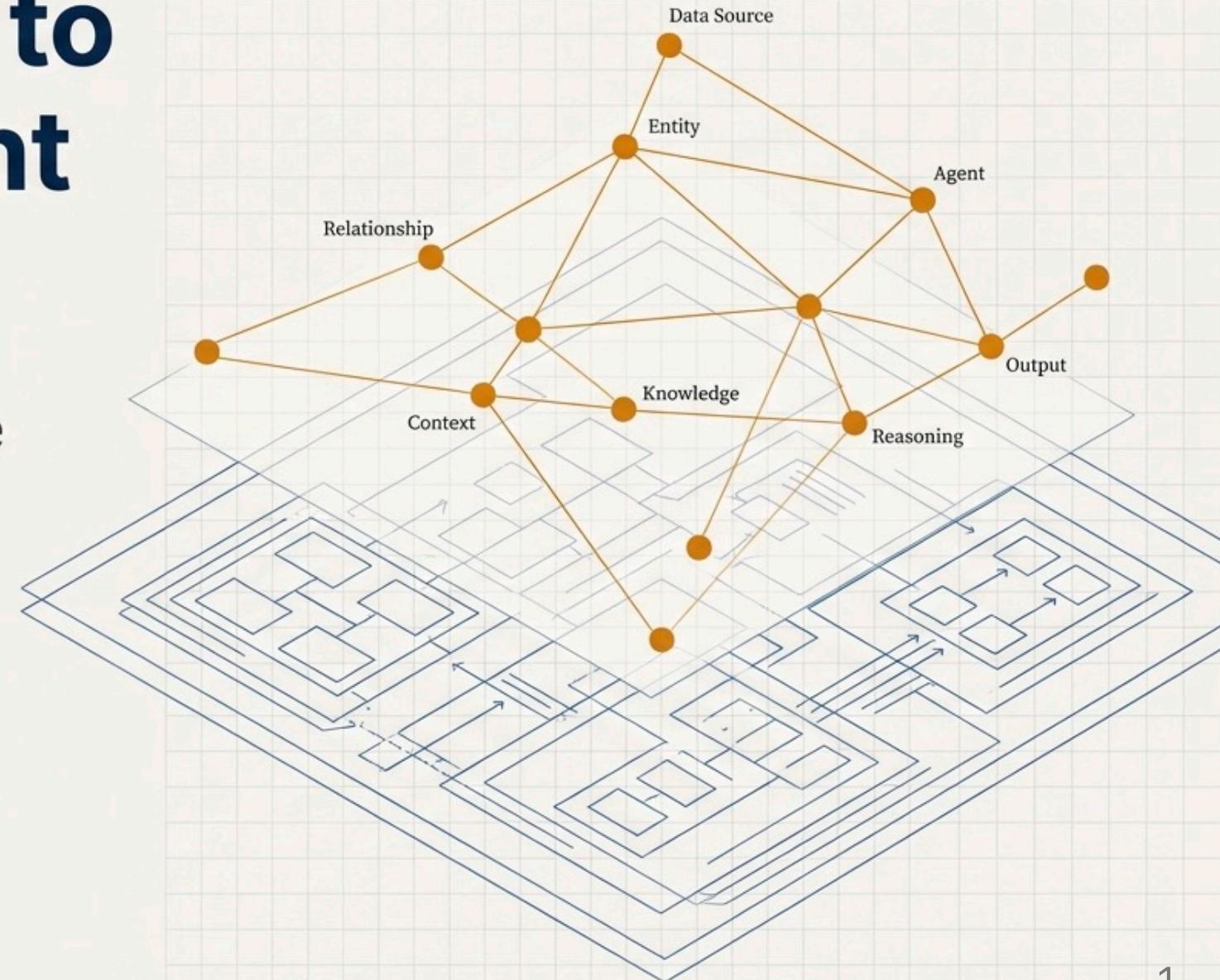


# From Raw Data to Intelligent Agent

A Blueprint for Building  
Production-Ready AI with  
GraphRAG on Neo4j and Azure



# Some of the Neo4j's customers who run on Azure



# **Neo4j Aura: Cloud Graph Database**

# What is Neo4j Aura?

Neo4j Aura is a **fully managed cloud graph database service** that eliminates the operational overhead of running a graph database.

## Key Characteristics:

- **Fully managed** - No infrastructure to maintain
- **Scalable** - Automatically scales with your data and queries
- **Secure** - Enterprise-grade security and compliance
- **Available everywhere** - Deploy in AWS, GCP, or Azure regions

# Why Use a Graph Database?

Traditional databases struggle with **connected data**:

Scenario	Relational DB	Graph DB
"Find friends of friends"	Complex JOINs, slow	Natural traversal, fast
"What impacts what?"	Multiple queries	Single query
"How are these connected?"	Hard to express	Native pattern matching

**Graphs excel at relationship-heavy queries** that would require dozens of JOINs in SQL.

# The Value of Aura for AI/GenAI

Neo4j Aura provides unique capabilities for building AI applications:

## GraphRAG Foundation:

- Store knowledge graphs that power AI agents
- Vector search for semantic similarity
- Graph traversal for relationship reasoning

## Production-Ready:

- Built-in vector indexes for embeddings
- Cypher query language for complex retrieval
- APIs for integration with LLM frameworks

# Graph Analytics in Explore

The **Explore** tool includes built-in graph algorithms for visual analysis:

**Available in Explore:**

Category	Algorithms
Centrality	Betweenness, Degree, Eigenvector, PageRank
Community Detection	Label Propagation, Louvain, Weakly Connected Components

**Full Algorithm Library (65+):**

Neo4j Aura Graph Analytics provides the complete library via serverless compute with Zero ETL:

Category	Additional Algorithms	Use Cases
Similarity	Node Similarity, K-Nearest Neighbors	Recommendations, duplicate detection
Path Finding	Dijkstra, A*, Yen's K-Shortest	Routing, supply chain optimization
Link Prediction	Common Neighbors, Adamic Adar	Predict future connections
Node Embeddings	FastRP, GraphSAGE, Node2Vec	ML feature generation

# Aura Tools: Query Workspace

The **Query Workspace** is a developer-friendly environment for Cypher:

## Core Features:

- Write and execute Cypher queries against your database
- Syntax highlighting and auto-completion
- Save and organize query collections
- Export results in multiple formats

## Query Log Forwarding:

- Send logs to your cloud logging service
- Better compliance, monitoring, and operational visibility
- Manage directly from Aura console

# Aura Tools: Explore

**Explore** (powered by Neo4j Bloom) is a visual graph exploration tool:

## Visual Graph Scene:

- Interactive canvas showing your graph data
- Click and drag nodes to arrange layouts
- Export as PNG, CSV, or shareable scenes

## Search-First Experience:

- Natural language and pattern-based search
- "Show me a graph" sample queries
- Find nodes and relationships without Cypher

## AI-Powered Features:

- GenAI Copilot for query assistance
- Find hidden connections automatically

# Aura Tools: Dashboards

Dashboards in the Neo4j Console provide data visualization capabilities with low code / no code:

## Visualization Types:

- Bar charts, line charts, pie charts, etc.
- Geographic maps
- **3D graph visualizations** (WebGL-powered)

## GenAI Copilot:

- AI-powered dashboard creation
- Natural language to visualization

## Enterprise Ready:

- SSO integration
- Role-based access

# Aura Agents: No-Code GraphRAG

**Coming up in this workshop:** Neo4j Aura Agents

Aura Agents let you build **AI-powered conversational interfaces** to your graph:

- **No code required** - Configure through a simple UI
- **Natural language queries** - Ask questions in plain English
- **Automatic Cypher generation** - LLM translates questions to graph queries
- **Knowledge graph reasoning** - Leverage relationships for better answers

**Why Agents matter:**

- Democratize access to graph insights
- Build chatbots that understand your domain
- Combine vector search + graph traversal automatically

# Summary

Neo4j Aura provides:

- **Managed graph database** - Focus on your data, not infrastructure
- **Graph-native storage** - Relationships are first-class citizens
- **AI/GenAI capabilities** - Vector indexes, GraphRAG support
- **Graph Analytics** - Built-in algorithms for insights
- **Integrated Tools** - Query, Explore, and Dashboards
- **Aura Agents** - No-code conversational AI over your graph

Next: Learn about Aura Agents for no-code GraphRAG applications.

# The GenAI Promise and Its Limits

## What Generative AI Does Well

LLMs excel at tasks that rely on pattern recognition and language fluency:

- **Text generation:** Creating human-like responses, summaries, explanations
- **Language understanding:** Parsing intent, extracting meaning, following instructions
- **Pattern completion:** Continuing sequences, filling in blanks, generating variations
- **Translation and transformation:** Converting between formats, styles, languages

These capabilities emerge from training on vast amounts of text data.

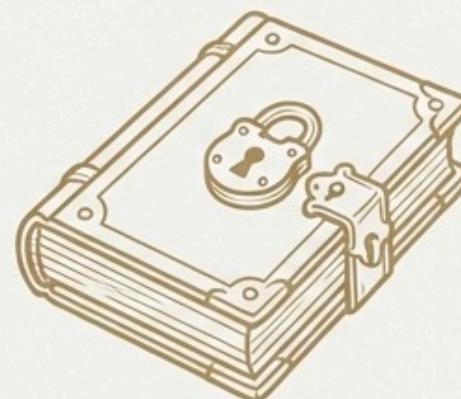
# The Three Gaps in the GenAI Promise

LLMs excel at text generation, language understanding, and pattern completion. However, production systems must address three fundamental limitations.



## Hallucination (Confident but Wrong)

LLMs generate statistically *probable* text, not factually *verified* text. When they don't know, they fabricate with full confidence.



## Knowledge Cutoff (Your Data is Invisible)

Models are trained on public data with a cutoff date. They have no access to your private company documents, databases, or real-time events.



## Relationship Blindness (Can't Connect the Dots)

LLMs process text sequentially and struggle to answer questions that require reasoning over relationships across multiple documents or sources.

# 1. Hallucination: Confident But Wrong

LLMs generate responses based on statistical likelihood, not factual verification.

## The Problem:

- Produces the most *probable* continuation, not the most *accurate*
- Doesn't say "I don't know"—generates plausible-sounding text instead
- Complete with fabricated details and citations

**Real Example:** In 2023, US lawyers were sanctioned for submitting an LLM-generated brief with six fictitious case citations.

## 2. Knowledge Cutoff: No Access to Your Data

LLMs are trained at a specific point in time on publicly available data.

**They don't know:**

- Recent events after their training cutoff
- Your company's documents, databases, or internal knowledge
- Real-time data: current prices, live statistics, changing conditions

**The Risk:** Ask about your Q3 results or last week's board meeting, and the LLM may still generate a confident (and wrong) response.

### **3. Relationship Blindness: Can't Connect the Dots**

LLMs process text sequentially and treat each piece in isolation.

**Questions they struggle with:**

- "Which asset managers own companies facing cybersecurity risks?"
- "What products are mentioned by companies that share risk factors?"
- "How are these two companies connected through their executives?"

These questions require *reasoning over relationships*—connecting entities across documents and traversing chains of connections.

## The Solution: Providing Context

All three limitations have a common solution—**providing context**.

When you give an LLM relevant information in its prompt:

- It has facts to work with (reduces hallucination)
- It can access your specific data (overcomes knowledge cutoff)
- You can structure that information to show relationships (enables reasoning)

This is the foundation of **Retrieval-Augmented Generation (RAG)**.

# Summary

In this lesson, you learned about the fundamental limitations of LLMs:

- **Hallucination:** LLMs generate probable responses, not verified facts
- **Knowledge cutoff:** LLMs can't access recent events or your private data
- **Relationship blindness:** LLMs struggle with cross-document reasoning

The solution is providing context—which leads us to RAG.

**Next:** Learn how traditional RAG works and why it has its own limitations.

# Traditional RAG: Chunking and Vector Search

# Why RAG Was Adopted

Remember the LLM limitations we discussed:

- **Hallucination** - Generates confident but wrong information
- **Knowledge cutoff** - No access to your private data
- **Relationship blindness** - Can't connect information

**The insight:** If we could provide LLMs with relevant context, we could address these limitations.

This led to **Retrieval-Augmented Generation (RAG)**.

# The Power of Context

Providing context in prompts dramatically improves LLM responses.

**When you include relevant information, the model can:**

- Generate accurate summaries grounded in actual documents
- Answer questions about your specific data
- Reduce hallucination by having facts to reference

**RAG automates this:** Instead of manually adding context, retrieve it automatically based on the user's question.

# How Traditional RAG Works

Traditional RAG follows a simple pattern:

- 1. Index documents:** Break documents into chunks and create embeddings
- 2. Receive query:** User asks a question
- 3. Retrieve context:** Find chunks with embeddings similar to the query
- 4. Generate response:** Pass retrieved chunks to LLM as context

Let's understand each component: chunking, embeddings, and vector search.

# Why Chunking Matters

LLMs have **context window limits**—they can only process so much text at once.

**The problem:**

- Documents can be thousands of pages
- You can't send everything to the LLM
- You need to find the *relevant* parts

**The solution:** Break documents into smaller **chunks** that can be:

- Indexed for search
- Retrieved when relevant
- Fit within the LLM's context window

# Common Chunking Strategies

Strategy	How It Works	Best For
Fixed-size	Split every N characters/tokens	Simple, predictable
Sentence	Split on sentence boundaries	Preserving complete thoughts
Paragraph	Split on paragraph breaks	Structured documents
Semantic	Split when topic changes	Long-form content
Recursive	Try multiple strategies in order	General purpose

**Overlap** between chunks helps preserve context at boundaries.

# From Words to Meaning: The Power of Embeddings

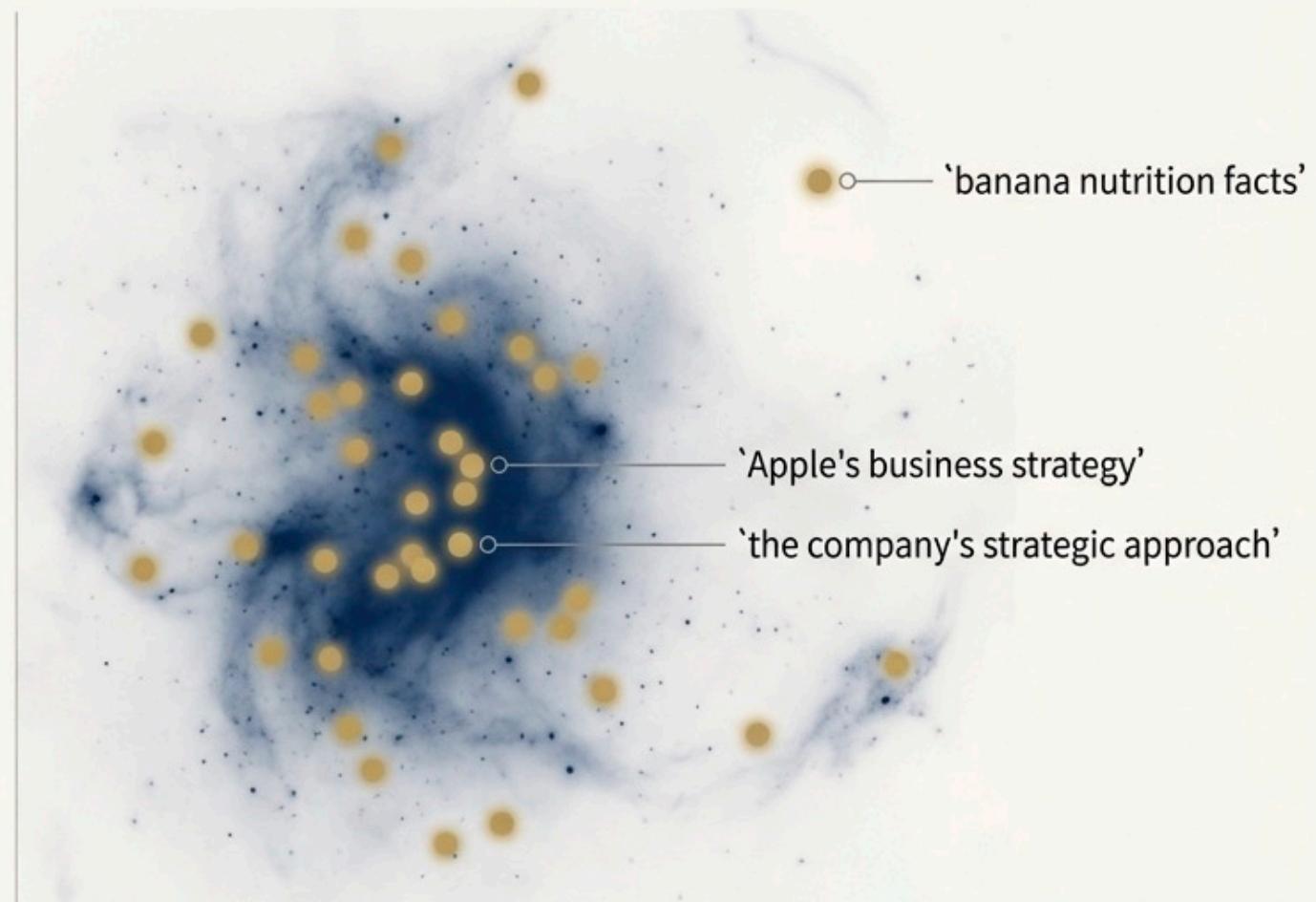
## Key Concept

Embeddings are numerical representations of text as high-dimensional vectors (e.g., 1,536 dimensions for OpenAI models).

## Core Principle

The most important property of embeddings is that **similar meanings produce similar vectors**.

## GALAXY OF MEANING (2D Projection of Semantic Space)



The phrases 'Apple's business strategy' and 'the company's strategic approach' have very different words, but their embeddings are nearly identical because their meaning is the same.

## What is a Vector?

Vectors are lists of numbers.

The vector [1, 2, 3] represents a point in three-dimensional space.

In machine learning, vectors can represent much more complex data—including the *meaning* of text.

# The Smart Librarian Analogy

Think of embeddings like having a **really smart librarian** who has read every book in the library.

## Traditional catalog (keywords):

- Books organized by title, author, subject
- Search for "dogs" only finds books with "dogs" in the title/subject
- Miss books about "canines," "puppies," or "pets"

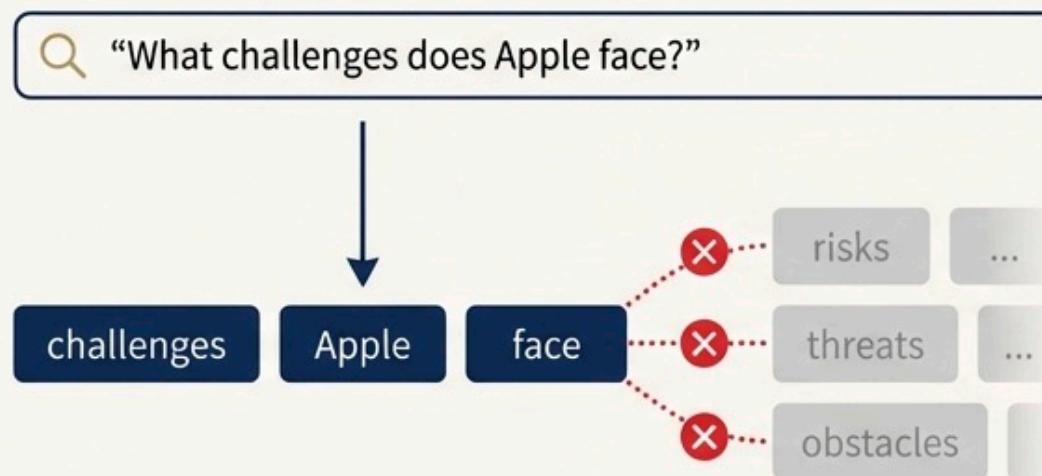
## Smart librarian (embeddings):

- Understands what each book is *about*
- "I want something about loyal companions" → finds dog books, even without the word "dog"
- Organizes by meaning, not just labels

# Beyond Keywords: Searching by Meaning

## Without Vectors (Keyword Search)

### Brittle Keyword Matching

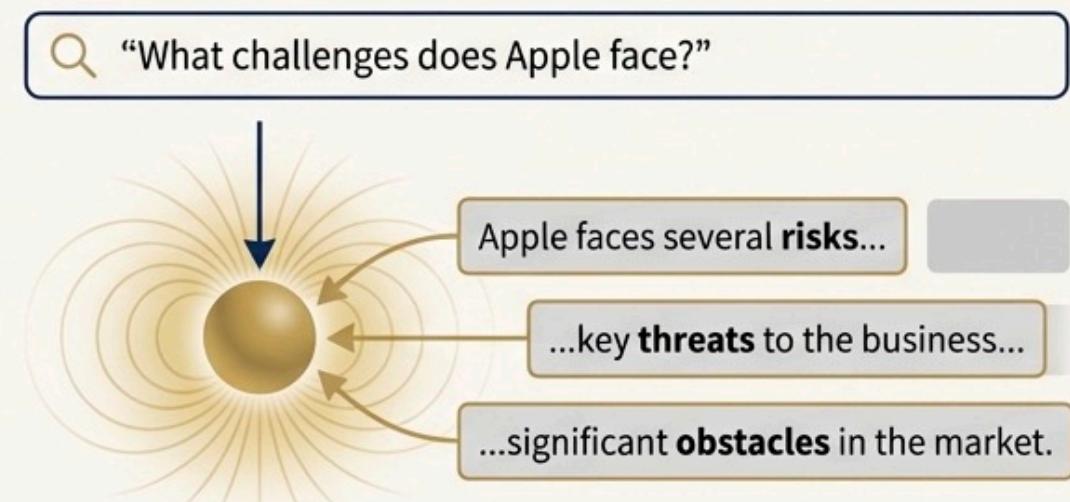


**Logic:** The system requires exact keyword matches.

**✗** Fails to find relevant documents that use synonyms, leading to incomplete or no results.

## With Vectors (Semantic Search)

### Flexible Semantic Search



**Logic:** The system converts the query to an embedding and finds text chunks with similar *meaning*.

**✓** Successfully retrieves chunks discussing 'risks,' 'threats,' and 'obstacles,' providing a comprehensive answer regardless of the specific words used.

# Similarity Search

Vector similarity is typically measured by **cosine similarity**—the angle between two vectors:

Score	Meaning
Near 1.0	Very similar meanings
Near 0.5	Somewhat related
Near 0.0	Unrelated

When you search, your question becomes an embedding, and the system finds chunks with embeddings close to your question.

# The RAG Retrieval Flow

User Question



Create embedding of question



Compare to all chunk embeddings



Return top K most similar chunks



Send chunks + question to LLM



LLM generates answer using chunks as context

# **Traditional RAG: What It Enables**

**Works well for:**

- "What does this document say about X?"
- Finding relevant passages by topic
- Answering questions within a single document

**The foundation of modern AI assistants**, but as we'll see, it has important limitations when dealing with connected information.

# Summary

In this lesson, you learned:

- **Chunking** breaks documents into searchable pieces
- **Embeddings** encode text meaning as vectors
- **Similar meanings** produce similar vectors
- **Semantic search** finds relevant content by meaning, not keywords
- **Traditional RAG** combines these to provide context to LLMs

**Next:** Understanding the limits of traditional RAG and how GraphRAG addresses them.

# The Limits of Traditional RAG

# RAG Helps, But Introduces New Challenges

We've seen how RAG provides context to LLMs:

- Retrieves relevant chunks based on semantic similarity
- Grounds responses in actual documents
- Reduces hallucination

**But traditional RAG also introduced new problems:**

- Retrieves similar content, not necessarily *relevant* content
- Misses relationships between pieces of information
- Can actually make responses worse when context is poor

# The Problem with Traditional RAG

Traditional RAG treats documents as isolated, unstructured blobs.

**What traditional RAG sees:**

Chunk 1: "Apple Inc. faces cybersecurity risks including..."

Chunk 2: "BlackRock Inc. holds shares in technology companies..."

Chunk 3: "The semiconductor supply chain impacts..."

**What traditional RAG misses:**

- Which specific companies does BlackRock own?
- Do any of those companies face cybersecurity risks?
- How are supply chain issues connected to specific products?

## **Retrieves Similar Content, Not Connected Information**

Traditional RAG can find text about cybersecurity and text about BlackRock.

**But it can't tell you:**

- Which asset managers are exposed to cybersecurity risks through their holdings

**Why?** Each chunk is independent—there's no understanding of how information connects.

# Context ROT: When More Context Makes Things Worse

A surprising discovery: **too much irrelevant context degrades LLM performance.**

**What happens:**

- RAG retrieves chunks that are *similar* but not truly *relevant*
- The LLM's context window fills with tangentially related information
- The model gets confused, distracted, or misled by the noise

This became known as "**Context ROT**" (Retrieval of Tangents)—the retrieved context actually *rots* the quality of the response.

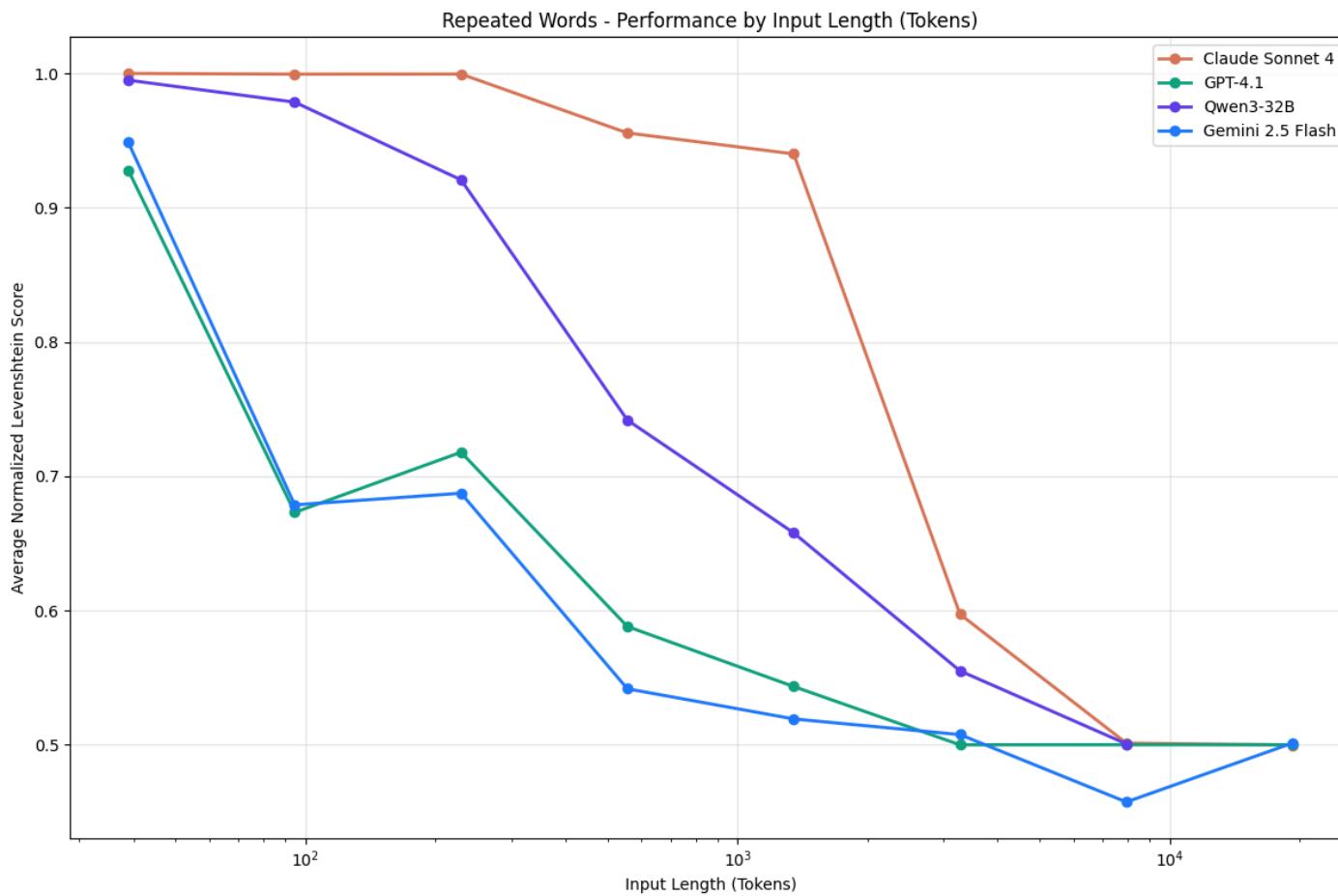
# Context ROT: The Research

Research shows that as irrelevant context increases, LLM accuracy **decreases dramatically**.

The graph shows how adding more retrieved chunks often hurts rather than helps.

**Key insight:** Quality of context matters more than quantity.

Source: Chroma Research - Context ROT



# Questions Traditional RAG Can't Answer

Question	Why Traditional RAG Struggles
"Which asset managers own companies facing cyber risks?"	Requires connecting ownership data to risk mentions
"What products are mentioned by companies that share risk factors?"	Requires finding shared entities across documents
"How many companies mention supply chain issues?"	Requires aggregation, not similarity search
"What executives work for companies in the tech sector?"	Requires traversing entity relationships

These questions need *structured context* that preserves relationships.

# From Unstructured to Structured

**The core insight:** Information isn't truly unstructured.

Documents contain:

- **Entities:** Companies, people, products, risks
- **Relationships:** Owns, faces, mentions, works for

Traditional RAG ignores this structure. It treats a document as a bag of words to embed and search.

# The GraphRAG Solution

GraphRAG extracts structure, creating a *knowledge graph* that preserves:

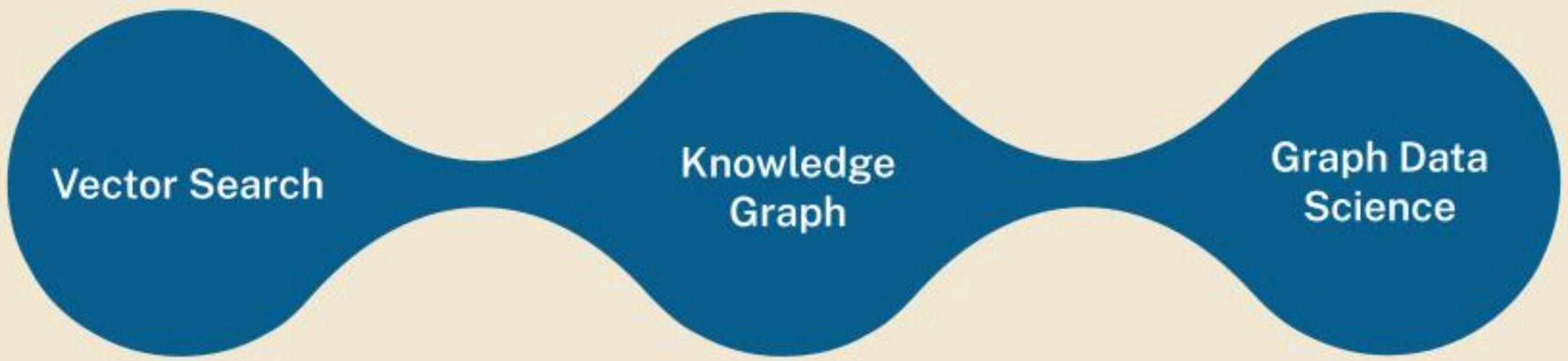
- **Entities:** The things mentioned in documents
- **Relationships:** How those things connect
- **Properties:** Attributes and details about entities

**Traditional RAG asks:** "What chunks are similar to this query?"

**GraphRAG asks:** "What entities and relationships are relevant to this query?"

# RAG with Neo4j - graphRAG

Unify vector search, knowledge graph and data science capabilities to improve RAG quality and effectiveness



Find similar documents  
and content

Identify entities associated  
to content and patterns  
in connected data

Improve GenAI inferences  
and insights. Discover new  
relationships and entities

# GraphRAG Patterns



## Vector & Hybrid Search

basic vector/full-text search  
graph-enhanced  
graph/metadata filtering



## Query Templates (Deterministic)

Cypher templates  
pattern matching



## GraphRAG



## Query Generation

dynamic Cypher gen.  
text2Cypher



## Graph Enrichment

community summaries  
graph embeddings  
pagerank

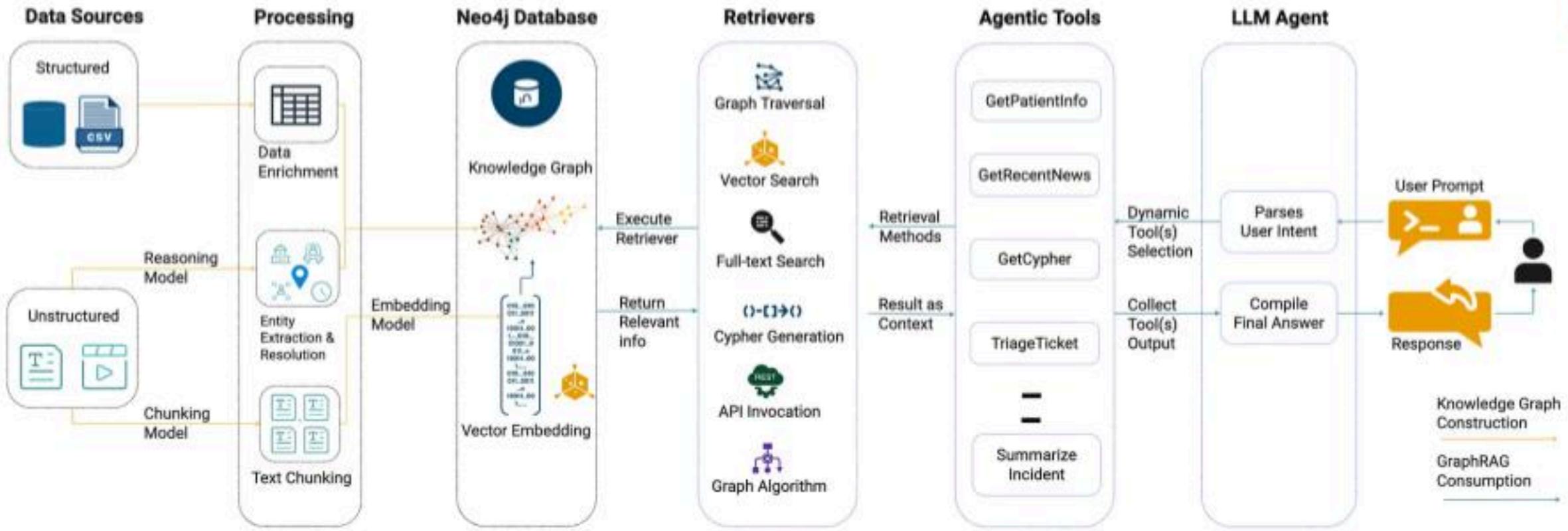


## Agent Brain

Memory  
Reasoning & Planning  
Decision & Execution



# Agentic Tools



# Summary

In this lesson, you learned:

- **Traditional RAG helps** but introduces new challenges
- **Context ROT:** Poor retrieval can make responses worse than no retrieval
- **The limitation:** Traditional RAG treats documents as isolated blobs, missing relationships
- **Questions requiring relationships** can't be answered with similarity search alone
- **GraphRAG** extracts structure from documents, preserving entities and relationships

**Next:** See how this applies to SEC filings and the knowledge graph you'll explore.

# The SEC Filings Knowledge Graph

# The SEC Filings Example

Throughout this workshop, you'll work with a knowledge graph built from SEC 10-K filings.

**These documents contain:**

- Companies and their business descriptions
- Risk factors they face
- Financial metrics they report
- Products and services they mention
- Executives who lead them

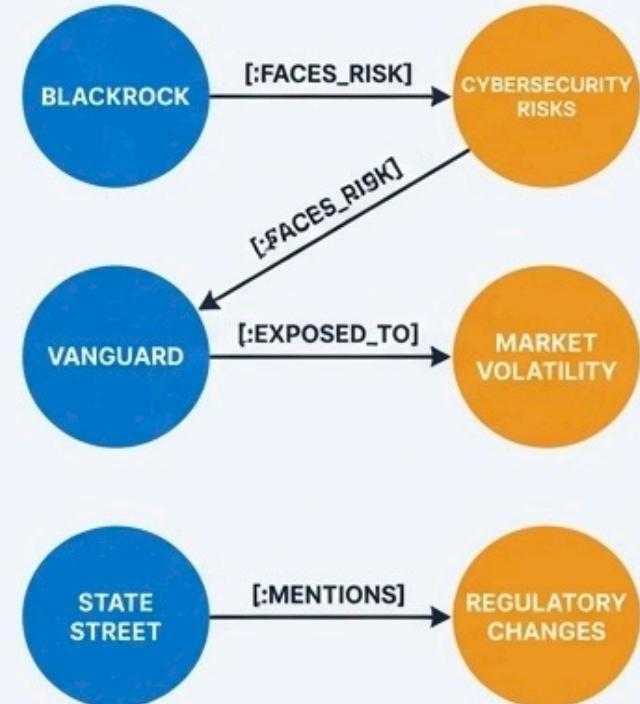
# The Core Insight: Structure is the Key to Intelligence

Unstructured Text (SEC Filings)



GraphRAG

Structured Knowledge Graph



Documents aren't just bags of words. They are containers for entities and the relationships between them. Traditional RAG throws away this inherent structure. **GraphRAG extracts, preserves, and leverages it.** By transforming unstructured text into a structured Knowledge Graph, we enable the AI to reason over a network of connected facts, not just a list of similar text snippets. This is the foundational shift that unlocks deeper, more accurate insights.

# What the Graph Enables

Question Type	How the Graph Helps
"What risks does Apple face?"	Traverse FACES_RISK relationships
"Which companies mention AI?"	Find MENTIONS relationships to AI products
"Who owns Apple?"	Follow OWNS relationships from asset managers
"How many risk factors are there?"	Count RiskFactor nodes

# The Pre-Built Knowledge Graph

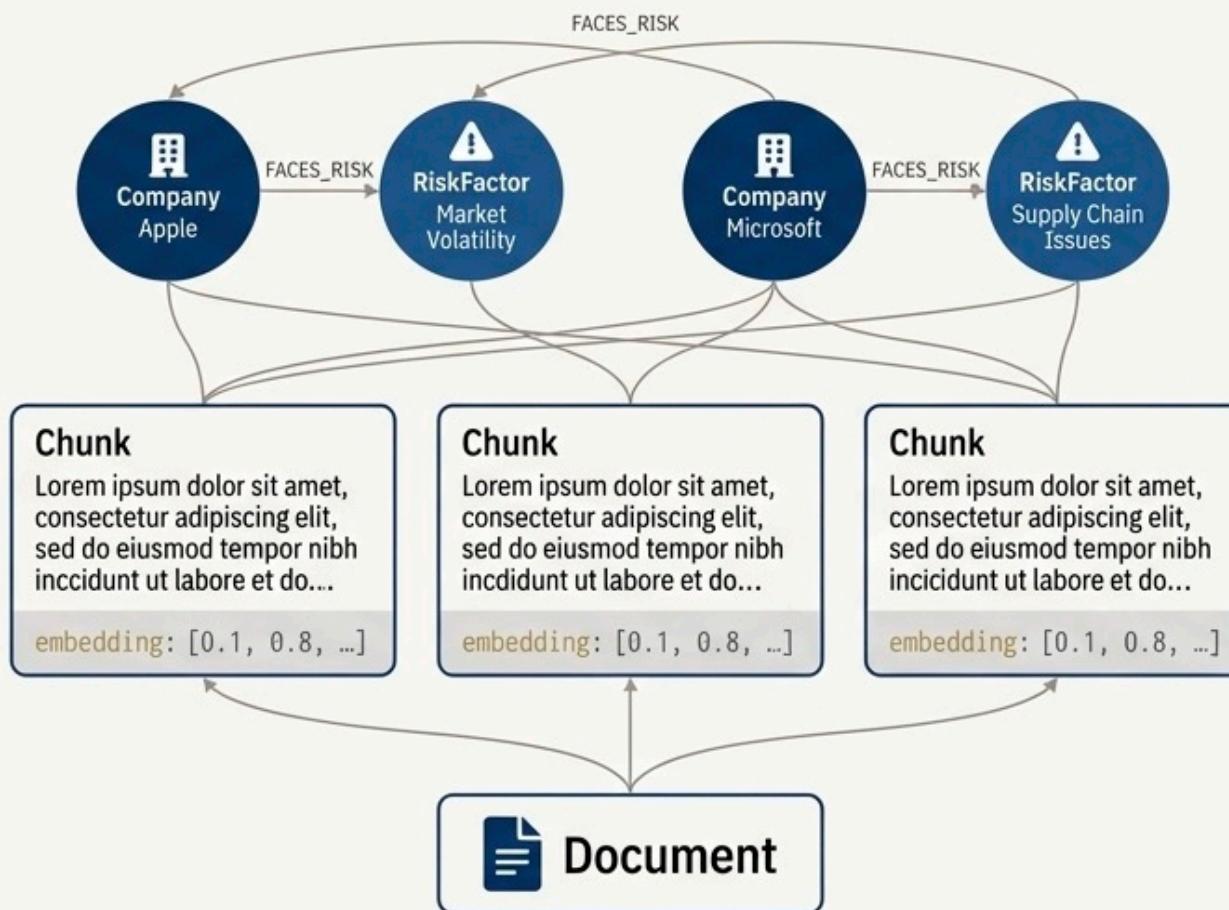
The graph you restored to your Aura instance has already been processed:

**What was done (Lab 3 covers this in detail):**

- SEC 10-K filing PDFs were ingested
- Documents were chunked into smaller pieces
- An LLM extracted entities and relationships
- Vector embeddings were generated for semantic search

**You get the finished result ready for exploration with Aura Agents.**

# The Complete Knowledge Graph: Structure Meets Meaning



Your graph now contains:

-  **Structured Entities & Relationships:** From schema-driven extraction.
-  **Appropriately Sized Chunks:** A result of your chunking strategy.
-  **Resolved Entities:** Canonical nodes with no duplicates.
-  **Vector Embeddings:** Enabling semantic search on all text content.

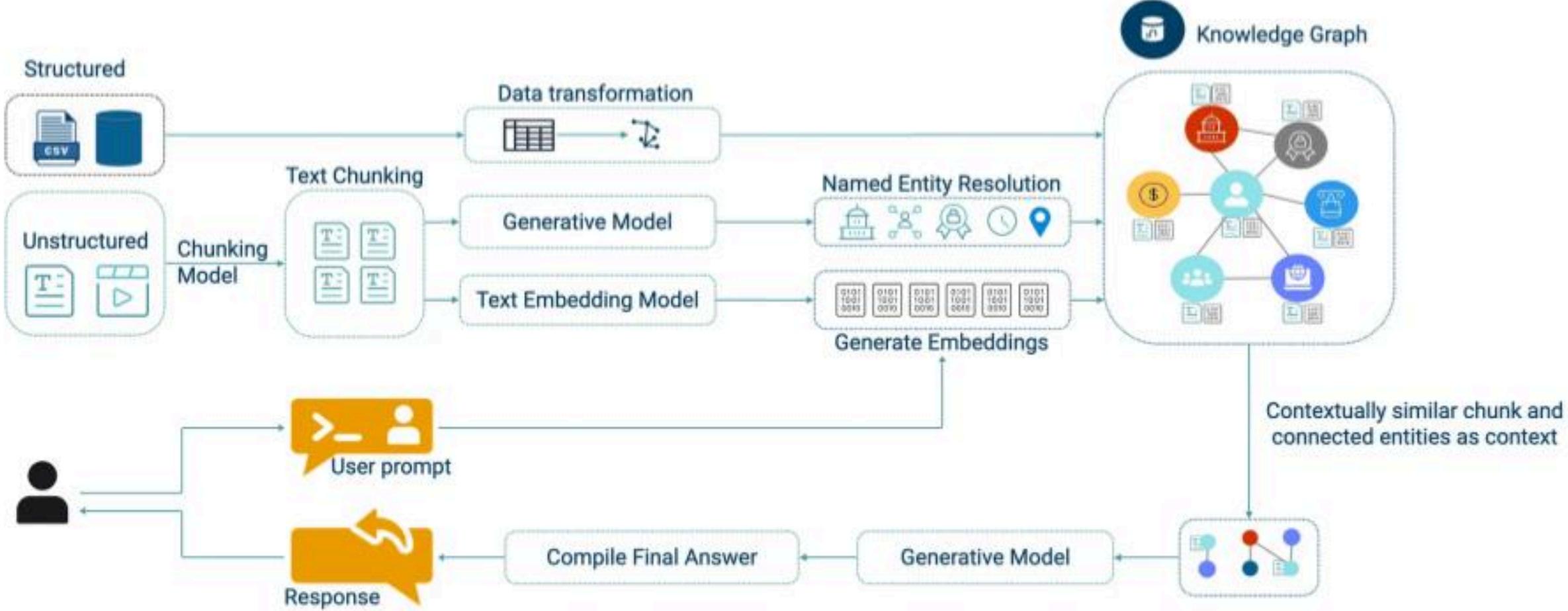
This unified structure enables sophisticated retrieval patterns that leverage both structured connections and semantic meaning.

# The Processing Pipeline (Preview)

This is what Lab 3 covers in depth:

Step	What Happens
Document Ingestion	Read SEC 10-K filing PDFs
Chunking	Break into smaller pieces for processing
Entity Extraction	LLM identifies companies, products, risks
Relationship Extraction	LLM finds connections between entities
Graph Storage	Save entities and relationships to Neo4j
Vector Embeddings	Generate embeddings for semantic search

# Building the Knowledge Graph



# **Ready to Explore with Aura Agents**

With the knowledge graph pre-built, you can now:

- 1. Create an Aura Agent** connected to your graph
- 2. Configure retrieval tools** (Cypher templates, similarity search, Text2Cypher)
- 3. Ask complex questions** that combine graph traversal and semantic search
- 4. See how GraphRAG works** before diving into the code

**Next:** Build your SEC Filings Analyst agent with Aura Agents.

# Neo4j Aura Agents

Building GraphRAG Applications Without Code

# What Are Aura Agents?

Aura Agents is a **no-code platform** for building AI agents grounded in your Neo4j graph data.

## Key Capabilities:

- Build agents directly in the Aura Console
- Configure retrieval tools without writing code
- Test in an integrated chat playground
- Deploy to production via API

# The Three-Tool Architecture

Aura Agents combine three complementary retrieval methods:

Tool	Purpose	Best For
Cypher Templates	Precise, parameterized queries	Known question patterns
Similarity Search	Semantic vector search	Finding relevant content
Text2Cypher	Natural language to Cypher	Ad-hoc exploration

The agent **automatically selects** the right tool for each question.

# Tool 1: Cypher Templates

Pre-defined queries with parameters for common questions.

## Example: Company Overview Tool

```
MATCH (c:Company {name: $company_name})
OPTIONAL MATCH (c)-[:FILED]->(d:Document)
OPTIONAL MATCH (c)-[:FACES_RISK]->(r:RiskFactor)
RETURN c.name AS company,
       collect(DISTINCT r.name) [0..10] AS risks
```

User asks: "Tell me about Apple's risks"

Agent: Extracts "APPLE INC", executes template, returns structured answer

## Tool 2: Similarity Search

Semantic search using vector embeddings stored in Neo4j.

### How It Works:

1. User question is embedded into a vector
2. Vector index finds similar content
3. Retrieval query enriches with graph context
4. Agent synthesizes the response

**Example Question:** "What do companies say about AI and machine learning?"

Returns semantically relevant passages with company context.

## Tool 3: Text2Cypher

Converts natural language to Cypher queries dynamically.

### How It Works:

1. User asks a question in plain English
2. LLM generates appropriate Cypher query
3. Query executes against the database
4. Results are returned to the agent

**Example:** "Which company has the most risk factors?"

```
MATCH (c:Company)-[:FACES_RISK]->(r:RiskFactor)
RETURN c.name, count(r) AS risk_count
ORDER BY risk_count DESC LIMIT 1
```

# Why Graph Context Matters

Traditional RAG retrieves text chunks. GraphRAG retrieves **connected knowledge**.

## Vector-Only RAG:

- "Find documents about cybersecurity risks"
- Returns: text chunks mentioning cybersecurity

## GraphRAG with Aura Agents:

- "Which asset managers own companies facing cybersecurity risks?"
- Returns: traverses OWNS and FACES\_RISK relationships

**Graphs enable questions that pure vector search cannot answer.**

# Creating an Aura Agent

Building an agent takes minutes:

- 1. Define the agent** - Name, description, system instructions
- 2. Select your database** - Connect to your AuraDB instance
- 3. Add tools** - Configure Cypher templates, similarity search, Text2Cypher
- 4. Test in playground** - Interact with your agent
- 5. Deploy** - Get an authenticated API endpoint

No coding required – just configuration.

# Agent Configuration Example

**Agent Name:** SEC Filings Analyst

**System Instructions:**

You are an expert financial analyst specializing in SEC 10-K filings analysis. You help users understand:

- Company risk factors and comparisons
- Asset manager ownership patterns
- Financial metrics and products
- Relationships between entities

Ground your responses in actual data from SEC filings.

# Testing Your Agent

The Aura Console provides an integrated chat playground:

## Cypher Template Test:

"What risks do Apple and Microsoft share?"

- Agent selects `find_shared_risks` template
- Executes with company parameters
- Returns comparison results

## Semantic Search Test:

"What do filings say about supply chain?"

- Agent uses similarity search tool
- Finds relevant passages across companies

# Tool Selection in Action

The agent reasons about which tool to use:

Question	Tool Selected	Why
"Tell me about NVIDIA"	Cypher Template	Matches known pattern
"Find content about climate"	Similarity Search	Semantic search needed
"How many products total?"	Text2Cypher	Aggregation query
"Compare Apple and Google risks"	Cypher Template	Two-company comparison

The agent explains its reasoning in each response.

# Deployment Options

## Internal Testing:

- Share with team members in Aura Console
- Test different configurations

## Production API:

- Deploy to authenticated REST endpoint
- Client credentials from user profile
- Integrate into your applications

## Coming Soon:

- MCP (Model Context Protocol) support
- Additional integration protocols

# From No-Code to Code

Aura Agents demonstrates the same patterns you'll implement programmatically:

Aura Agent Tool	Python Implementation
Cypher Template	Parameterized queries
Similarity Search	VectorCypherRetriever
Text2Cypher	Text2CypherRetriever
Agent orchestration	Microsoft Agent Framework

**Labs 5 and 6** implement these patterns in Python.

# The Value of Aura Agents

## For Prototyping:

- Validate GraphRAG approach quickly
- Test retrieval strategies without code
- Iterate on prompts and tools

## For Production:

- Deploy agents in minutes
- Secure, authenticated endpoints
- Scalable infrastructure

# Summary

Aura Agents provide:

- **No-code GraphRAG** - Build agents without programming
- **Three retrieval tools** - Cypher templates, similarity search, Text2Cypher
- **Intelligent orchestration** - Automatic tool selection
- **Easy deployment** - Test in playground, deploy to API

The fastest path from knowledge graph to AI agent.

## **Next Steps**

- 1. Create your Aura instance (if not done already)**
- 2. Build the knowledge graph (Labs 2-3)**
- 3. Create an Aura Agent using the console**
- 4. Test with sample questions**
- 5. Continue to Python implementation (Labs 5-6)**

# **Neo4j Graph Workload for Microsoft Fabric**

Bringing Graph Analytics to Your OneLake Data

# What is Microsoft Fabric?

Microsoft Fabric is a **unified analytics platform** that brings together all data and analytics tools in one place.

## Key Components:

- **Data Engineering** - Build data pipelines and transformations
- **Data Warehousing** - SQL-based analytics at scale
- **Data Science** - Machine learning and AI workloads
- **Real-Time Analytics** - Streaming data processing
- **Power BI** - Business intelligence and visualization

**Everything runs on a unified foundation.**

# What is OneLake?

OneLake is the **single, unified data lake** for all of Microsoft Fabric.

## Key Characteristics:

- **One copy of data** - No data duplication across services
- **Open format** - Delta Lake / Parquet for interoperability
- **Unified governance** - Single security and compliance model
- **Automatic organization** - Data organized by workspace

Think of OneLake as the "**OneDrive for data**" - all your organizational data in one place.

# What is a Fabric Workload?

A **Workload** is a specialized capability that extends Microsoft Fabric's functionality.

## Native Workloads:

- Data Factory, Synapse, Power BI, etc.

## Partner Workloads:

- Third-party tools integrated natively into Fabric
- Access OneLake data directly
- Use Fabric's security, identity, and compute
- Appear as first-class experiences in the Fabric portal

**Partner workloads extend Fabric without leaving the platform.**

# How Workloads Use OneLake

Partner workloads integrate deeply with the Fabric ecosystem:

Integration Point	What It Provides
Entra ID	Seamless authentication and authorization
Workspaces	Object persistence and sharing
Lakehouses	Secure access to tabular data
Fabric Capacity	In-platform computing resources

**Workloads operate on your data where it lives.**

# Introducing Neo4j Graph Workload

Neo4j provides a **native graph analytics workload** for Microsoft Fabric.

## What It Enables:

- Transform tabular OneLake data into graph models
- Run graph algorithms and analytics
- Visualize and explore data relationships
- Query with Cypher directly in Fabric

**Available now as a first-class Fabric experience.**



# Graph Intelligence Natively in Microsoft Fabric

## Bring graph intelligence into Fabric workflows

- **Reveal hidden insights in Fabric OneLake data:** Use advanced graph algorithms to analyze connected relationships across your existing structured data.
- **Easily create graphs:** Transform relational data into graphs in a few clicks, with AI assist graph data modelling.
- **Simplified experience:** Import, query, visualize and explore your graph data natively in Microsoft Fabric.
- **Fully managed:** Scalable Fabric experience that grows with your analytics needs, without managing infrastructure.
- **Consumption pricing:** Monthly subscription, and the option to pay for running graph algorithms on demand.
- **Keep analytics in sync:** Schedule Spark jobs to seamlessly move data in and out of OneLake tables.

# Testimonials

"We really like the seamless login experience all the way to the AuraDB database because it makes the whole Fabric experience effortless for our users."

*- Preview Customer*

"The generative AI assistance for creating the graph is a game changer for getting started if you are new to graphs."

*- Preview Customer*

Customers are “starting to see Fabric as their strategic data platform. This workload gives us the ability to offer our customers a graph solution that integrates into Fabric.”

*- Neo4j & Microsoft Partner*

# Fully Managed, Secure, Scalable Environment



Simplify operations with fully managed scaling and lifecycle management



Streamline procurement by running directly in Fabric and purchase through Azure Marketplace



Strengthen security with Azure enterprise controls and Neo4j's node-level access control

# Why Graph Analytics on OneLake Data?

Your OneLake data contains **hidden relationships** that tabular analysis misses.

Analysis Type	Traditional BI	Graph Analytics
Customer segments	Static groupings	Community detection
Influence patterns	Not possible	Centrality algorithms
Connected risks	Manual joins	Path traversal
Recommendations	Rule-based	Similarity algorithms
Fraud detection	After the fact	Pattern matching

**Graph analytics reveals the connections in your data.**

# Neo4j Graph Workload Capabilities

## Transform & Load:

- Map tabular data from Lakehouses to graph models
- Create nodes and relationships from your data
- Load into a Neo4j AuraDB Professional database

## Analyze:

- Full Cypher query language
- 65+ built-in graph algorithms
- Centrality, community detection, similarity, path finding

## Visualize:

- Explore interface for interactive graph visualization
- Discover patterns visually

# Available Graph Algorithms

The Neo4j Graph Workload includes the full algorithm library:

Category	Algorithms	Use Cases
Centrality	PageRank, Betweenness, Degree, Eigenvector	Find influential nodes
Community	Louvain, Label Propagation, WCC	Detect clusters and groups
Similarity	Node Similarity, KNN	Recommendations, deduplication
Path Finding	Dijkstra, A*, Shortest Path	Routing, dependencies
Link Prediction	Common Neighbors, Adamic Adar	Predict future connections
Embeddings	FastRP, Node2Vec, GraphSAGE	ML feature generation

Run any algorithm directly from the Fabric console.

# Find Graph Intelligence in the Workload Hub

The screenshot shows the Microsoft Fabric Workloads Hub interface. On the left is a vertical sidebar with icons for Home, Create, Monitor, Audit Trail, Workloads, and My workspace. The main area is titled "Workloads" and displays eight workload cards:

- 2TEST (preview)** by Celonis: Comprehensive Quality Assurance: Automated Testing and Data Quality checks. Add button.
- Informatica Cloud Data Quality (preview)** by Informatica: Informatica's Intelligent Data Management Cloud (IDMC) is the leading platform for cloud data management with Microsoft Fabric, offering data... Add button.
- Lumel EPM** by Lumel: Accelerate AI readiness by consolidating Planning, Analytics & Data Apps in Microsoft Fabric. Add button.
- Osmos** by Osmos: Accelerate your Fabric deployment with Osmos AI data agents. Seamlessly ingest, transform, and structure data using agents AI. Add button.
- Process Intelligence (preview)** by Celonis: Establish a zero-copy integration with Celonis to enhance your data with process intelligence and expose Celonis' unique class of data and context in... Add button.
- Profisee MDM** by Profisee: Match, merge, and standardize data in Microsoft Fabric to create trusted, consumption-ready data products for analytics and AI. Add button.
- Quantexa Unify (preview)** by Quantexa: Resolve one or more Microsoft OneLake Data Sources to produce a complete 360° view with unprecedented accuracy. Add button.
- SAS Decision Builder (preview)** by SAS Institute Inc.: Automate, optimize, and scale decision-making processes. Manage complex business rules, integrate machine learning models, or use Python code to... Add button.
- Statsig (preview)** by Statsig: Visualize and Analyse your data with the Statsig Data Platform. Add button.
- Teradata AI Unlimited (preview)** by Teradata: Power faster innovation with Teradata. Teradata AI Unlimited serverless engine via Microsoft Fabric accelerates innovation. Add button.
- Zebra AI (preview)** by Zebra BI: Zebra AI automatically understands, cleans, analyses and visualizes data using AI and best practice dashboarding. Add button.
- Neo4j Graph Intelligence** by Neo4j: Discover hidden patterns in your OneLake data with graph queries and algorithms. Learn more.

# 3 steps to finding Graph Powered Insights from Tabular Data

## Select Lakehouse



Choose any OneLake lakehouse in the same region as Fabric Capacity.

## Create Graph



Select the tables you want to transform into a graph model, Generative AI Assist will propose a data model that users can adjust before performing the import.

## Analyze Graph



Business users and analysts can explore the graph and run graph algorithms on the data without using code.

# Table to Graph Conversion

Transform relational data into nodes and relationships.

## Graph Model Interface:

- Guides the data mapping process visually
- Preview graph structure before loading

## AI-Assisted Modeling:

- AI Assistant suggests node and relationship types for review
- Generative AI uses schema analysis and Azure OpenAI for suggestions
- Accept, modify, or reject recommendations

## Execution:

- A Spark job executes the transformation into graph objects

# Example: Customer 360 Graph

Transform customer data into a connected view:

## Source Tables:

- `customers` - Customer profiles
- `orders` - Purchase history
- `products` - Product catalog
- `interactions` - Support tickets, emails

## Graph Model:

```
(Customer)-[:PURCHASED]->(Product)  
(Customer)-[:CONTACTED]->(Support)  
(Product)-[:SIMILAR_TO]->(Product)
```

**Insights:** Community detection reveals customer segments, PageRank identifies influential customers.

# **Explore: Visual Graph Analysis**

The **Explore** interface provides interactive visualization:

## **Visual Capabilities:**

- Interactive graph canvas
- Drag and arrange nodes
- Filter and highlight patterns
- Export visualizations

## **Search & Discovery:**

- Pattern-based search
- Find connections between entities
- Expand neighborhoods

**Built into Fabric - no external tools needed.**

# 3 ways to analyze the graph

## Explore



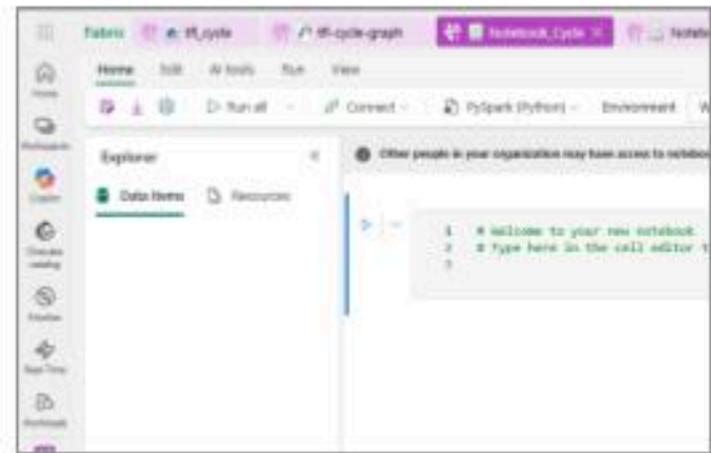
**No coding required** - business users and analysts can point & click to explore the graph and run graph algorithms.

## Query



**AI-powered Cypher queries** - no longer just for power users who know Cypher, data analysts can use text to Cypher to write GQL Cypher and run algorithms.

## Notebooks



**Fabric Notebooks** - data scientists who prefer Python can run algorithms from their Notebooks, create spark jobs and write results back to OneLake.

# Integration with Fabric Security

The Neo4j Graph Workload inherits Fabric's enterprise security:

## Authentication:

- Microsoft Entra ID (Azure AD)
- Single sign-on across Fabric

## Authorization:

- Workspace-level permissions
- Role-based access control

## Data Governance:

- Data stays in your tenant
- Audit logging

# Use Cases for Graph Analytics on OneLake

## Supply Chain:

- Map supplier relationships
- Identify bottlenecks and risks
- Optimize logistics paths

## Financial Services:

- Detect fraud patterns
- Analyze transaction networks
- Risk propagation analysis

## Manufacturing:

- Bill of materials graphs
- Quality issue tracing
- Asset relationship mapping

# Current Capabilities & Roadmap

## Available Now:

- Graph creation from Lakehouse tables
- Full Cypher query support
- 65+ graph algorithms
- Explore visualization

## Coming Soon:

- Writeback of insights to OneLake
- Additional data source support

# Summary

Neo4j Graph Workload for Microsoft Fabric:

- **Native Integration** - First-class workload in the Fabric portal
- **Zero ETL** - Transform OneLake data directly to graphs
- **Full Analytics** - 65+ algorithms, Cypher queries, visualization
- **Enterprise Ready** - Inherits Fabric security and governance
- **One-Click Setup** - Install from the Workload Hub

**Unlock the relationships hidden in your OneLake data.**

# Learn More

## Resources:

- [Neo4j Graph Analytics for Microsoft Fabric](#)
- [Neo4j Fabric Workload Documentation](#)
- [Announcing the Neo4j Graph Workload](#)

## Next Steps:

1. Install the Neo4j workload in your Fabric workspace
2. Connect to a Lakehouse with your data
3. Create your first graph model
4. Run algorithms and explore connections

# What is an AI Agent?

# The Evolution of AI Assistants

From simple Q&A to intelligent agents:

Generation	Capability	Limitation
Chatbots	Pre-scripted responses	No real understanding
LLMs	Natural language understanding	No access to external data
RAG	Retrieval + Generation	Single retrieval strategy
Agents	Reasoning + Tools + Action	Full autonomy

# What is an Agent?

An agent is an LLM with access to **tools**.

Component	What It Does
Reasoning	LLM analyzes the question and decides what to do
Tools	Capabilities the agent can call (search, query, etc.)
Action	Executes the selected tool(s) and returns results

# How Agents Use Tools

Agents take action by calling tools—functions that get information or perform tasks.

How tool selection works:

Question	Agent Reasoning	Tool Selected
"How many companies?"	Needs a count	query_database
"What risks does Apple face?"	Needs content search	search_content
"What's in the database?"	Needs structure	get_graph_schema

The agent matches the question to the best tool description.

# 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 capability
- Force users to choose which tool to use
- Complex user experience

## With agents:

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

# Agents vs Traditional Software

Aspect	Traditional Software	AI Agents
Input	Structured commands	Natural language
Logic	Pre-programmed rules	LLM reasoning
Flexibility	Fixed workflows	Dynamic tool selection
Adaptation	Code changes needed	Learns from context

# 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 Model Context Protocol (MCP).

# **What is MCP?**

## **Model Context Protocol**

# The Problem: Tool Integration

Every AI framework has its own way of defining tools:

- **OpenAI**: Function calling with JSON schemas
- **Anthropic**: Tool use with specific formats
- **LangChain**: Tool wrappers and chains
- **Custom Agents**: Proprietary integrations

**Result:** Building tools once means rebuilding for each framework.

# The Solution: MCP

**Model Context Protocol (MCP)** is an open standard for connecting AI models to external tools and data sources.

Think of it as **USB for AI tools**:

USB	MCP
Standard port for devices	Standard protocol for AI tools
Plug any device into any computer	Plug any tool into any AI model
One cable, many devices	One server, many clients

## MCP vs Standard APIs: Who Is It Built For?

The biggest difference is **who the server is built for**.

Standard API (REST/GraphQL)	MCP Server
Built for Programmers	Built for AI Models
Assumes you've read the manual	Assumes the AI knows nothing
You must know endpoints exist	Server tells AI what's available
/users , /invoices , etc.	"Here are 5 things I can do"

## "Read the Manual" vs "Ask Me"

**Standard API:** Passive. If you don't know the magic words, you get an error.

**MCP Server:** Active and self-documenting. Discovery is built-in.

AI: "Hello, what can you do?"

MCP Server: "I can read these 3 files (Resources), and I have a tool called query\_graph that requires a cypher parameter (Tool)."

AI: "Great, I'll use that."

**The AI doesn't need documentation—it asks the server directly.**

# MCP Architecture

How an Agent fits in:

Component	Role	Example
Agent	The AI client hosted in Foundry	Foundry Agent
Tools	Capabilities the agent can use	MCP Servers
MCP Server	Provides tools via the protocol	Neo4j MCP Server

The Agent has tools—MCP servers are one way to provide those tools.

# What MCP Servers Provide

MCP servers can expose:

- **Tools** - Functions the AI can call (query database, search, etc.)
- **Resources** - Data the AI can read (files, database schemas)
- **Prompts** - Pre-defined prompt templates

## Benefits of MCP

Benefit	Description
<b>Standardization</b>	One protocol for all tools
<b>Reusability</b>	Build once, use everywhere
<b>Security</b>	Controlled access to resources
<b>Composability</b>	Combine multiple servers
<b>Ecosystem</b>	Growing library of servers

# **Neo4j MCP Server**

The **Neo4j MCP Server** is an official MCP implementation that gives AI agents the ability to:

- **Explore** your graph schema
- **Read** data using Cypher queries
- **Write** data (only when explicitly enabled)

It's the bridge between natural language questions and graph database answers.

## Core Tools Provided

The Neo4j MCP Server exposes three main tools:

Tool	Purpose	Default
<code>get_neo4j_schema</code>	Retrieve graph structure	Always enabled
<code>read_neo4j_cypher</code>	Execute read queries	Always enabled
<code>write_neo4j_cypher</code>	Execute write queries	<b>Disabled by default</b>

## Tool 1: Get Schema

```
get_neo4j_schema
```

Returns the structure of your knowledge graph:

- **Node labels** (e.g., Company, RiskFactor, Filing)
- **Relationship types** (e.g., HAS\_RISK, FILED)
- **Properties** on nodes and relationships

**Why it matters:** The AI needs to understand your data model before it can write correct Cypher queries.

## Tool 2: Read Cypher

```
read_neo4j_cypher
```

Executes **read-only** Cypher queries against the database.

**Example flow:**

User: "What risks does Apple face?"

AI uses `read_neo4j_cypher:`

```
MATCH (c:Company {name: 'Apple Inc'})-[:HAS_RISK]->(r:RiskFactor)  
RETURN r.description
```

Result: [List of risk factors from SEC filings]

**Safe by design:** Cannot modify any data.

## Tool 3: Write Cypher

```
write_neo4j_cypher
```

Executes queries that **create, update, or delete** data.

**Important:** This tool is **disabled by default** for safety.

To enable, you must explicitly configure:

```
{
  "neo4j_write_enabled": true
}
```

## In This Lab

We'll use the Neo4j MCP Server with:

- **get\_neo4j\_schema** - To help the agent understand our SEC filings graph
- **read\_neo4j\_cypher** - To answer questions about companies and risks

**Write access is not needed** - we're analyzing existing data.

## Summary

- MCP is an open standard for AI tool integration
- Works like **USB for AI** - one protocol, many tools
- **Neo4j MCP Server** provides three tools for graph access
- **Secure-by-default** design prevents accidental data modification
- **Microsoft Foundry** supports MCP out of the box

Next: Learn about Microsoft Foundry and deploy your agent.

# **Microsoft Foundry**

**Microsoft's AI App and Agent Factory**

# What is Microsoft Foundry?

Microsoft describes it as:

"An interoperable AI platform that enables developers to build faster and smarter, while organizations gain fleetwide security and governance in a unified portal."

**Key concept:** An "AI app and agent factory" for the enterprise.

# Key Foundry Components

Component	Purpose
Foundry Models	Access to 11,000+ AI models
Model Router	Auto-selects best model per task
Foundry Agent Service	Build autonomous, context-aware agents
Foundry IQ	Next-gen RAG for knowledge grounding
MCP Tool Catalogue	Unified tool discovery and management

# Agents in Foundry

Foundry agents combine:

- **Instructions** - What the agent should do
- **Model** - Which LLM powers reasoning (or Model Router)
- **Tools** - MCP servers and built-in capabilities

## Finance Agent

Instructions: "You are a financial analyst assistant specializing in SEC 10-K filings analysis..."

Model: gpt-4o-mini

Tools: Neo4j MCP Server

# Enterprise Governance

**Foundry Control Plane** provides:

Capability	Description
Observability	Monitor all agents fleet-wide
Compliance	Enforce policies across deployments
Security	Microsoft Defender + Entra ID integration
Content Safety	Built-in threat detection

**Enterprise-ready from day one.**

# What We'll Build

A Finance Agent that:

Capability	How
Understands SEC filings	Knowledge graph context
Queries company data	Neo4j MCP Server
Answers risk questions	Graph traversal
Analyzes relationships	Connected data

**User:** "What risks does Apple face?"

**Agent:** Queries Neo4j, returns risk factors

# The Lab Flow

- 1. Create Foundry resource and project**
- 2. Deploy gpt-4o-mini model**
- 3. Create finance-agent**
- 4. Add Neo4j MCP tool**
- 5. Test with SEC filing questions**
- 6. Publish your agent**

# The GenAI Promise and Its Limits

## What Generative AI Does Well

LLMs excel at tasks that rely on pattern recognition and language fluency:

- **Text generation:** Creating human-like responses, summaries, explanations
- **Language understanding:** Parsing intent, extracting meaning, following instructions
- **Pattern completion:** Continuing sequences, filling in blanks, generating variations
- **Translation and transformation:** Converting between formats, styles, languages

These capabilities emerge from training on vast amounts of text data.

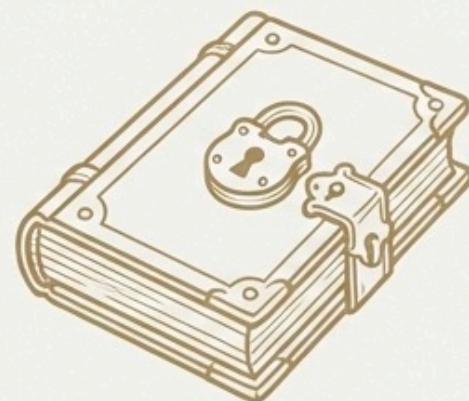
# The Three Gaps in the GenAI Promise

LLMs excel at text generation, language understanding, and pattern completion. However, production systems must address three fundamental limitations.



## Hallucination (Confident but Wrong)

LLMs generate statistically *probable* text, not factually *verified* text. When they don't know, they fabricate with full confidence.



## Knowledge Cutoff (Your Data is Invisible)

Models are trained on public data with a cutoff date. They have no access to your private company documents, databases, or real-time events.



## Relationship Blindness (Can't Connect the Dots)

LLMs process text sequentially and struggle to answer questions that require reasoning over relationships across multiple documents or sources.

# 1. Hallucination: Confident But Wrong

LLMs generate responses based on statistical likelihood, not factual verification.

## The Problem:

- Produces the most *probable* continuation, not the most *accurate*
- Doesn't say "I don't know"—generates plausible-sounding text instead
- Complete with fabricated details and citations

**Real Example:** In 2023, US lawyers were sanctioned for submitting an LLM-generated brief with six fictitious case citations.

## 2. Knowledge Cutoff: No Access to Your Data

LLMs are trained at a specific point in time on publicly available data.

**They don't know:**

- Recent events after their training cutoff
- Your company's documents, databases, or internal knowledge
- Real-time data: current prices, live statistics, changing conditions

**The Risk:** Ask about your Q3 results or last week's board meeting, and the LLM may still generate a confident (and wrong) response.

### **3. Relationship Blindness: Can't Connect the Dots**

LLMs process text sequentially and treat each piece in isolation.

**Questions they struggle with:**

- "Which asset managers own companies facing cybersecurity risks?"
- "What products are mentioned by companies that share risk factors?"
- "How are these two companies connected through their executives?"

These questions require *reasoning over relationships*—connecting entities across documents and traversing chains of connections.

# Why These Limitations Matter

Limitation	Impact	Example
Hallucination	Can't trust answers without verification	Legal brief with fabricated citations
Knowledge cutoff	Can't answer questions about your data	"What did our CEO say last quarter?"
Relationship blindness	Can't reason across connected information	"Which investors are exposed to these risks?"

Building production AI systems means addressing these limitations directly.

## The Solution: Providing Context

All three limitations have a common solution—**providing context**.

When you give an LLM relevant information in its prompt:

- It has facts to work with (reduces hallucination)
- It can access your specific data (overcomes knowledge cutoff)
- You can structure that information to show relationships (enables reasoning)

This is the foundation of **Retrieval-Augmented Generation (RAG)**.

# Summary

In this lesson, you learned about the fundamental limitations of LLMs:

- **Hallucination:** LLMs generate probable responses, not verified facts
- **Knowledge cutoff:** LLMs can't access recent events or your private data
- **Relationship blindness:** LLMs struggle with cross-document reasoning

The solution is providing context—which leads us to RAG.

**Next:** Learn how traditional RAG works and why it has its own limitations.

# **Context and the Limits of Traditional RAG**

# The Power of Context

Providing context in prompts dramatically improves LLM responses.

**When you include relevant information, the model can:**

- Generate accurate summaries grounded in actual documents
- Answer questions about your specific data
- Reduce hallucination by having facts to reference

This insight leads to **Retrieval-Augmented Generation (RAG)**.

# How Traditional RAG Works

Traditional RAG follows a simple pattern:

- 1. Index documents:** Break documents into chunks and create embeddings
- 2. Receive query:** User asks a question
- 3. Retrieve context:** Find chunks with embeddings similar to the query
- 4. Generate response:** Pass retrieved chunks to LLM as context

This works well for many use cases. But it has fundamental limitations.

# The Problem with Traditional RAG

Traditional RAG treats documents as isolated, unstructured blobs.

**What traditional RAG sees:**

Chunk 1: "Apple Inc. faces cybersecurity risks including..."

Chunk 2: "BlackRock Inc. holds shares in technology companies..."

Chunk 3: "The semiconductor supply chain impacts..."

**What traditional RAG misses:**

- Which specific companies does BlackRock own?
- Do any of those companies face cybersecurity risks?
- How are supply chain issues connected to specific products?

## **Retrieves Similar Content, Not Connected Information**

Traditional RAG can find text about cybersecurity and text about BlackRock.

**But it can't tell you:**

- Which asset managers are exposed to cybersecurity risks through their holdings

**Why?** Each chunk is independent—there's no understanding of how information connects.

# Questions Traditional RAG Can't Answer

Question	Why Traditional RAG Struggles
"Which asset managers own companies facing cyber risks?"	Requires connecting ownership data to risk mentions
"What products are mentioned by companies that share risk factors?"	Requires finding shared entities across documents
"How many companies mention supply chain issues?"	Requires aggregation, not similarity search
"What executives work for companies in the tech sector?"	Requires traversing entity relationships

These questions need *structured context* that preserves relationships.

# From Unstructured to Structured

**The core insight:** Information isn't truly unstructured.

Documents contain:

- **Entities:** Companies, people, products, risks
- **Relationships:** Owns, faces, mentions, works for

Traditional RAG ignores this structure. It treats a document as a bag of words to embed and search.

# The GraphRAG Solution

GraphRAG extracts structure, creating a *knowledge graph* that preserves:

- **Entities:** The things mentioned in documents
- **Relationships:** How those things connect
- **Properties:** Attributes and details about entities

**Traditional RAG asks:** "What chunks are similar to this query?"

**GraphRAG asks:** "What entities and relationships are relevant to this query?"

# Three Retrieval Patterns

GraphRAG enables three retrieval patterns:

Pattern	What It Does
<b>Vector search</b>	Find semantically similar content (what traditional RAG does)
<b>Graph traversal</b>	Follow relationships from relevant content to connected entities
<b>Database queries</b>	Answer precise questions about entities and relationships

The combination is more powerful than any single approach.

# Summary

In this lesson, you learned:

- **Context improves LLM responses** by providing relevant information
- **Traditional RAG** retrieves similar chunks based on vector similarity
- **The limitation:** Traditional RAG treats documents as isolated blobs, missing relationships
- **Questions requiring relationships** can't be answered with similarity search alone
- **GraphRAG** extracts structure from documents, preserving entities and relationships

**Next:** Learn how to transform documents into knowledge graphs.

# **From Documents to Knowledge Graphs**

## **with the Neo4j GraphRAG Package**

# The neo4j-graphrag Python Package

The official Neo4j GenAI package for Python provides a first-party library to integrate Neo4j with generative AI applications.

## Key benefits:

- Long-term support and fast feature deployment
- Reduces hallucinations through domain-specific context
- Combines knowledge graphs with LLMs for GraphRAG

# Supported Providers

## LLMs:

- OpenAI, Anthropic, Cohere, Google, MistralAI, Ollama

## Embeddings:

- OpenAI, sentence-transformers, provider-specific models

This flexibility lets you choose the models that best fit your requirements and budget.

# Building and Querying

The package provides tools for both constructing and querying knowledge graphs:

Category	Components
Construction	<code>SimpleKGPipeline</code> , <code>Pipeline</code> class
Retrieval	<code>VectorRetriever</code> , <code>Text2CypherRetriever</code> , hybrid methods
Orchestration	<code>GraphRAG</code> class for retrieval + generation

# SimpleKGPipeline

The key component for graph construction:

**What it does:**

1. Extracts text from documents (PDFs, text files)
2. Breaks text into manageable chunks
3. Uses an LLM to identify entities and relationships
4. Stores the structured data in Neo4j
5. Creates vector embeddings for semantic search

# The Transformation Process

Step	What Happens
Document Ingestion	Read source documents (PDFs)
Chunking	Break into smaller pieces for processing
Entity Extraction	LLM identifies companies, products, risks, metrics
Relationship Extraction	LLM finds connections between entities
Graph Storage	Save entities and relationships to Neo4j
Vector Embeddings	Generate embeddings for semantic search

# The SEC Filings Example

Throughout this workshop, you'll work with a knowledge graph built from SEC 10-K filings.

**These documents contain:**

- Companies and their business descriptions
- Risk factors they face
- Financial metrics they report
- Products and services they mention
- Executives who lead them

# From PDF to Graph

**In raw PDF form:** Information is locked in narrative text.

**In a knowledge graph:** It becomes structured and queryable:

```
(Apple Inc)-[:FACES_RISK]->(Cybersecurity Threats)  
(Apple Inc)-[:MENTIONS]->(iPhone)  
(BlackRock Inc)-[:OWNS]->(Apple Inc)
```

# The Complete Picture

After processing, your knowledge graph contains:

Documents → Chunks (with embeddings)



Entities (Company, Product, RiskFactor, Executive, FinancialMetric)



Relationships (FACES\_RISK, MENTIONS, OWNS, WORKS\_FOR, HAS\_METRIC)

This structure enables questions that traditional RAG can't answer.

## What the Graph Enables

Question Type	How the Graph Helps
"What risks does Apple face?"	Traverse FACES_RISK relationships
"Which companies mention AI?"	Find MENTIONS relationships to AI products
"Who owns Apple?"	Follow OWNS relationships from asset managers
"How many risk factors are there?"	Count RiskFactor nodes

## Quality Depends on Decisions

The quality of your knowledge graph depends on several key decisions:

- **Schema design:** What entities and relationships should you extract?
- **Chunking strategy:** How large should chunks be?
- **Entity resolution:** How do you handle the same entity mentioned differently?
- **Prompt engineering:** How do you guide the LLM to extract accurately?

The following lessons cover each of these decisions.

# Summary

In this lesson, you learned:

- **neo4j-graphrag** is the official package for building GraphRAG applications
- **SimpleKGPipeline** orchestrates the transformation from documents to graphs
- **The process:** Document → Chunks → Entity Extraction → Relationship Extraction → Graph Storage → Embeddings
- **Graph structure** enables queries that traverse relationships, not just find similar text

**Next:** Learn about schema design in SimpleKGPipeline.

# **Schema Design in SimpleKGPipeline**

# Why Schema Matters

Without a schema, extraction is unconstrained—the LLM extracts *everything*.

This creates graphs that are:

- **Non-specific:** Too many entity types with inconsistent labeling
- **Hard to query:** No predictable structure to write queries against
- **Noisy:** Irrelevant entities mixed with important ones

Providing a schema tells the LLM exactly what to look for.

## Schema in SimpleKGPipeline

SimpleKGPipeline accepts a `schema` parameter that guides extraction:

- **Node types:** What kinds of entities should be extracted
- **Relationship types:** What connections between entities matter
- **Patterns:** Which node-relationship-node combinations are valid

The pipeline uses the schema to guide extraction, prune invalid data, and ensure consistency.

## Three Schema Modes

Mode	Description	Best For
User-Provided	You define exactly what to extract	Production systems
Extracted	LLM discovers schema from documents	Exploration
Free	No constraints, extract everything	Initial discovery

# User-Provided Schema

```
schema = {  
    "node_types": [  
        {"label": "Company", "description": "A business organization"},  
        {"label": "RiskFactor", "description": "A business risk or threat"},  
        {"label": "Product", "description": "A product or service"},  
    ],  
    "relationship_types": [  
        {"label": "FACES_RISK", "description": "Company faces this risk"},  
        {"label": "MENTIONS", "description": "Company mentions this product"},  
    ],  
    "patterns": [  
        ("Company", "FACES_RISK", "RiskFactor"),  
        ("Company", "MENTIONS", "Product"),  
    ]  
}
```

# Automatic Schema Extraction

Let the LLM discover the schema from your documents:

```
pipeline = SimpleKGPipeline(  
    driver=driver,  
    llm=llm,  
    embedder=embedder,  
    schema="EXTRACTED", # or simply omit schema  
)
```

Useful when you don't know what entities exist in your documents.

## Free Mode

Extract everything without constraints:

```
pipeline = SimpleKGPipeline(  
    driver=driver,  
    llm=llm,  
    embedder=embedder,  
    schema="FREE",  
)
```

Most comprehensive extraction, but inconsistent structure.

# Defining Node Types

Node types can be simple strings or detailed dictionaries:

**Simple:**

```
node_types = ["Company", "Product", "RiskFactor"]
```

**With descriptions and properties:**

```
node_types = [
    {"label": "Company", "description": "A business organization",
     {
         "label": "Product",
         "properties": [{"name": "category", "type": "STRING"}]
     }
]
```

Descriptions help the LLM understand what each type means.

## Patterns: Valid Connections

Patterns specify which relationships are valid between node types:

```
patterns = [  
    ("Company", "FACES_RISK", "RiskFactor"),  
    ("Company", "MENTIONS", "Product"),  
    ("Executive", "WORKS_FOR", "Company"),  
]
```

Without patterns, the LLM might create nonsensical relationships like:

(Product)–[:FACES\_RISK]→(Company)

# Schema for This Workshop

Node Type	Description
Company	Organizations filing reports
RiskFactor	Business risks identified
Product	Products and services mentioned
Executive	Company leaders
FinancialMetric	Financial measures reported

## Relationships for This Workshop

Relationship	Pattern
FACES_RISK	Company → RiskFactor
MENTIONS	Company → Product
WORKS_FOR	Executive → Company
HAS_METRIC	Company → FinancialMetric

This focuses extraction on business-relevant information.

## When to Use Each Mode

Mode	Best For
User-Provided	Production systems with known query patterns
Extracted	Exploration when you're learning the domain
Free	Initial discovery of what's in your documents

For most production GraphRAG applications, a user-provided schema produces the most reliable results.

## Learn More

This lesson covers how to use schemas in SimpleKGPipeline.

For deep dives into schema design principles—including iterative refinement, domain modeling, and advanced patterns—see the dedicated **Neo4j GraphRAG Python course** on GraphAcademy.

# Summary

In this lesson, you learned:

- **Schema guides extraction:** Tells SimpleKGPipeline what to find
- **Three modes:** User-provided (control), Extracted (discovery), Free (exploration)
- **Node and relationship types:** Define what to extract with optional descriptions
- **Patterns:** Specify valid connections between node types
- **Production systems:** Benefit most from user-provided schemas

**Next:** Learn about chunking strategies and their trade-offs.

# Chunking Strategies

# Why Chunking Matters

LLMs have context limits. You can't pass an entire 200-page SEC filing to an LLM for entity extraction.

Documents must be broken into smaller pieces—**chunks**—that fit within processing limits.

**But chunking isn't just a technical necessity.** How you chunk documents affects both:

- Extraction quality
- Retrieval quality

# The Dual Impact of Chunk Size

Chunk size creates a fundamental trade-off:

## For Entity Extraction:

- Larger chunks provide more context for understanding entities
- The LLM sees more surrounding text, making better extraction decisions
- "The Company" can be resolved to "Apple Inc" when full context is visible

## For Retrieval:

- Smaller chunks enable more precise matches
- When searching, you want the most relevant *portion*, not a huge blob
- Less irrelevant content mixed with relevant content

# The Trade-off Illustrated

## Large Chunks (2000 chars)

[Full paragraph about Apple's risk factors, mentioning cybersecurity, supply chain, and regulatory risks with full context about each]

- Better entity extraction
- Less precise retrieval
- Returns more than needed

## Small Chunks (200 chars)

[Apple faces cybersecurity...]  
[Supply chain disruptions...]  
[Regulatory changes in...]

# Chunk Size Parameters

The `FixedSizeSplitter` has two key parameters:

`chunk_size` : Maximum number of characters per chunk

`chunk_overlap` : Characters shared between consecutive chunks

```
from neo4j_graphrag.experimental.components.text_splitters.fixed_size_splitter import FixedSizeSplitter  
splitter = FixedSizeSplitter(chunk_size=500, chunk_overlap=50)
```

Overlap ensures context isn't lost at chunk boundaries.

# Configuring the Pipeline

Pass a custom text splitter to SimpleKGPipeline:

```
from neo4j_graphrag.experimental.components.text_splitters.fixed_size_splitter import FixedSizeSplitter

# Create a splitter with your preferred settings
splitter = FixedSizeSplitter(chunk_size=500, chunk_overlap=50)

# Pass it to the pipeline
pipeline = SimpleKGPipeline(
    driver=driver,
    llm=llm,
    embedder=embedder,
    entities=entities,
    relations=relations,
    text_splitter=splitter, # Custom splitter
)
```

## Typical Chunk Sizes

Chunk Size	Best For
200-500 chars	High-precision retrieval, FAQ-style content
500-1000 chars	Balanced extraction and retrieval
1000-2000 chars	Context-heavy extraction, narrative documents
2000+ chars	Maximum context, fewer chunks

For SEC filings with complex, interconnected information, **500-1000 characters** often provides a good balance.

# Evaluating Chunk Quality

After chunking, evaluate the results:

```
// Check chunk count per document
MATCH (d:Document)<-[ :FROM_DOCUMENT ]-(c:Chunk)
RETURN d.path, count(c) AS chunkCount
ORDER BY chunkCount DESC

// Check chunk size distribution
MATCH (c:Chunk)
RETURN
    min(size(c.text)) AS minSize,
    max(size(c.text)) AS maxSize,
    avg(size(c.text)) AS avgSize
```

# What to Look For

## Good chunks:

- Reasonable count per document
- Consistent sizes
- Contain coherent, complete thoughts

## Signs of problems:

- Too few chunks → may be too large
- Highly variable sizes → inconsistent processing
- Incomplete sentences → overlap may be too small

## Experiment and Iterate

The best chunk size depends on:

- Document type and structure
- Query patterns you expect
- Balance between extraction and retrieval needs

**Start with a moderate size (500-800 characters), evaluate results, and adjust.**

# Summary

In this lesson, you learned:

- **Chunking** breaks documents into processable pieces
- **Chunk size affects both** entity extraction quality and retrieval precision
- **Larger chunks** = better context for extraction, less precise retrieval
- **Smaller chunks** = more precise retrieval, less context for extraction
- **Overlap** helps preserve context across chunk boundaries
- **Experimentation** is key to finding the right balance

**Next:** Learn about entity resolution—handling duplicate entities.

# Entity Resolution

# The Duplicate Entity Problem

When entities are extracted from text, the same real-world entity can appear with different names:

- "Neo4j" vs "Neo4j Graph Database" vs "Neo4j, Inc."
- "Apple" vs "Apple Inc" vs "Apple Inc." vs "the Company"
- "Tim Cook" vs "Timothy Cook" vs "CEO Tim Cook"

**Without resolution:** Your graph contains multiple nodes representing the same thing.

## Why This Breaks Queries

```
// This might miss risks if Apple appears under different names
MATCH (c:Company {name: 'Apple Inc'})-[:FACES_RISK]->(r:RiskFactor)
RETURN r.name
```

If some risks are connected to "Apple" and others to "APPLE INC", your query returns incomplete results.

**You can't trust basic queries like "How many risk factors does Apple face?"**

# Why Entity Resolution Matters

Entity resolution ensures:

- **Query accuracy:** One node per real-world entity
- **Relationship completeness:** All relationships connect to the canonical entity
- **Aggregation correctness:** Counts and summaries reflect reality

# Default Resolution in SimpleKGPipeline

By default, `SimpleKGPipeline` performs basic resolution:

- Entities with the **same label** and **identical name** are merged
- "Company: Apple Inc" + "Company: Apple Inc" = one node

But it misses variations:

- "Apple Inc" and "APPLE INC" (case difference)
- "Apple Inc" and "Apple Inc." (punctuation)
- "Apple" and "Apple Inc" (name variation)

# Resolution Trade-offs

## Too Aggressive

- "Apple Inc" (tech) merged with "Apple Records" (music)
- Distinct entities incorrectly combined
- Relationships become meaningless

## Too Conservative

- "Apple Inc" and "APPLE INC" remain separate
- Queries miss connections
- Aggregations are wrong

The right balance depends on your domain.

# Resolution Strategies

## Strategy 1: Upstream Normalization

Guide the LLM during extraction:

```
prompt_template = """  
When extracting company names, normalize to official names:  
- "Apple", "Apple Inc", "the Company" → "APPLE INC"  
- Use uppercase for company names  
- Use the full legal name when known  
"""
```

## Strategy 2: Reference Lists

Provide a canonical list of entities:

```
prompt_template = """  
Only extract companies from this approved list:  
- APPLE INC  
- MICROSOFT CORP  
- ALPHABET INC
```

Match variations to the canonical name.

"""

This works well when you know the entities in advance.

## Strategy 3: Post-Processing Resolvers

Apply resolvers after extraction:

```
from neo4j_graphrag.experimental.components.entity_resolvers import FuzzyMatchResolver

resolver = FuzzyMatchResolver(
    driver=driver,
    similarity_threshold=0.85, # How similar names must be to merge
)

# Run after pipeline completion
resolver.resolve()
```

Available Resolvers:

- **SpacySemanticMatchResolver**: Semantic similarity using spaCy
- **FuzzyMatchResolver**: String similarity using RapidFuzz

# Disabling Resolution

You can disable entity resolution entirely:

```
pipeline = SimpleKGPipeline(  
    driver=driver,  
    llm=llm,  
    embedder=embedder,  
    entities=entities,  
    relations=relations,  
    perform_entity_resolution=False, # No resolution  
)
```

Useful for debugging or applying custom resolution logic later.

# Validating Resolution

After resolution, verify your entity counts:

```
// Check for potential duplicates
MATCH (c:Company)
WITH c.name AS name, collect(c) AS nodes
WHERE size(nodes) > 1
RETURN name, size(nodes) AS duplicates

// Check company name variations
MATCH (c:Company)
WHERE c.name CONTAINS 'Apple' OR c.name CONTAINS 'APPLE'
RETURN c.name, count{(c)-[:FACES_RISK]->()} AS risks
```

# Summary

In this lesson, you learned:

- **Entity resolution** merges duplicate nodes representing the same real-world entity
- **Default resolution** catches exact matches only
- **Post-processing resolvers** catch variations using semantic or fuzzy matching
- **The trade-off:** Too aggressive merges distinct entities; too conservative keeps duplicates
- **Strategies include:** Upstream normalization, reference lists, post-processing resolvers

**Next:** Learn about vectors and semantic search.

# Vectors and Semantic Search

# What is a Vector?

Vectors are lists of numbers.

The vector [1, 2, 3] represents a point in three-dimensional space.

In machine learning, vectors can represent much more complex data—including the *meaning* of text.

# What are Embeddings?

Embeddings are numerical representations of text encoded as high-dimensional vectors (often 1,536 dimensions).

**The key property:** Similar meanings produce similar vectors.

- "Apple's business strategy" and "the company's strategic approach" → vectors close together
- "Apple's business strategy" and "banana nutrition facts" → vectors far apart

This enables **semantic search**—finding content by meaning, not just keywords.

# Why Vectors Matter for GraphRAG

Your knowledge graph now has:

- Structured entities (companies, risks, products)
- Relationships (FACES\_RISK, OWNS, MENTIONS)
- Text chunks from source documents

**But how do you *find* relevant information when a user asks a question?**

# Without Vectors vs With Vectors

## Without vectors:

- You need exact keyword matches
- "What challenges does Apple face?" won't find chunks about "risks" or "threats"

## With vectors:

- The question and chunks become embeddings
- You find chunks with similar *meaning*, regardless of exact words
- "Challenges" finds content about "risks" and "threats"

# Similarity Search

Vector similarity is typically measured by **cosine similarity**—the angle between two vectors:

Score	Meaning
Near 1.0	Very similar meanings
Near 0.5	Somewhat related
Near 0.0	Unrelated

When you search, your question becomes an embedding, and the system finds chunks with embeddings close to your question.

# Storing Vectors in Neo4j

When SimpleKGPipeline processes documents:

1. Each chunk gets an embedding from the embedding model
2. The embedding is stored as a property on the Chunk node
3. A vector index enables fast similarity search across all chunks

```
// Chunks have embedding properties
MATCH (c:Chunk)
RETURN c.text, size(c.embedding) AS embeddingDimensions
LIMIT 1
```

# Searching a Vector Index

```
// Create an embedding for the query
WITH genai.vector.encode(
    "What risks does Apple face?",
    "OpenAI",
    { token: $apiKey }
) AS queryEmbedding

// Search the vector index for similar chunks
CALL db.index.vector.queryNodes('chunkEmbeddings', 5, queryEmbedding)
YIELD node, score

RETURN node.text AS content, score
ORDER BY score DESC
```

This finds the 5 chunks most semantically similar to the query.

# Combining Vectors with Graph Traversal

The real power of GraphRAG: Start with semantic search, then traverse the graph.

```
WITH genai.vector.encode(  
    "What risks does Apple face?",  
    "OpenAI",  
    { token: $apiKey }  
) AS queryEmbedding  
  
CALL db.index.vector.queryNodes('chunkEmbeddings', 5, queryEmbedding)  
YIELD node, score  
  
// Traverse to connected entities  
MATCH (node)-[:FROM_CHUNK]-(entity)  
RETURN node.text AS content, score, collect(entity.name) AS relatedEntities
```

Returns both similar text AND the entities extracted from that text.

# The Complete Knowledge Graph

Your knowledge graph now has everything needed for GraphRAG:

Component	Purpose
Documents	Source provenance
Chunks	Searchable text units
Embeddings	Enable semantic search
Entities	Structured domain knowledge
Relationships	Connections between entities

# Three Retrieval Patterns

This structure enables three retrieval patterns:

1. **Vector search:** Find semantically similar content
2. **Vector + Graph:** Find similar content, then traverse to related entities
3. **Text2Cypher:** Query the graph structure directly

You'll learn these patterns in detail in Lab 5.

# Summary

In this lesson, you learned:

- **Vectors** are numerical representations of data
- **Embeddings** encode text meaning as high-dimensional vectors
- **Similar meanings** produce similar vectors, enabling semantic search
- **Neo4j stores vectors** alongside graph data with vector indexes
- **Semantic search** finds relevant chunks by meaning, not keywords
- **Vector + Graph** combines semantic search with relationship traversal

**Your knowledge graph is complete.** In Lab 5, you'll learn how to retrieve context using Vector, Vector Cypher, and Text2Cypher retrievers.

# **Neo4j GraphRAG Retrievers Overview**

# From Knowledge Graph to Answers

You have a knowledge graph with:

- **Entities:** Companies, products, risks, executives
- **Relationships:** OWNS, FACES\_RISK, MENTIONS, WORKS\_FOR
- **Embeddings:** Vector representations for semantic search

The question: How do you *retrieve* the right information to answer user questions?

# What is a Retriever?

A **retriever** searches your knowledge graph and returns relevant information.

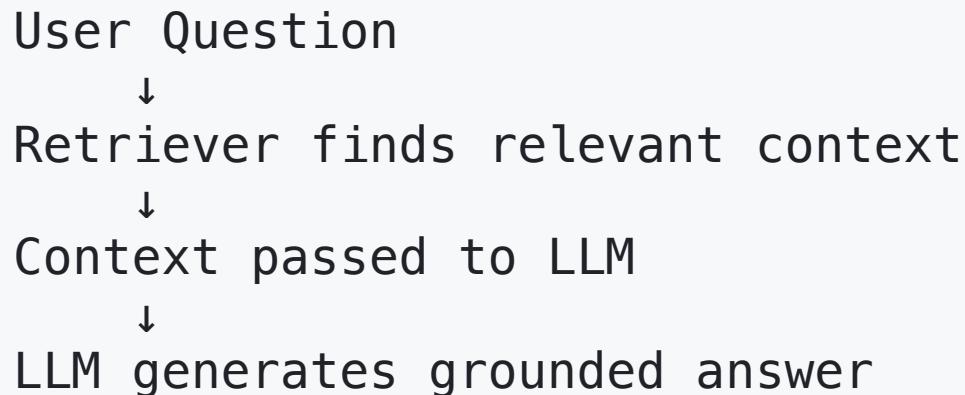
Three retrieval patterns:

Retriever	What It Does
Vector	Semantic similarity search across text chunks
Vector Cypher	Semantic search + graph traversal for relationships
Text2Cypher	Natural language → Cypher query for precise facts

Each pattern excels at different question types.

# The GraphRAG Class

Retrievers work with the **GraphRAG** class, which combines retrieval with LLM generation:



The retriever's job is finding the right context. The LLM's job is generating a coherent answer from that context.

# Vector Retriever

## How it works:

- Converts your question to an embedding
- Finds chunks with similar embeddings
- Returns semantically related content

## Best for:

- "What is Apple's strategy?"
- "Tell me about cybersecurity threats"
- Conceptual, exploratory questions

**Limitation:** Returns text chunks only—no entity relationships.

# Vector Cypher Retriever

## How it works:

- Vector search finds relevant chunks
- Custom Cypher query traverses from chunks to related entities
- Returns content + structured data

## Best for:

- "Which asset managers are affected by crypto regulations?"
- "What risks do tech companies face?"
- Questions needing both content and relationships

**Key insight:** The chunk is the anchor—you traverse from what vector search finds.

# Text2Cypher Retriever

## How it works:

- LLM converts natural language to Cypher
- Query executes against the graph
- Returns precise, structured results

## Best for:

- "How many risk factors does Apple face?"
- "List all companies owned by BlackRock"
- Counts, lists, specific lookups

**Limitation:** Question must map to graph schema.

# Choosing the Right Retriever

Question Pattern	Best Retriever
"What is...", "Tell me about..."	Vector
"Which [entities] are affected by..."	Vector Cypher
"How many...", "List all..."	Text2Cypher
Content about topics	Vector
Content + relationships	Vector Cypher
Facts, counts, aggregations	Text2Cypher

# The Decision Framework

Ask yourself:

## 1. Am I looking for content or facts?

- Content → Vector or Vector Cypher
- Facts → Text2Cypher

## 2. Do I need related entities?

- No → Vector
- Yes → Vector Cypher

## 3. Is this about relationships?

- Traversals → Vector Cypher or Text2Cypher
- Semantic → Vector

# Summary

In this lesson, you learned:

- **Retrievers** search and return relevant information from your knowledge graph
- **Vector Retriever**: Semantic similarity search across chunks
- **Vector Cypher Retriever**: Semantic search + graph traversal
- **Text2Cypher Retriever**: Natural language to precise database queries
- **Each excels at different question types**—choosing the right one matters

Next: Deep dive into each retriever type.

# Vector Retriever

# What is a Vector Retriever?

The **simplest retriever**—finds content by meaning, not keywords.

**How it works:**

1. Convert your question to an embedding
2. Search vector index for similar chunk embeddings
3. Return the most semantically similar chunks

**Key insight:** "Cybersecurity threats" finds content about "data breaches" and "hacking risks" even without exact word matches.

# Creating a Vector Retriever

```
from neo4j_graphrag.retrievers import VectorRetriever

vector_retriever = VectorRetriever(
    driver=driver,                      # Neo4j connection
    index_name='chunkEmbeddings',        # Vector index name
    embedder=embedder,                  # Embedding model
    return_properties=['text']          # Properties to return
)
```

## Components:

- **Driver:** Connection to Neo4j
- **Index:** Where embeddings are stored
- **Embedder:** Model that creates embeddings (e.g., OpenAI)

# Performing a Search

```
query = "What are the risks that Apple faces?"  
  
results = vector_retriever.search(  
    query_text=query,  
    top_k=5 # Return 5 most similar chunks  
)  
  
for record in results.records:  
    print(f"Score: {record['score']:.4f}")  
    print(f"Text: {record['text'][:200]}...")
```

Each result includes:

- **text**: The chunk content
- **score**: Similarity score (0-1, higher = more similar)

# Understanding Similarity Scores

Score Range	Interpretation
0.95-1.0	Extremely similar (near-exact match)
0.90-0.95	Highly relevant
0.85-0.90	Relevant
0.80-0.85	Moderately relevant
< 0.80	Weak relevance

Higher scores indicate stronger semantic matches.

# Best For

**Use Vector Retriever when:**

- Finding conceptually similar content
- Questions like "What is...", "Tell me about...", "Explain..."
- Exploratory questions about topics
- When exact keywords don't match but meaning does

**Example questions:**

- "What is Apple's business strategy?"
- "Describe cybersecurity threats"
- "What challenges do tech companies face?"

# Limitations

**Vector Retriever returns text only:**

- No entity relationships
- No structured data from the graph
- Can't aggregate across entities
- Can't traverse connections

**Example limitation:**

- Question: "What risks does Apple face?"
- Returns: Chunks about risks (may not be Apple-specific)
- Missing: Structured FACES\_RISK relationships

**When you need more:** Use Vector Cypher Retriever.

# The top\_k Parameter

Controls how many results to return:

top_k	Trade-off
1-3	Fastest, most relevant only
5-10	Balanced coverage
15-20	Maximum coverage, may include less relevant

**Rule of thumb:** Start with 5, adjust based on results.

# Summary

Vector Retriever is your foundation for semantic search:

- **Converts queries to embeddings** for meaning-based search
- **Returns semantically similar chunks** regardless of keywords
- **Best for content questions, topic exploration**
- **Limitation:** No graph relationships or structured data

**Next:** Learn how Vector Cypher Retriever adds graph intelligence.

# Vector Cypher Retriever

# Beyond Basic Vector Search

**Vector Retriever:** Returns text chunks only.

**Vector Cypher Retriever:** Returns text chunks + related entities from graph traversal.

Query: "What risks affect companies?"



Vector Search: Find relevant chunks



Graph Traversal: From chunks → Companies → RiskFactors → AssetManagers



Result: Content + structured entity data

# How It Works

**Two-step process:**

## 1. Vector Search (semantic)

- Find chunks similar to your question
- Same as Vector Retriever

## 2. Cypher Traversal (structural)

- From each chunk, traverse the graph
- Gather related entities and relationships
- Return enriched context

**The combination:** Semantic relevance + graph intelligence.

# Creating a Vector Cypher Retriever

```
from neo4j_graphrag.retrievers import VectorCypherRetriever

retrieval_query = """
MATCH (node)-[:FROM_DOCUMENT]-(doc:Document)-[:FILED]-(company:Company)
OPTIONAL MATCH (company)-[:FACES_RISK]->(risk:RiskFactor)
WITH node, score, company, collect(risk.name)[0..20] AS risks
RETURN node.text AS text, score,
       {company: company.name, risks: risks} AS metadata
ORDER BY score DESC
"""

retriever = VectorCypherRetriever(
    driver=driver,
    index_name='chunkEmbeddings',
    embedder=embedder,
    retrieval_query=retrieval_query
)
```

# Understanding the Retrieval Query

The library provides automatically:

```
CALL db.index.vector.queryNodes($index_name, $top_k, $embedding)  
YIELD node, score  
-- Your query starts here with node and score --
```

Your retrieval\_query:

- Receives `node` (matched chunk) and `score` (similarity)
- Traverses from node to related entities
- Returns enriched results

## Query Breakdown

```
-- Traverse from chunk to company
MATCH (node)-[:FROM_DOCUMENT]-(doc:Document)-[:FILED]-(company:Company)

-- Get related risks (OPTIONAL so companies without risks still appear)
OPTIONAL MATCH (company)-[:FACES_RISK]-(risk:RiskFactor)

-- Aggregate risks, limit to 20
WITH node, score, company, collect(risk.name)[0..20] AS risks

-- Return chunk text + metadata
RETURN node.text AS text, score,
       {company: company.name, risks: risks} AS metadata
```

# Why OPTIONAL MATCH Matters

Without OPTIONAL MATCH:

```
MATCH (company)-[:FACES_RISK]->(risk)
```

Only returns companies that *have* risk factors.

With OPTIONAL MATCH:

```
OPTIONAL MATCH (company)-[:FACES_RISK]->(risk)
```

Returns *all* companies; risks list is empty if none exist.

Use OPTIONAL MATCH for complete results.

## **Best For**

**Use Vector Cypher Retriever when:**

- You need content AND related entities
- Questions involve relationships
- You want to traverse from relevant content to connected data

**Example questions:**

- "Which asset managers are affected by cryptocurrency policies?"
- "What products do companies mention alongside AI?"
- "What risks connect to companies in the tech sector?"

# The Chunk as Anchor

**Critical concept:** You can only traverse from what vector search finds.

**Example problem:**

- Query: "What risks does Apple face?"
- Vector search finds: Chunks about "risk management" (not Apple-specific)
- Traversal: Goes to companies mentioned in those chunks
- Result: May not include Apple!

**Solution:** Ensure your question surfaces relevant chunks, or use Text2Cypher for entity-specific queries.

# When to Use Vector Cypher vs Text2Cypher

Question Type	Best Retriever
Content + related entities	Vector Cypher
Specific entity facts	Text2Cypher
"What does [topic] affect?"	Vector Cypher
"How many [entities]?"	Text2Cypher
Semantic + relationships	Vector Cypher
Precise counts/lists	Text2Cypher

# Summary

Vector Cypher Retriever combines semantic search with graph traversal:

- **Two-step process:** Vector search → Graph traversal
- **Custom Cypher query** defines what entities to gather
- **Returns:** Text chunks + structured metadata
- **Best for:** Questions needing content AND relationships
- **Key insight:** The chunk is the anchor for traversal

**Next:** Learn Text2Cypher for precise, entity-specific queries.

# Text2Cypher Retriever

# From Natural Language to Database Queries

**The problem:** Some questions need precise facts, not semantic search.

**Text2Cypher solution:**

1. User asks a question in natural language
2. LLM generates a Cypher query from the question
3. Query executes against the graph
4. Precise, structured results returned

**Example:**

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

```
MATCH (c:Company {name:'APPLE INC'})-[:FACES_RISK]->(r)  
RETURN count(r)
```
- Result: 45

# How It Works

User: "Which companies does BlackRock own?"



[LLM + Schema] → Generate Cypher



```
MATCH (am:AssetManager {managerName: 'BlackRock Inc.'})  
  -[:OWNS]→(c:Company)
```

RETURN c.name



[Execute Query]



Result: Apple Inc., Microsoft Corp., Alphabet Inc., ...

# Creating a Text2Cypher Retriever

```
from neo4j_graphrag.retrievers import Text2CypherRetriever
from neo4j_graphrag.schema import get_schema

# Schema tells LLM what's queryable
schema = get_schema(driver)

text2cypher_retriever = Text2CypherRetriever(
    driver=driver,
    llm=llm,                      # LLM for Cypher generation
    neo4j_schema=schema            # Graph structure
)
```

The schema is critical: Without it, the LLM guesses (often incorrectly).

# The Role of Schema

**Schema tells the LLM:**

Node properties:

```
Company {name: STRING, ticker: STRING}
```

```
RiskFactor {name: STRING}
```

```
AssetManager {managerName: STRING}
```

Relationships:

```
(:Company)-[:FACES_RISK]->(:RiskFactor)
```

```
(:AssetManager)-[:OWNS]->(:Company)
```

**With schema:** LLM knows exactly what entities and relationships exist.

**Without schema:** LLM invents non-existent properties and relationships.

# Best For

**Use Text2Cypher when:**

- You need precise facts, counts, or lists
- Question is about specific entities
- Aggregations are needed
- Direct graph queries (no semantic search)

**Example questions:**

- "How many risk factors does Apple face?"
- "List all companies owned by Vanguard"
- "Which company has the most products?"
- "What is the average number of risks per company?"

# Performing a Search

```
query = "What companies does BlackRock own?"  
  
results = text2cypher_retriever.search(query_text=query)  
  
# Results contain:  
# - The generated Cypher query  
# - The query results  
for record in results.records:  
    print(record)
```

**Behind the scenes:** LLM analyzes your question, generates Cypher, executes it.

# Limitations

**Text2Cypher requires questions that map to schema:**

- Question: "What's the sentiment about AI regulation?"
- Problem: No "sentiment" property in schema
- Result: Cannot generate valid query

**Text2Cypher may struggle with:**

- Ambiguous questions
- Questions requiring interpretation
- Content that lives in text chunks (use Vector instead)

# Security Considerations

Text2Cypher executes LLM-generated queries. Important safeguards:

- **Use read-only credentials:** Prevent accidental data modification
- **Validate queries:** Check for dangerous operations (DELETE, DROP)
- **Limit results:** Ensure LIMIT clauses prevent unbounded returns
- **Monitor usage:** Log generated queries for review
- **Trust boundaries:** Don't expose to untrusted users

# Generated Query Quality

LLMs may generate imperfect Cypher:

- Syntax errors
- Deprecated syntax
- Non-existent properties
- Inefficient patterns

Mitigation:

- Use custom prompts to guide Cypher generation
- Validate generated queries
- Handle errors gracefully

# Comparing All Three Retrievers

Question	Best Retriever	Why
"What is AI safety?"	Vector	Semantic content
"Which companies mention AI?"	Vector Cypher	Content + entities
"How many companies mention AI?"	Text2Cypher	Precise count
"Tell me about Apple"	Vector	Exploratory content
"List Apple's risks"	Text2Cypher	Specific entity facts

# Summary

Text2Cypher Retriever converts natural language to database queries:

- **LLM generates Cypher** from your question
- **Schema guides generation** for accuracy
- **Best for:** Facts, counts, lists, specific entities
- **Limitation:** Questions must map to graph schema

You now know all three retrieval patterns:

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

Next: Learn to build agents that choose the right retriever automatically.

# 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

<b>Problem</b>	<b>Solution</b>
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

Documents → Knowledge Graph → Retrievers → Agent → User

## ↓ Chunking Embeddings

# ↓ Schema Design      Entity Resolution

↓  
Vector      Text  
VectorCypher  
Text2Cypher

# Tool Selection ↓ Tool Selection pher ReAct Pattern er

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.