Agentic Architecture: A Graph-Theoretic Framework

Robert Cunningham

January 2025

Introduction

This paper presents a mathematical framework for modeling interactions between users, AI agents, and tools as directed acyclic graphs (DAGs). We formalize how agent interactions are structured, stored, and analyzed through a graph-theoretic lens, where nodes contain input-output pairs and edges represent invocation relationships. This approach simplifies the representation while maintaining complete information about the interaction history.

Core Definitions

Definition 1 (Agent System)**.** *Let A be an **agent** with system prompt s_A and toolset $T_A = \{T_1, T_2, \ldots, T_n\}$.*

Definition 2 (Interaction Cycle)**.** *A **cycle** c_i represents a complete user-agent interaction, initiated by user input u_i and concluded with agent response v_i. We denote:*

$$c_i = (u_i, v_i)$$

Definition 3 (Chat History)**.** *The **chat history** after k cycles is:*

$$h_k = [c_i]_{i=1}^k = [(u_1, v_1), (u_2, v_2), \ldots, (u_k, v_k)]$$

Derivative Chat Histories

When an agent A receives input u_1 and determines tool T_i should be called, it constructs query $q_1^{(i)}$ yielding:

$$r_1^{(i)} = T_i(q_1^{(i)})$$

The agent interprets this result via its internal chat method chat^*:

$$v_1 = \text{chat}^*(r_1^{(i)})$$

Definition 4 (Derivative History)**.** *The **first derivative** of chat history includes intermediate tool interactions:*

$$h' = [c'_i] \text{ where } c'_i = (u_i, q_i^{(j)}, r_i^{(j)}, v_i)$$

*The **second derivative** h'' captures the full depth of all nested interactions, including agent-to-agent invocations through tools.*

Graph-Theoretic Representation

Node-Based Model

We represent each interaction as a node containing an input-output pair:

Definition 5 (Interaction Node)**.** *A node N in the interaction graph is defined as:*

$$N = (e_{in}, e_{out}, \lambda)$$

where e_{in} is the input, e_{out} is the output, and λ is a label identifying the processing entity (agent or tool).

Cycle as DAG

Each cycle c_i forms a directed acyclic graph $G_i = (V_i, E_i)$ where:

- V_i is the set of interaction nodes
- $E_i \subseteq V_i \times V_i$ represents invocation relationships
- The root node contains (u_i, v_i) labeled with agent A
- Child nodes contain tool or sub-agent interactions

Proposition 1 (DAG Property)**.** *For any cycle c_i, the graph G_i is acyclic, i.e., there exists no sequence of edges forming a directed cycle.*

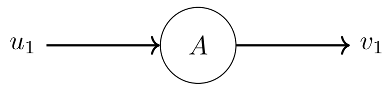
Proof. By construction, edges represent invocation relationships with strict temporal ordering. A tool cannot invoke its calling agent within the same execution context, ensuring acyclicity. □

Formal Graph Examples

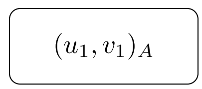
Consider the following cycle types expressed as graphs:

Simple Input-Output (Type 1)

A simple cycle with no tool calls, just user input and agent response:



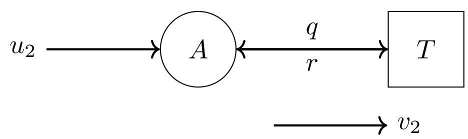
Node representation: Single node containing the pair



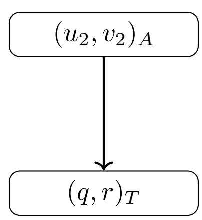
$$G_1 = (\{N_A\}, \emptyset) \text{ where } N_A = (u_1, v_1, A)$$

Single Tool Call (Type 2)

Agent receives input, calls a tool, and responds:



Node representation: Parent-child relationship

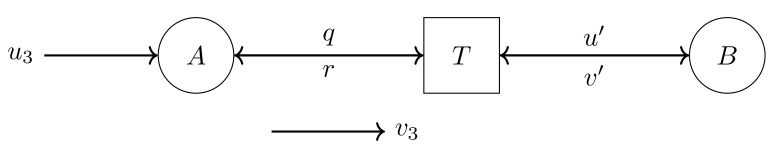


$$G_2 = (\{N_A, N_T\}, \{(N_A, N_T)\})$$

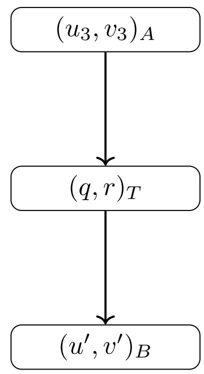
where $N_A = (u_2, v_2, A)$ and $N_T = (q_2, r_2, T)$

Nested Agent Invocation (Type 3)

Tool T invokes another agent B during execution:



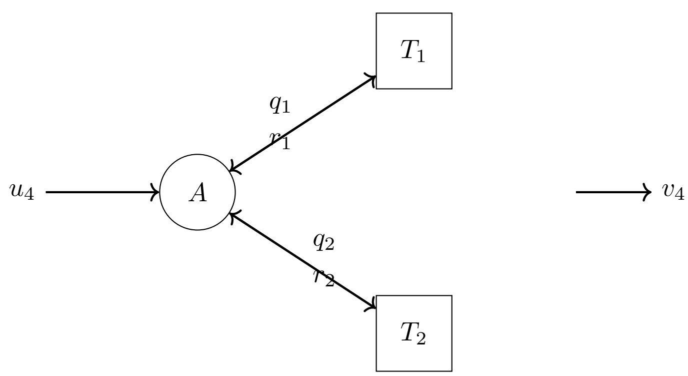
Node representation: Chain of invocations



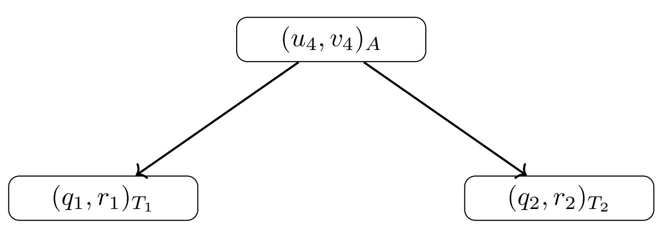
$$G_3 = (\{N_A, N_T, N_B\}, \{(N_A, N_T), (N_T, N_B)\})$$

Multiple Tool Calls (Type 4)

Agent calls two different tools before responding:



Node representation: Multiple children



$$G_4 = (\{N_A, N_{T_1}, N_{T_2}\}, \{(N_A, N_{T_1}), (N_A, N_{T_2})\})$$

Matrix Representations

Tool Selection Matrix

For a cycle with m tool calls across n available tools, we define the selection matrix:

$$S \in \{0,1\}^{m \times n}$$

where $S_{ij} = 1$ if the i-th call invokes tool T_j, and 0 otherwise.

Lemma 1 (Row Sum Property)**.** *Each row of S sums to exactly 1:*

$$\sum_{j=1}^{n} S_{ij} = 1 \quad \forall i \in \{1, \ldots, m\}$$

Access Control Matrix

In multi-agent systems, we formalize access control through a block matrix $\mathcal{A}$:

$$\mathcal{A} = \begin{bmatrix} M & R \end{bmatrix}$$

where:

- $M \in \{0,1\}^{|A| \times |A|}$ is the agent-to-agent adjacency matrix
- $R \in \{0,1\}^{|A| \times |T|}$ is the agent-to-tool access matrix
- A is the set of agents, T is the set of tools

Definition 6 (Agent Dispatch System)**.** *A dispatch system is a tuple* $(A, T, T_0, A_0, \mathcal{A})$ *where:*

- $A = \{a_1, \ldots, a_n\}$ *is the agent set*
- $T = \{t_0, t_1, \ldots, t_m\}$ *is the tool set*
- $T_0 = \{t_0\} \subseteq T$ *is the dispatch tool*
- $A_0 = \{a_1\} \subseteq A$ *is the user-facing agent*
- $\mathcal{A}$ *encodes access relations*

The dispatch tool t_0 mediates inter-agent communication: $M_{ij} = 1 \iff$ agent a_i can invoke agent a_j via t_0.

Reachability and Path Generation

Definition 7 (Reachability)**.** *Agent a_j is reachable from agent a_i if $(M^k)_{ij} > 0$ for some $k \geq 1$, where M^k denotes the k-th power of matrix M.*

The set of all reachable agents from a_i is:

$$\mathcal{R}(a_i) = \{a_j : \exists k \geq 1, (M^k)_{ij} > 0\}$$

Theorem 1 (Transitive Closure)**.** *The transitive closure $M^* = \sum_{k=1}^{n-1} M^k$ encodes all reachability relations, where $(M^*)_{ij} > 0$ iff $a_j \in \mathcal{R}(a_i)$.*

Loop Prevention Through Nilpotency

Definition 8 (Nilpotent Matrix)**.** *A matrix M is nilpotent if $\exists k \in \mathbb{N}$ such that $M^k = 0$. The smallest such k is the nilpotency index.*

Theorem 2 (Loop-Free Characterization)**.** *An agent dispatch system is loop-free if and only if its agent-to-agent adjacency matrix M is nilpotent.*

Proof. ($\Rightarrow$) Suppose the system is loop-free. Then the directed graph $G = (A, E)$ where $E = \{(a_i, a_j) : M_{ij} = 1\}$ is acyclic. For an acyclic graph with n vertices, any path has length at most $n-1$. Thus $(M^n)_{ij}$ counts paths of length n from a_i to a_j, which must be zero for all i, j. Hence $M^n = 0$.

($\Leftarrow$) Suppose $M^k = 0$ for some k. Then no paths of length k or greater exist in the dispatch graph. If a cycle existed, we could traverse it repeatedly to create arbitrarily long paths, contradicting $M^k = 0$. Thus the system is loop-free. $\square$

[Trace Test for Acyclicity] A dispatch system has no self-loops of length k if and only if $\mathrm{tr}(M^k) = 0$.

Proof. The trace $\mathrm{tr}(M^k) = \sum_{i=1}^{n} (M^k)_{ii}$ counts closed walks of length k. These exist iff there are cycles in the graph. $\square$

Proposition 2 (Maximum Invocation Depth)**.** *For a loop-free dispatch system with nilpotency index ν, the maximum invocation chain length is $\nu - 1$.*

Example: Five-Agent System

Consider the access control matrix:

$$\mathcal{A} = \begin{array}{c} A \\ B \\ C \\ D \\ E \end{array} \left[\begin{array}{ccccc|ccccccc} 0 & 0 & 0 & 1 & 1 & 1 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 1 & 0 & 0 \end{array}\right]$$

The agent-to-agent submatrix is:

$$M = \begin{pmatrix} 0 & 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \end{pmatrix}$$

Computing powers:

$$M^2 = \begin{pmatrix} 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \end{pmatrix}, \quad M^3 = \begin{pmatrix} 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \end{pmatrix}, \quad M^4 = 0$$

Since $M^4 = 0$ with $\mathrm{tr}(M^k) = 0$ for all k, the system is provably loop-free with maximum invocation depth 3.

Database Schema Formalization

Trace Record Structure

Each node in the persistent storage is represented as a tuple:

$$\tau = (\mathrm{id}, \mathrm{parent\_id}, \mathrm{cycle\_id}, \mathrm{call\_order}, \mathrm{fn}, e_{\mathrm{in}}, e_{\mathrm{out}}, \epsilon, \mathrm{pv}, \mathrm{av}, t_c, t_u)$$

where:

- id $\in \mathbb{N}$: unique identifier
- parent_id $\in \mathbb{N} \cup \{\mathrm{NULL}\}$: reference to parent node
- cycle_id $\in \mathbb{N}$: cycle grouping
- call_order $\in \mathbb{N}$: sibling ordering
- fn: symbolic function name
- $e_{\mathrm{in}}, e_{\mathrm{out}}$: JSON input/output
- ϵ: exception data (if any)
- pv: prompt versions
- av: application version
- t_c, t_u: creation and update timestamps

Tree Reconstruction

Given traces $\{\tau_i\}$ with the same cycle_id, we reconstruct the DAG:

Theorem 3 (Unique Reconstruction)**.** *The parent-child relationships encoded in parent_id fields uniquely determine a DAG structure for each cycle.*

Proof. The parent_id field creates a forest structure. Within each cycle_id group, exactly one node has parent_id = NULL (the root), and all other nodes have exactly one parent, forming a tree (which is a special case of a DAG). $\square$

Algebraic Properties

Composition of Cycles

The full interaction history over k cycles can be viewed as a forest:

$$\mathcal{F}_k = \bigcup_{i=1}^{k} G_i$$

where each G_i is a tree rooted at the user-agent interaction node.

Path Analysis

For any node N in cycle G_i, the depth $d(N)$ is the length of the unique path from the root to N:

$$d(N) = \begin{cases} 0 & \text{if } N \text{ is root} \\ 1 + d(\text{parent}(N)) & \text{otherwise} \end{cases}$$

Definition 9 (Interaction Depth)**.** *The depth of a cycle is:*

$$\Delta(G_i) = \max_{N \in V_i} d(N)$$

This represents the maximum nesting level of tool/agent invocations within the cycle.

Query and Result Arrays

For m total tool calls across all cycles, we maintain:

$$\mathbf{q} = [q_1, q_2, \ldots, q_m] \quad \text{and} \quad \mathbf{r} = [r_1, r_2, \ldots, r_m]$$

These can be indexed by a mapping function $\phi : (i, j) \mapsto l$ where (i, j) represents the j-th tool call in cycle i and l is the global index.

Tensor Representation

For advanced analysis, we construct a 3-tensor:

$$\mathcal{T} \in \mathbb{R}^{k \times \mu \times n}$$

where:

- k = number of cycles
- $\mu = \max_i |V_i|$ = maximum nodes in any cycle
- n = number of available tools

Element $\mathcal{T}_{ijl} = 1$ if the j-th node in cycle i invokes tool T_l, and 0 otherwise.

Proposition: Complete Reconstruction

Proposition 3. *The quadruple* $(h, \mathbf{q}, S, \mathbf{r})$ *where:*

- h *is the chat history*
- $\mathbf{q}$ *is the query array*
- S *is the selection matrix*
- $\mathbf{r}$ *is the result array*

is necessary and sufficient to reconstruct the complete interaction history including all tool invocations and their relationships.

Proof. **Necessity**: Each component captures essential information that cannot be derived from the others.

Sufficiency: Given $(h, \mathbf{q}, S, \mathbf{r})$:

1. h provides user inputs and agent outputs for each cycle
2. S identifies which tools were called and in what order
3. $\mathbf{q}$ provides the queries sent to each tool
4. $\mathbf{r}$ provides the results from each tool

The temporal ordering implicit in the arrays, combined with the selection matrix, allows complete reconstruction of the interaction DAG. $\square$

Advantages of the Graph Model

1. **Simplified Storage**: Each node stores a complete interaction pair, reducing edge complexity
2. **Natural Hierarchy**: Parent-child relationships directly model invocation patterns
3. **Efficient Queries**: Tree structure enables fast traversal and analysis
4. **Version Tracking**: Node-level version information supports evolution analysis
5. **Composability**: Cycles compose naturally as a forest of trees

Conclusion

This graph-theoretic framework provides a rigorous foundation for understanding and implementing agent-tool-user interaction systems. By representing interactions as DAGs with node-stored data pairs, we achieve both mathematical clarity and practical efficiency. The framework naturally supports persistence, analysis, and evolution of complex agent systems while maintaining the complete information needed for auditing and optimization.

The shift from edge-based to node-based storage of interaction data represents a key insight: by bundling input-output pairs within nodes, we transform a complex multi-graph into a simple tree structure, dramatically simplifying both theoretical analysis and practical implementation.