# Dynamic Ontology Generator: LLM-Driven Knowledge Graph Architecture

## Executive Summary

This document describes a dynamic ontology generation system that combines Large Language Models (LLMs) with Neo4j graph databases to create adaptive, narrative-driven knowledge graphs. Rather than pre-defining schemas and relationships, the system enables LLMs to act as intelligent ontology generators that dynamically construct graph structures based on natural language inputs. The architecture supports recursive agent-based modeling where each node can function as an autonomous reasoning unit capable of spawning its own sub-ontologies.

## Introduction

Traditional knowledge representation systems require extensive upfront schema design and rigid taxonomies. This approach becomes problematic when dealing with diverse domains (Roman Republic, volcanic systems, automotive engineering) that each demand unique structural models. The Dynamic Ontology Generator addresses this limitation by treating LLMs as schema-free interpreters that can narratively construct graph ontologies on demand.

### Core Philosophy

The system embodies the "Chrystallum philosophy" - a paradigm where LLMs become persona-aware schema generators that dynamically scaffold metaphors and taxonomies into graph form. Users don't define ontologies; they narrate concepts, and the system operationalizes them into executable graph structures.

## System Architecture

### Component Overview

The architecture consists of three primary layers:

- **Interface Layer**: Natural language input processing
- **LLM Processing Layer**: Entity extraction, relationship inference, and Cypher generation
- **Graph Storage Layer**: Neo4j database for persistent graph structures

### Operational Flow

1. **User Input → Entity Extraction**: The LLM receives a concept (e.g., "Roman Republic") and identifies relevant entities
2. **Entity Typing → Node Creation**: Each entity becomes a labeled node with properties

3. **Relationship Inference**: The LLM determines logical connections based on domain knowledge
4. **Cypher Generation**: Creates executable Neo4j queries
5. **Graph Execution**: Queries run against the database to materialize the ontology

## Core Methodology

### How It Works

**Step 1: User Input → Entity Extraction**

When a user provides a concept like "Roman Republic," the LLM analyzes the domain and identifies key components:

- **Institutions**: Senate, Popular Assembly, Magistrates
- **Roles**: Senator, Consul, Tribune
- **Military**: Legion, Centurion, Cavalry
- **Events**: Punic Wars, Reforms, Civil Wars

**Step 2: Entity Typing → Node Creation**

Each identified entity becomes a Neo4j node with appropriate labels and properties:

```
CREATE (s:Senate {name: "Roman Senate"})
CREATE (c:Consul {name: "Consul"})
CREATE (l:Legion {name: "Legio X Equestris"})
```

**Step 3: Relationship Inference**

The LLM infers edges based on historical or logical context:

- Senate HAS_MEMBER Senator
- Consul COMMANDS Legion
- Citizen ELECTS Consul

Example Cypher implementation:

```
CREATE (sen:Senate {name: "Roman Senate"})
CREATE (senator:Senator {name: "Cicero"})
CREATE (sen)-[:HAS_MEMBER]->(senator)
CREATE (consul:Consul {name: "Julius Caesar"})
CREATE (legion:Legion {name: "Legio X"})
CREATE (consul)-[:COMMANDS]->(legion)
```

**Step 4: Dynamic Expansion**

The system supports iterative refinement. Users can request expansions like "Add more details about the Roman Republic," prompting the LLM to generate additional nodes (Tribunes, Patricians, Plebeians) and relationships (REPRESENTS, OPPOSES, ALLIES_WITH).

## Implementation Pattern

### General-Purpose Prompt Template

The following prompt scaffold enables reusable ontology generation across arbitrary domains:

You are a graph ontology generator.

Given a user concept, identify key entities, roles, institutions, objects, or processes that define it. For each entity:

1. Create a Neo4j node with a label and properties.
2. Infer relationships between entities and express them as Cypher edges.
3. Use descriptive relationship names (e.g., :HAS_MEMBER, :COMMANDS, :REPRESENTS).
4. Output only Cypher code.

Concept: [INSERT USER CONCEPT HERE]

### Example: Roman Republic

**Input:**
Concept: Roman Republic

**LLM Output:**
```
CREATE (r:Republic {name: "Roman Republic", era: "509 BC - 27 BC"})
CREATE (s:Senate {name: "Roman Senate"})
CREATE (c:Consul {name: "Consul"})
CREATE (senator:Senator {name: "Cicero"})
CREATE (l:Legion {name: "Legio X Equestris"})
CREATE (citizen:Citizen {role: "Plebeian"})
CREATE (r)-[:HAS_INSTITUTION]->(s)
CREATE (r)-[:HAS_OFFICE]->(c)
CREATE (s)-[:HAS_MEMBER]->(senator)
CREATE (c)-[:COMMANDS]->(l)
CREATE (citizen)-[:ELECTS]->(c)
```

### Example: Volcano

**Input:**
Concept: Volcano

**LLM Output:**
```
CREATE (v:Volcano {name: "Mount Etna", type: "Stratovolcano"})
CREATE (m:MagmaChamber {depth_km: 10})
CREATE (eruption:Eruption {style: "Explosive"})
CREATE (ash:VolcanicAsh {composition: "Silica-rich"})
CREATE (v)-[:HAS_MAGMA_CHAMBER]->(m)
CREATE (v)-[:HAS_ERUPTION]->(eruption)
CREATE (eruption)-[:PRODUCES]->(ash)
```

# Advanced Architecture: Multi-Agent Graph System

## Conceptual Foundation

The architecture extends beyond static graph generation to create a **multi-agent system** where each node represents an autonomous LLM agent with its own ontology-generation capacity. This transforms the graph from a storage mechanism into a federation of reasoning units.

## Core Principles

- **Node** = **Agent**: Each Neo4j node represents an LLM agent capable of recursive reasoning
- **Edges** = **Protocols**: Relationships function as communication channels and delegation contracts
- **Recursive Ontology Generation**: Each agent can spawn sub-agents to handle scoped complexity

## Agent-Based Example: Roman Republic

When the concept "Roman Republic" is deemed too broad, the root agent spawns specialized sub-agents:

- **Senate Agent** → Generates ontology of senators, debates, decrees
- **Consul Agent** → Generates ontology of consuls, elections, campaigns
- **Legion Agent** → Generates ontology of military units, generals, battles

```
CREATE (r:Agent {name:"Roman Republic", role:"Root Ontology"})
CREATE (s:Agent {name:"SenateAgent", role:"Institution"})
CREATE (c:Agent {name:"ConsulAgent", role:"Office"})
CREATE (l:Agent {name:"LegionAgent", role:"Military"})
CREATE (r)-[:SPAWNS]->(s)
CREATE (r)-[:SPAWNS]->(c)
CREATE (r)-[:SPAWNS]->(l)
```

## Recursive Ontology Generation

Each sub-agent independently generates its domain-specific subgraph:

**Senate Agent:**
```
CREATE (:Senate)-[:HAS_MEMBER]->(:Senator)
```

**Consul Agent:**
```
CREATE (:Consul)-[:COMMANDS]->(:Legion)
```

**Legion Agent:**
```
CREATE (:Legion)-[:PART_OF]->(:Army)
```

Subgraphs remain contained but expose interfaces to parent agents, enabling modular complexity management.

### Edge Semantics in Multi-Agent Systems

Edges transcend simple relationships to function as protocols:

| **Edge Type** | **Function** |
|---|---|
| :SPAWNS | Delegation of scope to specialized agent |
| :DELEGATES | Passing ontology generation tasks |
| :EXPOSES | Making subgraph elements visible to other agents |

Table 1: Edge semantics in agent-based architecture

### Emergent Hierarchical Structure

The system produces fractal graphs with recursive depth:

- **Level 1**: Root agent (e.g., Roman Republic)
- **Level 2**: Domain subagents (Senate, Consulship, Legions)
- **Level 3**: Entity sub-subagents (Cicero, Legio X, Plebeian Council)

Each level narrates its domain independently while maintaining compositional relationships with parent and child agents.

### Visual Metaphor: Tree of Narrators

The architecture resembles an arboreal structure:

- **Trunk (Root Agent)**: "I am the Roman Republic"
- **Branches (Subagents)**: "I am the Senate," "I am the Consulship," "I am the Legions"
- **Leaves (Sub-subagents)**: "I am Cicero," "I am Legio X," "I am the Plebeian Council"

Each node functions as both a data structure and a narrative voice within the knowledge graph ecosystem.

## System Benefits

### Key Advantages

1. **No Schema Upfront**: Eliminates need for pre-defined taxonomies across heterogeneous domains
2. **Narratable Graphs**: Knowledge structures emerge through storytelling rather than formal specification
3. **Cross-Domain Universality**: Seamlessly transitions between disparate concepts (Roman Republic → Photosynthesis → Quantum Mechanics)
4. **Scalability Through Recursion**: Complex concepts decompose into manageable sub-ontologies
5. **Pluralistic Modeling**: Multiple agents can model the same domain from different perspectives
6. **Dynamic Narratability**: The graph becomes a federation of narrators with distinct voices

### Scalability Considerations

The recursive agent architecture provides natural scaling boundaries:

- Single agents avoid scope overload by delegating to specialists
- Containment prevents ontological explosion
- Emergent hierarchies maintain tractable complexity at each level

## Usage Patterns

### Basic Implementation Steps

1. **Prompt the LLM**: "Given the concept 'Roman Republic,' generate Cypher that creates nodes and relationships for its key institutions, people, and military"
2. **Execute Cypher in Neo4j**: Run generated queries directly against the database
3. **Iterate**: Feed new prompts to expand or refine the graph structure

### Practical Workflow

**For any arbitrary concept:**

1. Submit concept to LLM using the prompt template
2. Review generated Cypher code
3. Execute queries in Neo4j
4. Request expansions or refinements as needed
5. Allow recursive agent spawning for complex domains

### Example Session

User: "Define the Roman Republic"
System: [Generates base ontology with Senate, Consul, Legion nodes]

User: "Expand with more details about the Senate"
System: [SenateAgent spawns, generates Senator nodes, faction relationships]

User: "Add military campaign information"
System: [LegionAgent expands with Campaign, General, Battle nodes]

## Technical Considerations

### Integration Points

- **LLM API**: Requires access to a language model with strong domain knowledge (GPT-4, Claude, etc.)
- **Neo4j Database**: Graph database instance with Cypher query execution capability
- **Orchestration Layer**: Middleware to manage LLM prompting and query execution

### Quality Control

- **Validation**: Generated Cypher should be syntactically verified before execution
- **Conflict Resolution**: Multiple agents may generate overlapping ontologies requiring merge strategies
- **Version Control**: Graph evolution should be tracked for rollback capability

### Performance Optimization

- **Batch Processing**: Group related entity creations in single transactions
- **Index Management**: Create appropriate indexes on frequently queried node properties
- **Agent Lifecycle**: Implement spawning/despawning strategies to manage computational overhead

## Future Extensions

### Potential Enhancements

1. **Multi-Persona Generation**: Allow different personas (historian, scientist, engineer) to generate alternative graph structures for the same concept
2. **Cross-Graph Linking**: Enable agents to discover and create relationships across independently generated ontologies
3. **Temporal Modeling**: Add time-based evolution tracking for historical or process-oriented concepts
4. **Conflict Detection**: Implement reasoning systems to identify contradictory relationships across subgraphs
5. **Explanatory Interfaces**: Generate natural language explanations of graph structure and agent reasoning

## Conclusion

The Dynamic Ontology Generator represents a paradigm shift in knowledge representation, moving from rigid schema-first approaches to narrative-driven, emergent graph structures. By treating LLMs as intelligent ontology generators and enabling recursive agent-based modeling, the system provides a scalable, adaptable framework for representing complex knowledge across arbitrary domains.

The architecture transforms knowledge graphs from static storage systems into living, reasoning ecosystems where each node functions as an autonomous narrator capable of dynamically constructing its own semantic universe. This approach aligns with the Chrystallum philosophy of persona-aware, narratable knowledge systems that operationalize human understanding through computational graph structures.

## Appendix: Code Reference

## Complete Roman Republic Example

```
// Root concept
CREATE (r:Republic {name: "Roman Republic", era: "509 BC - 27 BC"})

// Institutions
CREATE (s:Senate {name: "Roman Senate"})
CREATE (pa:Assembly {name: "Popular Assembly"})

// Offices
CREATE (c:Consul {name: "Consul", term_years: 1})
CREATE (t:Tribune {name: "Tribune", power: "Veto"})

// People
CREATE (senator:Senator {name: "Cicero", faction: "Optimate"})
CREATE (citizen:Citizen {role: "Plebeian"})

// Military
CREATE (l:Legion {name: "Legio X Equestris", strength: 5000})
CREATE (general:General {name: "Julius Caesar"})

// Relationships - Institutional
CREATE (r)-[:HAS_INSTITUTION]->(s)
CREATE (r)-[:HAS_INSTITUTION]->(pa)
CREATE (r)-[:HAS_OFFICE]->(c)
CREATE (r)-[:HAS_OFFICE]->(t)

// Relationships - Membership
CREATE (s)-[:HAS_MEMBER]->(senator)
CREATE (pa)-[:HAS_MEMBER]->(citizen)

// Relationships - Command
CREATE (c)-[:COMMANDS]->(l)
CREATE (general)-[:LEADS]->(l)

// Relationships - Political
CREATE (citizen)-[:ELECTS]->(c)
CREATE (t)-[:PROTECTS]->(citizen)
```

## Agent-Based Implementation Example

```
// Create root agent
CREATE (root:Agent {
name: "RomanRepublicAgent",
role: "Root Ontology Generator",
scope: "Full domain"
})

// Spawn domain-specific subagents
CREATE (senate_agent:Agent {
name: "SenateAgent",
role: "Institution Specialist",
scope: "Legislative branch"
})

CREATE (military_agent:Agent {
name: "MilitaryAgent",
role: "Military Specialist",
scope: "Armed forces"
})

CREATE (political_agent:Agent {
name: "PoliticalAgent",
role: "Political Process Specialist",
scope: "Elections and governance"
})

// Create delegation relationships
CREATE (root)-[:SPAWNS {reason: "Scope limitation"}]->(senate_agent)
CREATE (root)-[:SPAWNS {reason: "Scope limitation"}]->(military_agent)
CREATE (root)-[:SPAWNS {reason: "Scope limitation"}]->(political_agent)

// Each subagent generates its own ontology
// (Executed as separate LLM calls with scoped prompts)
```