

1. **Start with one tool**: Understand how agents use tools
2. **Add semantic search**: Handle content questions
3. **Add database queries**: Handle factual questions

Each step adds capability while keeping the system testable.

Tool 1: Schema Introspection

Purpose: Let users explore what data exists.

When the agent uses it:

- "What types of data are in this database?"
- "What relationships exist?"
- "What can I ask about?"

```
def get_graph_schema() -> str:
    """Get the schema of the graph database including node labels,
    relationships, and properties."""
    return get_schema(driver)
```

Why start here: Simple, deterministic, helps users understand possibilities.

Tool 2: Semantic Search

Purpose: Find content about topics and concepts.

Based on: Vector Retriever from Lab 5.

When the agent uses it:

- "What is Apple's business strategy?"
- "Tell me about cybersecurity threats"
- "What risks do tech companies face?"

```
def search_content(query: str) -> str:
    """Search for content about topics using semantic search.
    Use for 'What is...', 'Tell me about...', 'Explain...'"""
    return vector_retriever.search(query)
```

Tool 3: Database Queries

Purpose: Answer factual questions with precision.

Based on: Text2Cypher Retriever from Lab 5.

When the agent uses it:

- "How many companies are in the database?"
- "What companies does BlackRock own?"
- "Which company has the most risk factors?"

```
def query_database(query: str) -> str:
    """Query the database for specific facts, counts, lists, and relationships.
    Use for 'How many...', 'List all...', 'Who owns...'"""
    return text2cypher_retriever.search(query)
```

How the Agent Chooses

With three tools, the agent matches questions to descriptions:

Question	Agent Reasoning	Tool
"What data types exist?"	About structure	Schema
"What is Apple's strategy?"	Content/concept	Semantic Search
"How many companies?"	Count/fact	Database Query
"Tell me about AI"	Topic exploration	Semantic Search
"Who owns Microsoft?"	Relationship fact	Database Query

Example: Schema Question

Question: "What types of data are in this database?"

```
Agent thinks: "Asking about data structure"
Agent calls: get_graph_schema()
Agent observes: "Node labels: Company, RiskFactor, Product..."
Agent responds: "The database contains Companies, Risk Factors,
                 Products, Executives, and their relationships..."
```

Example: Content Question

Question: "What does Apple say about AI?"

```
Agent thinks: "Asking about content on a topic"
Agent calls: search_content("Apple AI")
Agent observes: Chunks about Apple's AI initiatives
Agent responds: "Apple discusses AI in several contexts,
                 including privacy-focused machine learning..."
```

Example: Fact Question

Question: "How many risk factors does Apple face?"

```
Agent thinks: "Asking for a count"
Agent calls: query_database("How many risk factors does Apple face?")
Agent observes: "45"
Agent responds: "Apple faces 45 risk factors according to
                 their SEC filing."
```

Testing Your Agent

Test each tool type:

Schema questions:

- "What entities exist?"
- "What relationships are there?"

Content questions:

- "What is [topic]?"
- "Tell me about [concept]"

Fact questions:

- "How many [entities]?"
- "Who [relationship] [entity]?"

Common Issues

Problem	Solution
Wrong tool selected	Improve docstrings to be more specific
Overlapping tools	Ensure each tool has distinct purpose
No tool selected	Make sure a tool covers the question type
Poor results	Check underlying retrievers work correctly

The Complete Agent

With all three tools, your agent handles:

Category	Tool	Examples
Structure	Schema	"What data exists?"
Content	Semantic Search	"What is...?", "Tell me about..."
Facts	Database Query	"How many...?", "List all..."

This covers the full range of GraphRAG queries.

Summary

- **Build progressively:** Start with one tool, add more
- **Three tools:** Schema (structure), Semantic Search (content), Database Query (facts)
- **Selection via docstrings:** Clear descriptions guide correct choice
- **Test incrementally:** Verify each tool before adding more

Next: Learn design patterns for effective multi-tool agents.

Agent Design Patterns

Pattern 1: Tool Specialization

Each tool should have a **distinct, non-overlapping purpose**.

Bad design (overlapping):

```
Tool 1: "Search for companies"
Tool 2: "Find company information"
Tool 3: "Look up companies"
```

Good design (specialized):

```
Tool 1: "Explore database structure and schema"
Tool 2: "Search content semantically about topics"
Tool 3: "Query specific facts, counts, relationships"
```

Pattern 2: Descriptive Signatures

Tool names and docstrings guide selection. Be specific.

Vague:

```
def search(query: str) -> str:
    """Search the database."""
```

Specific:

```
def search_content_semantically(query: str) -> str:
    """Search for content about topics using semantic similarity.

    Use this tool when the user asks:
    - "What is..." or "Tell me about..."
    - Questions about strategies or concepts

    Do NOT use for counting or listing."""
```

Pattern 3: Tool Composition

Complex questions may need multiple tools in sequence.

Question: "What are Apple's main risks and which investors are affected?"

Agent process:

1. Semantic search for Apple's risk content
2. Database query for investors owning Apple
3. Synthesize both results

Agent instructions can help:

```
"For complex questions requiring both content and facts,
use multiple tools and combine results."
```

Pattern 4: Graceful Fallbacks

Handle empty results and errors gracefully.

Poor handling:

```
User: "What is XYZ Corp's strategy?"
Agent: Error: No results found.
```

Graceful handling:

```
User: "What is XYZ Corp's strategy?"
Agent: "I couldn't find specific information about XYZ Corp.
       You might want to check if XYZ Corp is in the database,
       or try a different company name."
```

Pattern 5: Clear Error Messages

When tools can't answer, explain *why*.

Unhelpful:

```
"I don't know."
```

Helpful:

```
"I searched for content about quantum computing but found no relevant
documents. The database primarily contains SEC filings which may not
discuss this topic in detail."
```

This helps users understand system limitations and refine their questions.

Anti-Pattern 1: Too Many Tools

Don't overwhelm the agent:

```
tools = [
    get_schema, get_nodes, get_relationships,
    search_companies, search_products, search_risks,
    count_companies, count_products, count_risks,
    list_companies, list_products, list_risks,
    ...  # 20+ tools
]
```

Problems: Decision paralysis, high token costs, more errors.

Rule of thumb: 3-7 tools is optimal.

Anti-Pattern 2: Tools That Do Everything

Don't create super-tools:

```
def do_everything(query, mode, options, filters):
    """Does everything: schema, search, queries, aggregations..."""
    if mode == "schema":
        return get_schema()
    elif mode == "search":
        return vector_search(query)
    # ... 100 more lines
```

Problems: Agent doesn't know when to use it, hard to debug.

Anti-Pattern 3: Vague Boundaries

Avoid ambiguous overlap:

```
def search_documents(query):
    """Search for information in documents"""

def find_information(query):
    """Find information about topics"""
```

When are they different? Agent can't tell.

Fix: Consolidate or clearly differentiate.

The GraphRAG Sweet Spot

Three specialized tools:

Tool	Purpose
Schema	Structure understanding
Semantic Search	Content discovery
Database Query	Precise facts

Why this works:

- Clear non-overlapping purposes
- Covers all major question types
- Easy for agent to choose correctly
- Low token overhead

Production Considerations

Security:

- Use read-only database credentials
- Validate generated queries
- Limit result sizes

Performance:

- Cache common queries
- Set appropriate timeouts
- Consider parallel tool execution

Monitoring:

- Log which tools are selected

Summary

Patterns:

- **Tool Specialization:** Non-overlapping purposes
- **Descriptive Signatures:** Clear docstrings guide selection
- **Tool Composition:** Multiple tools for complex questions
- **Graceful Fallbacks:** Handle empty results informatively
- **Clear Error Messages:** Explain *why* something failed

Anti-Patterns:

- Too many tools (>10)
- Tools that do everything
- Vague/overlapping boundaries

Sweet spot: 3 specialized tools (schema, semantic search, database query)

Congratulations

What You've Accomplished

You've completed the GraphRAG workshop:

Lab 3: Building Knowledge Graphs

- LLM limitations and why context matters
- Transforming documents into knowledge graphs
- Schema design, chunking, entity resolution, vectors

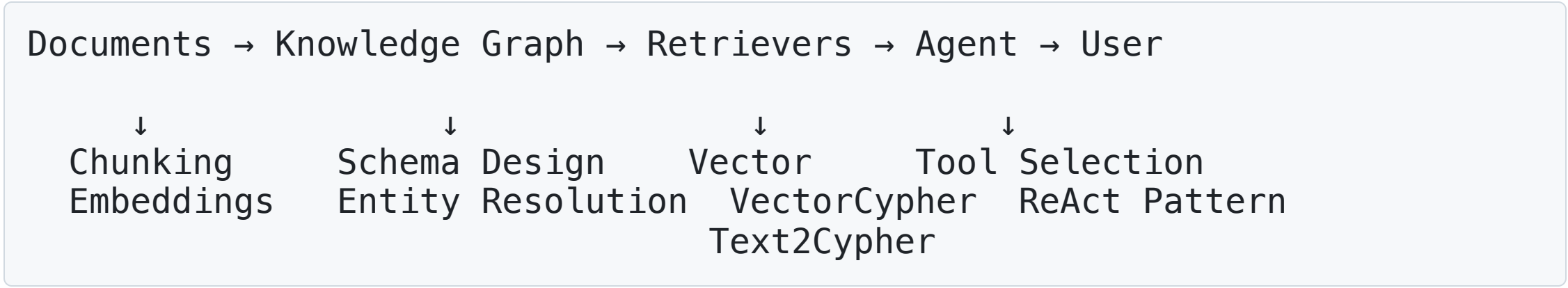
Lab 5: GraphRAG Retrievers

- Three retrieval patterns: Vector, Vector Cypher, Text2Cypher
- When to use each pattern

Lab 6: Intelligent Agents

- Agents that choose tools automatically

The Complete Picture



Each component plays a role in the complete system.

Key Takeaways

1. Structure Enables Intelligence

Traditional RAG treats documents as isolated blobs. GraphRAG extracts structure—entities, relationships—that enables relationship-aware retrieval.

2. Different Questions Need Different Approaches

- Semantic questions → Vector Retriever
- Relationship-aware questions → Vector Cypher Retriever
- Factual questions → Text2Cypher Retriever

3. Agents Automate Selection

Instead of forcing users to choose, agents analyze questions and select tools automatically.

What You Built

A complete GraphRAG system:

- **Knowledge Graph** from SEC filings with companies, risks, products, executives
- **Three retrieval patterns** for different question types
- **Intelligent agent** that chooses the right tool for each question
- **Conversational interface** that handles natural questions

Where to Go Next

Neo4j Aura Agents:

No-code agent creation through the Aura web interface.

Advanced Topics:

- Graph embedding models
- Multi-hop reasoning
- Agent memory and long-term context
- Evaluation frameworks

Resources:

- Neo4j GraphRAG Python Documentation
- Microsoft Agent Framework Documentation
- Neo4j Graph Academy

The Foundation

You've built the foundation for intelligent, context-aware AI applications.

GraphRAG combines:

- **Language model power** for understanding and generation
- **Knowledge graph structure** for precise, relationship-aware retrieval

Together, they answer questions that neither could handle alone.

Thank You

Take what you've learned and build something great.