**Tab 1**

Here is the Python implementation for the **Recursive Multi-Agent System**.

This code upgrades the orchestrator to detect when a concept is too broad. Instead of just generating data nodes, it creates Agent nodes and :SPAWNS relationships 11, then recursively triggers those new agents to build their own sub-ontologies 2.

### 🧬 Recursive Agent Orchestrator

Python

import time  
from neo4j import GraphDatabase  
  
def call\_llm\_api(prompt):  
 """  
 Simulates LLM decision-making.  
 Ref: [cite: 333] - LLM API Integration Point.  
 """  
 print(f"🤖 LLM Thinking...")  
   
 # Scenario A: The Broad Concept (Root Level)  
 if "Roman Republic" in prompt and "Identify key entities" in prompt:  
 return """  
 // The concept is too broad, so we spawn agents [cite: 260]  
 CREATE (root:Agent {name: "Roman Republic", role: "Root", status: "Active"})  
 CREATE (s:Agent {name: "Senate Agent", role: "Institution Specialist", status: "Pending"})  
 CREATE (c:Agent {name: "Military Agent", role: "Military Specialist", status: "Pending"})  
 CREATE (root)-[:SPAWNS {reason: "Scope limitation"}]->(s)  
 CREATE (root)-[:SPAWNS {reason: "Scope limitation"}]->(c)  
 """  
   
 # Scenario B: The Specialized Context (Sub-Agent Level)  
 # Ref: [cite: 276-279] - Specialized ontology generation  
 elif "Senate Agent" in prompt:  
 return """  
 MATCH (a:Agent {name: "Senate Agent"})  
 SET a.status = "Processed"  
 CREATE (sen:Senate {name: "Roman Senate"})  
 CREATE (curia:Building {name: "Curia Julia"})  
 CREATE (a)-[:MANAGES\_ONTOLOGY]->(sen)  
 CREATE (sen)-[:MEETS\_IN]->(curia)  
 """  
   
 return "// No operation"  
  
class RecursiveOrchestrator:  
 def \_\_init\_\_(self, uri, user, password):  
 self.driver = GraphDatabase.driver(uri, auth=(user, password))  
  
 def close(self):  
 self.driver.close()  
  
 def get\_agent\_prompt(self, agent\_name, agent\_role):  
 """  
 Context-aware prompt.  
 Ref: [cite: 217-221] - General-Purpose Template adapted for Agents.  
 """  
 return f"""  
 You are '{agent\_name}', acting as a {agent\_role}.  
 Generate a specialized ontology for your domain using Cypher.  
 If your domain is still too broad, spawn new :Agent nodes with status 'Pending'.  
 Otherwise, create standard nodes and relationships.  
 """  
  
 def execute\_cypher(self, cypher):  
 with self.driver.session() as session:  
 try:  
 # Clean and run logic  
 clean\_cypher = cypher.replace("```cypher", "").replace("```", "")  
 session.run(clean\_cypher)  
 print("✅ Graph Update Executed.")  
 except Exception as e:  
 print(f"❌ Execution Error: {e}")  
  
 def run\_recursive\_cycle(self, initial\_concept):  
 """  
 The Core Loop: Spawns -> Executes -> Detects New Agents -> Repeats  
 Ref: [cite: 316] - Iterate and expand.  
 """  
 print(f"🚀 Initializing Root Concept: {initial\_concept}")  
   
 # Step 1: Root Execution  
 root\_prompt = f"Identify key entities for {initial\_concept}. If broad, spawn agents."  
 cypher = call\_llm\_api(root\_prompt)  
 self.execute\_cypher(cypher)  
  
 # Step 2: Recursive Processing Loop  
 while True:  
 # Find 'Pending' agents in the graph [cite: 307-308]  
 pending\_agents = self.find\_pending\_agents()  
   
 if not pending\_agents:  
 print("✨ All agents processed. Ontology construction complete.")  
 break  
  
 print(f"🔄 Found {len(pending\_agents)} pending agents. Spawning execution...")  
  
 for agent in pending\_agents:  
 name = agent["name"]  
 role = agent["role"]  
 print(f" 👉 Activating Sub-Agent: {name} ({role})")  
  
 # Generate sub-ontology  
 prompt = self.get\_agent\_prompt(name, role)  
 sub\_cypher = call\_llm\_api(prompt)  
 self.execute\_cypher(sub\_cypher)  
   
 # Mark as processed so we don't loop forever  
 self.mark\_agent\_processed(name)  
   
 time.sleep(1) # Rate limiting simulation  
  
 def find\_pending\_agents(self):  
 """  
 Queries the graph for agents that need to run.  
 """  
 query = "MATCH (a:Agent {status: 'Pending'}) RETURN a.name as name, a.role as role"  
 with self.driver.session() as session:  
 result = session.run(query)  
 return [{"name": r["name"], "role": r["role"]} for r in result]  
  
 def mark\_agent\_processed(self, agent\_name):  
 query = "MATCH (a:Agent {name: $name}) SET a.status = 'Processed'"  
 with self.driver.session() as session:  
 session.run(query, name=agent\_name)  
  
# --- usage\_example.py ---  
  
if \_\_name\_\_ == "\_\_main\_\_":  
 orchestrator = RecursiveOrchestrator("bolt://localhost:7687", "neo4j", "password")  
   
 # This single call triggers the cascade:  
 # Roman Republic -> Spawns Senate/Military -> Triggers Senate Logic -> Materializes Ontology  
 orchestrator.run\_recursive\_cycle("Roman Republic")  
   
 orchestrator.close()

### ⚙️ How This Implements the Architecture

1. The :SPAWNS Logic:  
   The LLM is instructed to create Agent nodes with status: 'Pending' if the scope is too broad. This directly implements the Delegation of Scope principle3333.
2. The Recursive Loop:  
   The run\_recursive\_cycle method doesn't just run once. It queries the database (find\_pending\_agents) to see if new agents were born in the previous turn. This creates the "fractal graph" effect4.
3. State Management:  
   By adding a status property (Pending/Processed) to the Agent nodes, the system tracks the lifecycle of the ontology generation, ensuring we don't re-run agents or infinite loop5.

### ⏩ Next Step

Now that you have the logic for **Spawning** and **Recursion**, the system creates a complex graph of Agents.

**Would you like me to generate the Cypher query to *visualize* this specific structure (showing the Root Agent, the :SPAWNS edges, and the sub-ontologies) so you can verify the hierarchy in Neo4j Bloom or the Browser?**

**Tab 2**

Here are the specific Cypher queries designed to visualize the "federation of reasoning units" you have built.

These queries allow you to audit the system in the Neo4j Browser or Bloom, distinguishing between the **"Control Plane"** (the Agents and their hierarchy) and the **"Data Plane"** (the actual knowledge graph they created).

### 1. The "Tree of Narrators" View (Agent Hierarchy)

This query reveals the "skeleton" of your system—the fractal structure where the Root Agent delegates to Sub-Agents 1. It filters out the detailed data to focus purely on the command structure.

Cypher

// Visualize the Agent Hierarchy and Delegation Scope  
MATCH path = (root:Agent {role: 'Root'})-[:SPAWNS\*]->(sub:Agent)  
RETURN root, sub, path,   
 relationships(path) AS delegation\_contracts

**What to look for:**

* **The Root Node:** The central hub ("Roman Republic").
* **The Spawning Chains:** Long chains indicate deep recursion (Level 1 -> Level 2 -> Level 3)2.
* **Edge Properties:** Inspect the :SPAWNS relationships to see the reason property (e.g., "Scope limitation"), which explains *why* the split happened3.

### 2. The "Creator & Creation" View (Full Provenance)

This query bridges the gap between the agents and the ontology. It shows which agent is responsible for which part of the graph, effectively visualizing the "recursive containment"4.

*Note: This assumes your Python script adds a relationship like [:MANAGES\_ONTOLOGY] or [:CREATED] between the Agent and the nodes it generates.*

Cypher

// See which Agent 'owns' which part of the Graph  
MATCH (a:Agent)  
OPTIONAL MATCH (a)-[:SPAWNS]->(child:Agent)  
OPTIONAL MATCH (a)-[:MANAGES\_ONTOLOGY]->(domain\_entity)  
RETURN a, child, domain\_entity

**What to look for:**

* **Clusters:** You should see distinct clusters where the SenateAgent connects only to Senator and decree nodes, while the MilitaryAgent connects to Legion nodes 5.
* **Isolation:** Ideally, sub-graphs are contained but linked via their parent agents.

### 3. The "Story Mode" View (Path Tracing)

To see how the narrative flows from the general to the specific, use this query. It traces a path from the broad concept down to a specific granular fact.

Cypher

// Trace the path from the Concept to the Granular Detail  
MATCH path = (root:Agent)-[:SPAWNS\*]->(leaf\_agent:Agent)-[:MANAGES\_ONTOLOGY]->(fact)  
WHERE root.role = 'Root'  
RETURN path  
LIMIT 50

### 🎨 Visualization Tip for Neo4j Bloom

To make this "visually narratable" 6 in your dashboard:

1. **Color Agent nodes Red:** This highlights the "active" reasoning units.
2. **Color Domain Nodes (Senate, Legion) Blue/Green:** This highlights the "passive" knowledge.
3. **Increase size of Root Agents:** Visually reinforces the hierarchy.

**Would you like to move on to the final piece of the architecture: the "Cross-Graph Linking" logic, which allows these separate agents (e.g., Military and Senate) to discover each other and form connections (like a Senate decree affecting a Legion)?**