Coordination Without Orchestration

A Causal Event Model for Accountable Multi-Agent AI Systems

Miles Martin

rob@version2beta.com

Draft — November 3, 2026

Abstract

Multi-agent AI systems typically rely on a central orchestrator to delegate tasks, handle failures, and determine completion. This coupling concentrates fragility, obscures accountability, and produces coordination structures that cannot be analyzed before execution. We present a coordination model in which agents are typed transitions over event sets, execution history is an acyclic causal event tree (occurrence-net-inspired), and the system topology is fully analyzable from a declarative configuration before any agent fires. We show that graph-structural coordination properties—reachability, terminal consistency, subscriber completeness, orphaned-emitter detection, and structural deadlock detection—are statically decidable from the configuration alone, independently of agent behavior. We further show that failures, including those requiring human intervention, are first-class events in the causal graph, giving the system a uniform accountability structure across automated and manual resolution paths. We introduce an *ancestry-bounded firing* discipline that gives finite event histories even for cyclic subscription graphs, assuming one logical discharge per processing obligation, an aggregate child bound, and terminal exhaustion events. Eventual quiescence additionally requires eventual scheduling and finite processing or timeout. Retained event bytes, payloads, and recorded decisions permit reconstruction of the recorded history, not arbitrary re-execution of agents or external effects. We distinguish *tree correctness*, enforced by the runtime, from gate-relative *product correctness*, which does not establish artifact correctness. We describe a working implementation, EventedAI, and argue that the architecture generalizes to any system in which heterogeneous agents—human, automated, or AI—must coordinate over shared work.

1 Introduction

The dominant approach to multi-agent AI coordination is orchestration [24, 5]: a central agent or framework dispatches tasks to subordinate agents, collects results, and decides what happens next. This approach is pragmatic but carries structural costs. The orchestrator is a single point of failure. Failures in subordinate agents must surface back through the orchestrator, which may not have the context to handle them. The execution history may be implicit in the orchestrator's state; then accountability requires reconstructing that state. When topology is encoded in its logic rather than declared separately, checking possible handoffs before execution also requires analyzing that logic. Declaring the topology makes structural checks possible without claiming to predict what opaque agents will do.

We propose an alternative in which no agent has authority over any other. Agents communicate exclusively through typed events on a shared stream. Each agent declares the event types it will consume—its *subscription set*—and the event types it may produce—its *emission set*. The subscription graph defined by these declarations is the coordination structure. No orchestrator is needed because routing is implicit in the graph.

This framing has several consequences we develop in this paper.

1. **The execution history is a causal event tree**—an acyclic unfolding of the subscription graph into a causally ordered record of events. The tree is always acyclic by construction, even when the subscription graph contains cycles, because each event carries a unique identifier and its causal parent's identifier. In Petri net terms this is an *occurrence net* (§9); in distributed tracing terms it is a span tree.
2. **Coordination correctness is statically decidable.** From the team configuration alone—before any agent fires—we can decide reachability, terminal consistency, structural deadlock detection, and whether every non-terminal event type has a subscriber.
3. **Failures are first-class events.** A failed processing obligation produces a structured failure event linked to its trigger. The same subscription mechanism can route it to diagnosis, recovery, or a request for human input, rather than leaving continuation outside the declared workflow.
4. **Completion is derivable from finite quiescent prefixes of the causal event tree and its processing obligations.** A run prefix is team-complete when no team obligations remain pending or in flight and its frontier consists entirely of terminal event types.
5. **The architecture generalizes.** The model makes no assumptions about what agents are. They may be LLMs, humans, automated services, or any combination.

We describe EventedAI, a working implementation built on NATS JetStream, and demonstrate its application to a software development team in which LLM agents coordinate to produce working code from a project brief.

Contributions

- A formal model of multi-agent coordination with typed event interfaces, whose runs are represented as acyclic causal event trees with occurrence-net semantics for causation.
- Proofs that occurrence nets are acyclic and that coordination type-checking conditions are statically decidable in polynomial time.
- A characterization of two distinct failure modes—agent failure and suspension—and a uniform mechanism for handling both.
- An *ancestry-bounded firing* discipline giving finite histories under explicit obligation, branching, and terminal-exhaustion assumptions, even for cyclic subscription graphs (Theorem 8).
- Reconstruction of retained history and artifacts, including recorded model responses and operator answers, without re-running decisions or external effects.
- A separation of runtime-enforced tree correctness from gate-relative product correctness, which does not prove the artifact correct.
- A working implementation demonstrating the model applied to AI-driven software development.

2 Background

2.1 Petri Nets and Occurrence Nets

A Petri net [23] is a bipartite directed graph of places and transitions, with tokens marking places. A transition fires when all its input places are marked, consuming those tokens and producing tokens on its output places. A *colored Petri net* [13] extends the basic model by allowing tokens to carry typed data values. Transitions may be guarded by predicates over token colors and may produce tokens of different colors on different output places.

An *occurrence net* [19] is the unfolding of a net into a partial order of event occurrences. Where the original net may have cycles—modeling iteration or revision—the occurrence net records each individual firing as a distinct node. The occurrence net is always acyclic: it is a record of what happened, in causal order.

This structure is familiar under other names. In distributed tracing [20], the *span tree* of a request is structurally identical: each span references its parent span, the root span has no parent, and cycles in the underlying service topology unfold into a tree of distinct span instances. In event sourcing [6], the *event log* is the same object in serialized form—a totally ordered record whose partial-order structure is recovered by following causation pointers. We use "occurrence net" where the formal properties (acyclicity, reachability, unfolding) are being invoked, and "causal event tree" or "event history" in prose, where what matters is the intuitive shape rather than the theorem machinery.

2.2 Process Calculi

Milner's CCS [16] and Hoare's CSP [12] provide algebraic frameworks for reasoning about concurrent processes that communicate over channels. Both formalisms support analysis for deadlock, liveness, and behavioral equivalence. Our model is related but differs in that agents are opaque functions with declared interface types rather than algebraically defined processes, which forecloses some expressiveness but enables static analysis that the process calculi do not directly support.

2.3 Event Sourcing and Causal Ordering

Lamport [14] established the happens-before relation for distributed events: event A happens before event B if A causally precedes B. Event sourcing [6] applies this principle to application architecture: the event log is the source of truth, and current state is derived by replaying events in causal order. Distributed tracing systems (e.g., OpenTelemetry [20]) implement causal tracking through trace and span identifiers, forming an occurrence net over distributed operations.

2.4 The Saga Pattern

Garcia-Molina and Salem [7] introduced the saga as a long-running transaction decomposed into a sequence of local transactions, each with a compensating transaction for rollback. Our failure handling generalizes this: failure events route through the same subscription mechanism as normal work rather than through a separate compensation channel.

2.5 Choreographic Programming

Choreography [22] is the established term for coordination-by-subscription, in contrast to *orchestration* by a central conductor. Choreographic programming languages such as Chor [4] and its successors allow developers to write a *global choreography*—a single program describing how all parties interact—which is then mechanically *projected* to local behavior for each participant, with static guarantees that the projections are consistent, deadlock-free, and live. Montesi's survey [17] provides a comprehensive treatment.

Our model inverts this design. Rather than starting from a global choreography and deriving local behavior, we start from local declarations (each agent declares its own interface) and derive global topology from the subscription graph. This is necessary when agents are black boxes—LLMs, humans, external services—whose internal behavior cannot be specified at choreography-design time. The tradeoff is that we lose the end-to-end type safety guarantees that choreography projection provides; we recover partial structural guarantees through static analysis of the subscription graph.

2.6 Session Types

Multiparty session types [10] provide a type discipline for typed communicating processes in which a *global type* specifies the interaction protocol for all participants, and each participant is assigned a *local type* by projection. Type checking statically ensures deadlock freedom, progress, and protocol conformance. Session types are more expressive than our model: they support ordered message sequences, branching, and recursion with full static verification.

Our agents are not session-typed processes—they are nondeterministic functions with declared input and output *sets*, not ordered sequences. Session types require the protocol to be fully specified upfront; we do not require this. The contribution of this paper is not a more expressive type system but a coordination model suitable for agents whose behavior is open-ended and cannot be session-typed.

2.7 Blackboard Systems and Shared-Medium Coordination

Blackboard systems [18] (originating with the Hearsay-II speech-understanding system) coordinate autonomous agents through a shared data structure: agents read from and write to a blackboard without direct communication. Linda's tuple spaces [8] generalize this to generative communication: agents post typed tuples and retrieve them by pattern matching.

Our model is structurally similar: agents do not communicate directly but through a shared event stream. The key differences are (1) events are immutable and causally linked, providing an audit trail that blackboard writes and Linda tuples do not; (2) our typed subscription and emission sets enable static analysis; and (3) the event bus is ordered and durable, providing delivery guarantees absent from the original blackboard and Linda models.

The Reo coordination language [2] and BIP (Behavior-Interaction-Priority) [3] provide formal component-based coordination models with static verification. Both are closer in spirit to choreography languages and are discussed further in Section 9.

2.8 Multi-Agent LLM Systems

Recent work in LLM-based multi-agent systems includes delegation-based architectures [24], role-playing frameworks [11, 25], graph-structured flows [15], and simulated social agents [21]. These systems focus on task decomposition and emergent agent behavior but do not, to our knowledge, provide formal coordination models, static analysis of system topology, or first-class causal failure handling. Table 1 summarizes key distinctions.

System	Formal model	Static analysis	Causal audit	No orchestrator
AutoGen [24]	×	×	×	×
LangGraph [15]	partial	×	×	×
MetaGPT [11]	×	×	×	×
ChatDev [25]	×	×	×	×
EventedAI (this work)	✓	✓	✓	✓

Table 1: Comparison of LLM multi-agent systems. LangGraph provides graph-structured routing but no formal model, static type checking, or causal accountability.

3 Formal Model

3.1 Events and Event Vocabularies

Definition 1 (Event Vocabulary)**.** An event vocabulary $\mathcal{V}$ is a finite set of event type labels.

Definition 2 (Event)**.** An event e is an 8-tuple

$$(\mathit{id},\ \mathit{type},\ \mathit{project},\ \mathit{run},\ \mathit{causation},\ \mathit{correlation},\ \mathit{timestamp},\ \mathit{payload})$$

where:

- $\mathit{id} \in \mathcal{ID}$ is a globally unique identifier;
- $\mathit{type} \in \mathcal{V}$ is the event type;
- $\mathit{project}$, $\mathit{run} \in \mathcal{ID}$ identify the project and run;
- $\mathit{causation} \in \mathcal{ID} \cup \{\bot\}$ is the identifier of the causing event, or $\bot$ for the root event;
- $\mathit{correlation} \in \mathcal{ID}$ is the run identifier of the root event (invariant: $\mathit{correlation} = \mathit{run}$ for all events in a run);
- $\mathit{payload} \in \mathcal{H}$ is a content-addressed handle into a separate data store.

The payload field carries only a handle into a content-addressed store, not data itself. This decouples the event bus from payload size, allowing large artifacts—code, specifications, documents—to be referenced without transmission.

Definition 3 (Root Event)**.** A root event e_0 has $\mathit{causation} = \bot$. All other events in a run have $\mathit{causation} \neq \bot$.

We require the following runtime invariants of any conforming implementation:

Invariant 1 (Uniqueness). Event identifiers are globally unique: no two distinct events share the same id.

Invariant 2 (Causal precedence). For any non-root event e emitted during a run, $e.causation$ references an event that was already present in the run at the time e was created.

These are requirements on a conforming runtime, not consequences of interface declarations or a claim of complete prototype conformance. For attribution, each emitted event must also have durably recorded producer metadata, in its envelope or referenced payload, identifying the agent or runtime control rule and logical attempt that produced it. The history records triggering events and captured decisions; it need not expose every internal action or all data used by an opaque agent.

Definition 4 (Causation Relation)**.** The *causation relation* $\to_c$ on events in a run is defined by: $e \to_c e'$ if and only if $e'.causation = e.id$.

Definition 5 (Causal Predecessor)**.** Event e *causally precedes* event e', written $e \prec e'$, if $e \to_c e'$ (base case), or if there exists e'' such that $e \to_c e''$ and $e'' \prec e'$ (inductive case). That is, $\prec$ is the transitive closure of $\to_c$.

3.2 Agents

Definition 6 (Agent)**.** An agent A is a tuple (id, S, E, f) where:

- $id \in \mathcal{ID}$ is a unique identifier;
- $S \subseteq \mathcal{V}$ is the *subscription set*;
- $E \subseteq \mathcal{V}$ is the *emission set*;
- $f\colon \text{Events}(S) \to 2^{\text{Events}(E)}$ is a (possibly nondeterministic) transition function.

The transition function f is intentionally general. It may be a deterministic computation, an LLM invocation, or a human decision. The model makes no assumptions about f's internals beyond its typed interface: it consumes events of types in S and produces events of types in E.

Limitation: join patterns. The agent definition models single-event-triggered agents: each firing consumes one input event. Many practical workflows require *join patterns*, in which an agent waits for multiple correlated input events before firing (e.g., merging results from parallel tasks). The EventedAI prototype addresses this empirically via a `compose_handles` pragma: when an agent declares multiple input handles, the runtime resolves each from the content-addressed store and concatenates them into a single composed input before invoking the agent. This implements a restricted join over same-run outputs stored in Tether. The generalization to $f\colon 2^{\text{Events}(S)} \to 2^{\text{Events}(E)}$ with a formal join condition adds complexity to the static analysis; a full treatment is deferred to future work.

Definition 7 (Team Configuration)**.** A team configuration $T = (\mathcal{V}, \mathcal{A}, terminal)$ where $\mathcal{V}$ is the event vocabulary, $\mathcal{A}$ is a finite set of agents, and $terminal \subseteq \mathcal{V}$ is the user-declared set of *terminal* event types—the intended endpoints of the workflow. Terminal types may still be consumed by external systems or humans; they are terminal only with respect to the automated agents in T. This distinction is important for suspension (Section 6), in which an agent emits a terminal event that is subsequently consumed by a human outside the team's scope.

3.3 The Subscription Graph

Definition 8 (Subscription Graph)**.** Given team configuration T, the subscription graph $G_T = (\mathcal{V}, E_G)$ is a directed graph where there is an edge $(v, v') \in E_G$ if and only if there exists an agent $a \in \mathcal{A}$ such that $v \in a.S$ and $v' \in a.E$.

Definition 9 (Coordination Graph)**.** The *coordination graph* $H_T = (\mathcal{V} \cup \mathcal{A},\ E_H)$ is a bipartite directed graph where E_H contains an edge (v, a) if $v \in a.S$, and an edge (a, v') if $v' \in a.E$.

Definition 10 (Orphaned Emitter)**.** An agent $a \in \mathcal{A}$ has an *orphaned emission* if there exists $v \in a.E$ such that $v \notin \mathit{terminal}$ and no agent $a' \in \mathcal{A} \setminus \{a\}$ has $v \in a'.S$. The coordination graph H_T is used to detect orphaned emitters in $O(|\mathcal{A}| \cdot |\mathcal{V}|)$ time.

The subscription graph G_T is the projection of H_T onto event types and encodes agent-declared causal relationships between event types. An edge (v, v') in G_T means "an agent exists that, upon receiving an event of type v, may produce an event of type v'". Figure 1 shows a simplified subscription graph for a software development team.

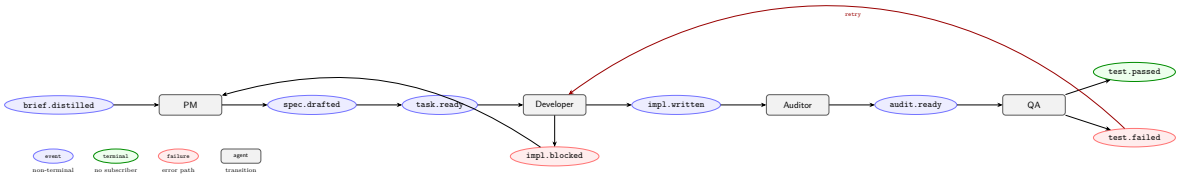


Figure 1: Subscription graph for a simplified software development team. Oval nodes are event types; rectangles are agents. The red arc from `test.failed` back to `Developer` is a cycle in the subscription graph—legitimate, as it models retry. Terminal events (green) have no subscribers.

3.4 Runs and Occurrence Nets

Definition 11 (Run)**.** A run $R(T, e_0)$ contains root event e_0 and all events produced by processing team obligations starting from it. Each event e creates one obligation (e, a) for each agent a with $e.\mathit{type} \in a.S$. An obligation is pending, in flight, or discharged. It has at most one logical discharge, committing a single output batch, possibly empty. Agent outputs e' satisfy $e'.\mathit{type} \in a.E$ and $e'.\mathit{causation} = e.\mathit{id}$. A declared runtime control rule may instead produce typed failure outputs with the same causal parent.

Operational model (for analysis claims). An execution prefix records its events and obligation states; we write P for its event set when considering only the graph. The scheduler may select a pending obligation. Before delivery it applies the runtime guards, including the ancestry guard when required. Guard rejection commits the specified control output and discharges the obligation. Otherwise processing commits agent outputs or runtime failure outputs. All committed outputs are fresh children of the triggering event, except that duplicate admission of the same committed output is a no-op. Transport redeliveries retry the same obligation and cannot reopen a discharged obligation or create additional committed

batches. A new logical retry represented by a fresh child event creates new obligations; it is not a repeated firing of the original pair.

Control rules declare their triggering types and output types in $\mathcal{V}$, including `processing.failed` and `cycle.exhausted`. Graph checks over agent declarations alone do not cover these extra routes; to check control paths, include the rules as additional typed transitions in the configuration being analyzed. No spontaneous external input is admitted inside a fixed run scope. Static graph results require no scheduling policy. Progress claims need additional scheduling and timeout assumptions.

Definition 12 (Occurrence Net of a Run)**.** The occurrence net $\mathcal{O}(R) = (R, \prec)$ is the set of events in R partially ordered by the causal predecessor relation $\prec$.

Definition 13 (Frontier)**.** The frontier $F(R) \subseteq R$ is the set of events in R with no causal successors: $F(R) = \{e \in R \mid \nexists\, e' \in R.\ e \prec e'\}$.

Definition 14 (Team-Complete Prefix)**.** A *team-complete prefix* is a finite execution prefix with event set $P \subseteq R$, no pending or in-flight team obligations, and a terminal frontier:

$$\forall e \in F(P).\ e.\mathit{type} \in \mathit{terminal}.$$

Definition 15 (Completed Run)**.** An execution with event set R is *complete (with respect to team T)* if it is a team-complete prefix.

4 Properties

4.1 Structural Properties of Occurrence Nets

Theorem 1 (Acyclicity)**.** *Given Invariants 1 and 2, the occurrence net $\mathcal{O}(R)$ of any run R is acyclic.*

Proof. Every direct causal edge points from an already present event to a fresh event, by Invariants 1 and 2. Thus each edge strictly advances creation order, regardless of the recorded wall-clock timestamp. A directed cycle would require creation order to advance strictly back to its starting event, which is impossible. The transitive closure therefore has no cycle. □

Figure 2 illustrates how a cyclic subscription graph unfolds into an acyclic occurrence net. Even though the net has a retry cycle, each instantiation of the cycle produces distinct events with distinct identifiers.

Corollary 2 (Causal Completeness)**.** *For any event e in a completed run R, the set $\{e' \in R \mid e' \prec e\}$ is finite and totally reconstructable from the event log by following* `causation` *pointers.*

4.2 Static Analysis of Team Configurations

The following properties of a team configuration T are decidable from T alone, before any run begins.

Definition 16 (Reachable Event Types)**.** Given T and root event type $v_0 \in \mathcal{V}$, the set of reachable types $\mathrm{Reach}(T, v_0)$ is the set of event types reachable from v_0 in G_T.



(a) Subscription graph (cyclic)

(b) Occurrence net (acyclic)

Figure 2: A cyclic subscription graph (left) unfolds into an acyclic occurrence net (right). The retry loop produces distinct event instances e_2 and e_4—both of type `impl.written`—with separate identifiers and causal links. No cycle can form in the occurrence net because each event's `causation` must point to an event that already exists.

Theorem 3 (Reachability is Decidable)**.** Reach(T, v_0) *is computable in* $O(|\mathcal{V}| + |E_G|)$ *time by breadth-first traversal of* G_T*.*

Definition 17 (Subscriber Completeness)**.** T is *subscriber-complete* if for every $v \in \mathcal{V} \setminus$ *terminal*, there exists $a \in \mathcal{A}$ with $v \in a.S$.

Theorem 4 (Subscriber Completeness is Decidable)**.** *Subscriber completeness of* T *is decidable in* $O(|\mathcal{V}| \cdot |\mathcal{A}|)$ *time.*

A configuration that fails subscriber completeness has event types that will be emitted but never consumed—silent dead ends in the occurrence net.

Definition 18 (Terminal Consistency)**.** T is *terminal-consistent* if for every $v \in \mathcal{V}$: $v \in$ *terminal* if and only if no agent in $\mathcal{A}$ has $v \in a.S$.

Theorem 5 (Terminal Consistency is Decidable)**.** *Terminal consistency of* T *is decidable in* $O(|\mathcal{V}| \cdot |\mathcal{A}|)$ *time.*

Terminal consistency is the bridge between the declared *terminal* set and the structural topology of G_T. When T is terminal-consistent, subscriber completeness is equivalent to the structural condition that every non-leaf type in G_T has a subscriber—making Definitions 16 and 17 non-tautological and the static analyzer actionable. Violations surface two classes of configuration bug:

- *Undeclared dead end*: a structural leaf not in *terminal*. The pipeline ends silently on a type the user did not intend as an endpoint. Either add a subscribing agent or declare the type terminal.
- *Spurious terminal*: a declared terminal type that has an internal subscriber. The declared endpoint is not structurally final. Either remove the subscription or redefine the terminal set.

Definition 19 (Structural Deadlock)**.** T has a *structural deadlock* if there exists a set $C \subseteq \mathcal{V}$ such that: all edges in G_T from C lead only within C; C contains no terminal event type; and C is reachable from the root.

Theorem 6 (Graph-Structural Deadlock is Decidable)**.** *Existence of structural deadlocks in* T *is decidable in* $O(|\mathcal{V}| + |E_G|)$ *time by identifying strongly connected components in* G_T *with no outgoing edges to terminal event types, using Tarjan's or Kosaraju's algorithm.*

Scope note. This characterizes *graph-structural* deadlock: cycles from which no terminal event type is reachable at the event-type level. Full deadlock analysis for colored Petri nets depends on token markings and transition guards, which our model does not specify (agents are black boxes). Absence of this structural condition does not imply runtime progress: an available exit need not be produced by an agent.

Definition 20 (Coordination Type Checking)**.** A team configuration T *passes coordination type checking* if: (0) T is terminal-consistent; (1) T is subscriber-complete; (2) T contains no structural deadlocks; (3) T has no orphaned emitters; (4) every terminal event type is reachable from the root; and (5) every agent is reachable from the root.

Theorem 7 (Coordination Type Checking is Decidable)**.** *Whether T passes Definition 20 is decidable in polynomial time in the size of T.*

4.3 Termination

Coordination type checking alone does not imply termination or successful completion.

Observation 1. If the graph of all possible agent and runtime-control emissions is acyclic, obligations discharge at most once, and every discharge has finite output, the event history is finite. Acyclicity bounds path depth; finite branching then bounds the number of events.

Observation 2. Cycles can produce unbounded causal chains of fresh events. Transport redelivery bounds do not prevent this: each logical retry may emit a fresh event with a new delivery counter.

Definition 21 (Ancestry-Bounded Firing)**.** Assume terminal consistency and reserve `cycle.exhausted` as a terminal type in $\mathcal{V}$ with no internal subscribers. Given finite limits $k : \mathcal{V} \setminus \mathit{terminal} \to \mathbb{N}_0$, processing an obligation (e, a) is admitted only if

$$\left|\{e' \in R \mid e' \prec e \ \wedge \ e'.\mathit{type} = e.\mathit{type}\}\right| < k(e.\mathit{type}).$$

The guard applies before agent processing or its runtime failure handling. On rejection, the runtime discharges the obligation by appending one fresh `cycle.exhausted` child of e, instead of delivering e. This terminal child creates no internal obligations and never triggers another exhaustion event. The rejected event itself already exists.

Theorem 8 (Bounded Termination)**.** *Let $\mathcal{V}$ and $\mathcal{A}$ be finite. Assume the operational model of Definition 11, terminal consistency, and ancestry-bounded firing. Assume a finite integer $B \geq 1$ bounds the total number of committed children of any event, aggregated over all its obligations. This bound includes fan-out, multiple outputs of the same type,* `multi_emit`*, logical retries, and runtime failure emissions. Set*

$$L = \sum_{v \in \mathcal{V} \setminus \text{terminal}} k(v), \qquad D = L + 1.$$

Every causal path has at most D edges, and the event history has at most $\sum_{j=0}^{D} B^j$ events. If, additionally, every pending obligation is eventually scheduled and every in-flight obligation discharges after finite processing or timeout, the run eventually becomes quiescent. This does not imply successful output or a terminal frontier.

Proof. Consider any finite causal path starting at the root. For each nonterminal type v, at most $k(v)$ occurrences on that path can be admitted for processing: a later occurrence has at least $k(v)$ same-type ancestors and is rejected. Hence at most L path nodes are admitted nonterminal events. Every nonfinal path node is either such an admitted event or a rejected event whose only possible children are terminal exhaustion events. Beyond the admitted nodes there can therefore be at most two nodes: a rejected event and its terminal child. A path ending at an unprocessed or other terminal event adds at most one node instead. It has at most $L + 2$ nodes, or $D = L + 1$ edges. A tree with this depth and at most B children per node has at most $\sum_{j=0}^{D} B^j$ nodes. There are therefore finitely many events and, since $\mathcal{A}$ is finite, finitely many obligations. Each discharges at most once. Under the additional scheduling and finite-discharge assumptions, all these obligations eventually discharge. The resulting frontier need not be terminal: an admitted firing may, for example, emit nothing. □

The aggregate bound B is an additional restriction, not a consequence of the number of declared output types. One output type can have many event instances. Finite history also does not bound elapsed time without the stated scheduling and timeout assumptions.

Under this theorem `cycle.exhausted` ends the team's work on that branch. An external operator or another team may handle it, but any resumed scope needs its own bounds. An internal automated exhaustion handler is outside this theorem and requires a separately bounded fallback discipline; recursively replacing exhaustion by exhaustion is not permitted here.

Limits $k(v)$ can be chosen according to the cost of a firing and the intended depth of revision cycles. Unlike per-message delivery limits, they are evaluated against causal ancestry, so emitting a fresh retry event does not reset the budget.

5 Correctness

5.1 Two Notions of Correctness

Definition 22 (Tree Correctness)**.** A run is *tree-correct* if every actual processing of an event by an agent respects that agent's subscription set; every committed agent output respects its producer's emission set; every runtime control output is permitted by a declared control rule for its trigger; and every non-root output names its already present triggering event as causal parent. Producer metadata is retained as required in Section 3.1. Terminal events need not be processed.

Claim 1. If the runtime enforces these routing, emission, control-rule, attribution, and causal-admission checks, every admitted run prefix is tree-correct. This is an enforcement property, not a consequence of declaring types.

Definition 23 (Quality Gate)**.** A *quality gate* is an agent $q \in \mathcal{A}$ with emission set $E_q = \{pass, fail\}$, where $pass \in terminal$ and *fail* routes back to a worker agent.

Definition 24 (Product Correctness)**.** A completed run R is *product-correct* with respect to gates $Q \subseteq \mathcal{A}$ if R is tree-correct and, for every $q \in Q$ that fired, the set of frontier events attributed to q is nonempty and contains only *pass* events.

This is gate-relative success, not proof that the artifact meets its requirements or that every artifact received adequate checks. Gate coverage and the quality of each check remain separate obligations.

Corollary 9 (Coordination Prerequisite)**.** *A run that is not tree-correct is not product-correct under this definition, irrespective of whether a gate emitted pass.*

Corollary 10 (Quality Gate Refinement)**.** *If a retained pass event references an artifact later found defective, its recorded producer metadata identifies the gate that passed that artifact. Together with the retained inputs and result, this provides evidence for examining and improving the check; it does not establish that the gate caused the defect.*

5.2 The Correctness Boundary

The separation of tree correctness from product correctness maps onto the separation between coordination and content. Figure 3 illustrates the two layers.

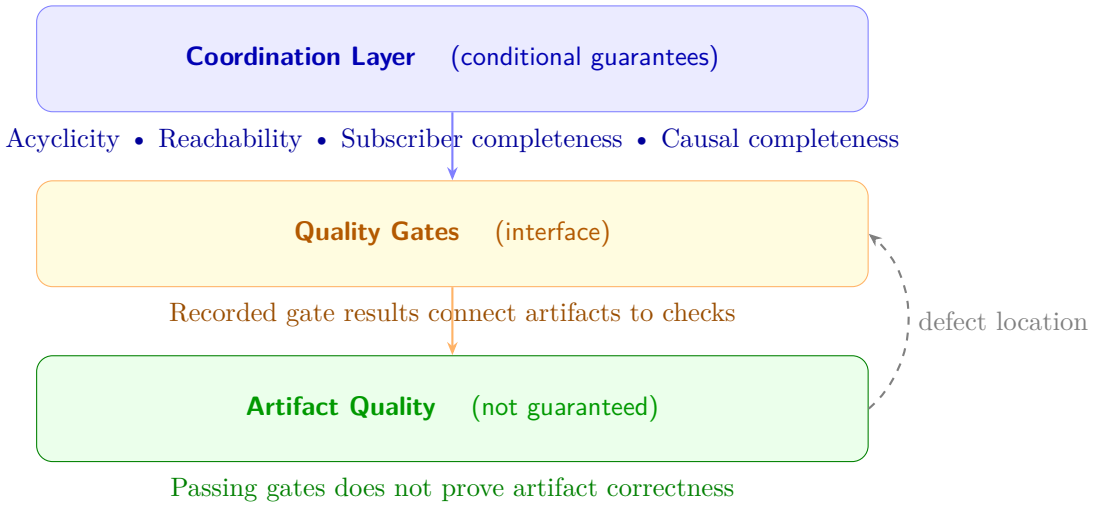


Figure 3: The two-layer correctness boundary. Graph properties are checked from declarations; valid execution additionally requires runtime enforcement. Gate-relative success does not prove artifact correctness. Recorded producer and artifact references identify the check to examine when an artifact that passed is later found defective.

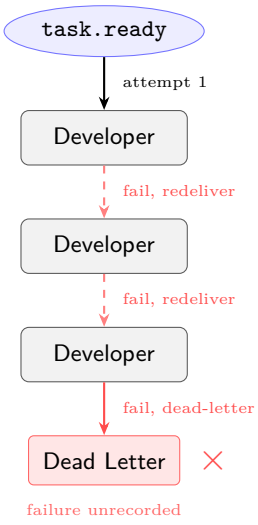
6 Failure Handling

6.1 Two Failure Modes

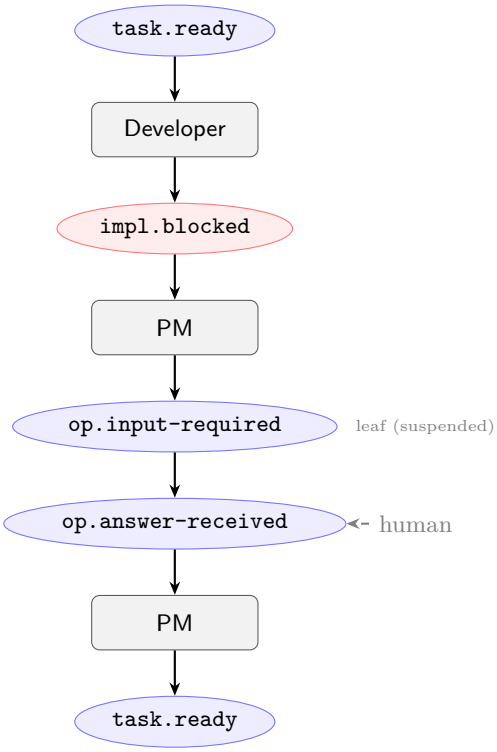
Definition 25 (Agent Failure)**.** An *agent failure* occurs when processing an obligation fails to produce an accepted result before its finite attempt/timeout policy is exhausted. Delivery claims do not remove the durable event from history; transport retries remain attempts to discharge the same obligation.

Definition 26 (Suspension)**.** A *suspension* occurs when a team reaches a quiescent prefix whose frontier contains one or more events intended for external handling (typically human input). Relative to the team, these frontier events are terminal; external actors may later extend the global event history with causally linked events outside this fixed team scope. A resumed automated scope needs its own processing and ancestry bounds.

If a failed delivery is placed in a dead-letter queue without recording the failure and its handling in the causal history, continuation requires work outside the declared workflow. Dead-letter queues can preserve causal identifiers; the issue is not the queue itself but whether failure and recovery participate in the same recorded routing system. Figure 4 contrasts an unrecorded dead-letter path with a causally linked suspension.



(a) Unrecorded failure: handling leaves the causal event history



(b) Suspension: chain intact, branch waits for human input

Figure 4: Two handling paths. On the left, dead-letter handling is not recorded in the causal history. On the right, the question, answer, and renewed work remain linked: the branch pauses for external input without losing the reason for continuation.

6.2 Failure as First-Class Events

Definition 27 (`processing.failed`)**.** When an admitted processing obligation exhausts its attempt policy, a declared runtime rule discharges it by appending a fresh `processing.failed` child of the triggering event. Its retained payload records the original event, attempt errors, and agent identity. The type belongs to $\mathcal{V}$ and is either handled

by subscribed agents or explicitly terminal. An ancestry-guard rejection instead uses the terminal `cycle.exhausted` rule of Definition 21.

Theorem 11 (Local Causal Preservation under Failure)**.** *Let P be a finite tree-correct prefix satisfying Invariants 1 and 2, and let $e \in P$. Suppose the runtime durably appends a fresh event x of declared type* `processing.failed`*, authorized for the failed obligation on e, with $x.causation = e.id$ and retained producer metadata and failure payload. Then $P \cup \{x\}$ remains tree-correct and acyclic; e has a recorded successor; and the ancestry of x is reconstructable from the retained log.*

Proof. All existing events and processing records are unchanged. The new event's type is authorized by the control rule, its identity is fresh, and its parent is already present. Thus its admission preserves tree correctness and the two invariants. Theorem 1 gives acyclicity. The edge $e \to_c x$ records the failure's trigger, and following parent identifiers in the finite prefix recovers its ancestry. □

This is a local preservation result, not a progress theorem. A subscriber can turn the failure into diagnosis, retry work, or a question to a human. Appending the event alone does not ensure any of those happens: handling, scheduling, and durable infrastructure are also required. Under Theorem 8, such recovery outputs count toward B and remain subject to the ancestry guard.

6.3 The Operator Loop

The operator loop handles suspensions requiring human input. Figure 5 shows the full sequence.

7 Implementation

7.1 EventedAI

EventedAI is a Go prototype of the coordination approach using NATS JetStream as the event bus and SQLite as the content-addressed store (Tether). The system implements typed routing, causal linking, content-addressed storage, delivery limits, and EVENT: declaration. In the current prototype, EVENT declarations are parsed from a constrained first-line convention in model output and validated against each agent's declared emission set.

Each agent is a durable JetStream consumer subscribed to its declared subscription subjects. Durable consumers with `MaxAckPending=1` serialize message processing per agent, eliminating the need for additional locking. `MaxDeliver=3` and `AckWait=180s` bound the retry window before escalation.

7.2 Agent Library

Agents are configured by composing a base type prompt with zero or more skill prompts from a library, performed at startup before any event is processed. Agent types implemented include: Product Manager, Developer, Designer, Writer, QA Engineer, and Release Engineer. Each type defines a system prompt specifying domain, constraints, and output format including EVENT: declarations.

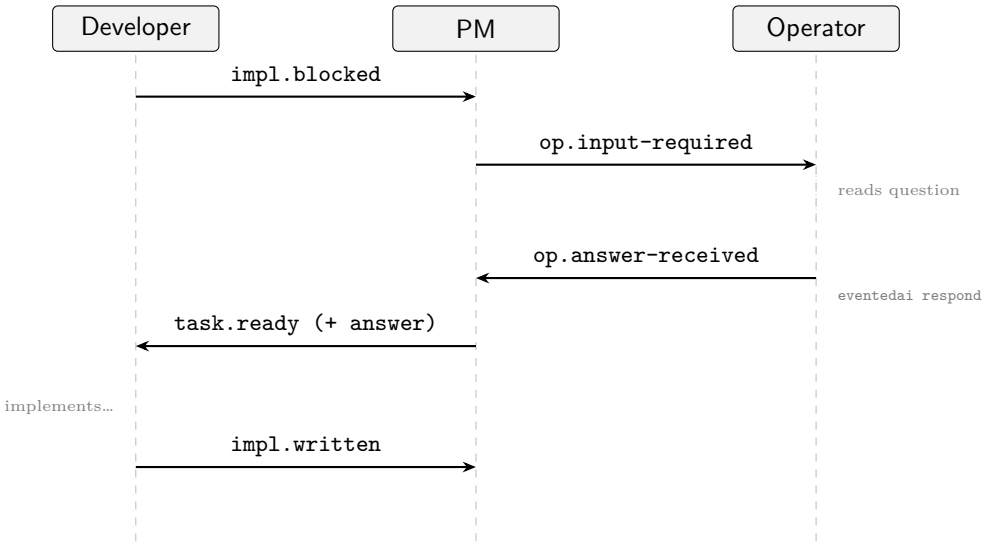


Figure 5: The operator loop. When a developer cannot proceed, it emits `impl.blocked`. The PM translates this into a human-readable question (`op.input-required`). The operator responds via `eventedai respond`, which emits `op.answer-received`. The PM incorporates the answer and re-emits `task.ready`. The entire exchange is causally linked in the event history.

7.3 Implementation Pragmas

Two pragmas specialize the agent interface. Their outputs must still satisfy the operational rules and any aggregate child bound used for termination.

`tether_file`: when an agent produces multiple named artifacts (e.g., an HTML style guide and a Markdown summary), downstream agents typically need only one. The `tether_file` pragma names the specific artifact to store as the event payload handle, keeping downstream context sizes manageable.

`multi_emit`: the formal model permits an output batch with multiple events, including multiple instances of one type. The pragma lets an agent return a JSON array, with one event per array element, each linked to the trigger. Planning agents use this to produce parallel tasks. A finite aggregate child bound must constrain the number of elements, not merely the number of declared output types.

7.4 Static Analyzer

A static analyzer checks team configurations before deployment. Given a configuration file, it verifies all conditions of Definition 20 and produces structured output listing violations with their locations. The analyzer is invoked as `eventedai check -team <config>` before `eventedai serve`. Conditions 0, 1, and 3 are enforced at configuration load time; the `check` command additionally verifies reachability (conditions 4 and 5) and detects structural deadlocks (condition 2) via Tarjan's SCC algorithm.

7.5 Formal-to-Implementation Traceability

Table 2 reports prototype mechanisms, not verification of all formal assumptions. In particular, an ancestry counter alone does not establish a finite aggregate child bound, durable once-only logical discharge, quiescence detection, or the reserved terminal-exhaustion policy required by Theorem 8. Full conformance to these requirements has not been established here. The prototype calls its failure event `agent.failed`, rather than `processing.failed`.

Formal claim/mechanism	**Runtime mechanism in EventedAI**	**Status**
Typed routing and emission checking	Agent subscriptions and emission-set validation at config-load and publish time	Implemented
Static coordination type checking (Definition 20)	`eventedai check`: terminal consistency, completeness, reachability, SCC structural deadlocks	Implemented
Ancestry-bounded firing (Definition 21)	Per-type ancestry counters with `cycle.exhausted` fallback	Mechanism only
Failure as first-class event	Structured runtime failure events (`agent.failed`); human-loop suspension events	Implemented
Structured emission declaration	First-line EVENT convention + validation; schema-constrained decoder under active design	Partially implemented
Retained-history reconstruction	Deterministic IDs + persisted artifacts; full reconstruction harness and evaluation	Proposed / partial

Table 2: Traceability from formal claims to prototype mechanisms.

8 Discussion

8.1 Generalization

The formal model makes no assumptions about what agents are. Human agents, AI agents, and automated services participate identically. A human responding to `op.input-required` via the command line is an agent: it consumes one event type and produces one event type.

Team configurations can be composed. A terminal event type in one team can serve as a root event type for another team, enabling hierarchical systems where one team's output kicks off another team's work. The parent-event link can be recorded as cross-team metadata, while each run retains its own CorrelationID equal to its RunID. Composition does not extend a bounded-termination guarantee across unlimited scopes.

8.2 The No-Orchestrator Property

The absence of a central orchestrator has two practical consequences. First, no distinguished agent holds authority over all other agents; failure handling can be declared independently

of that authority. This does not remove infrastructure failures or dependencies on another agent's results. Second, declared routes can be inspected without executing opaque decision logic. Actual processing still requires a recorded history and enforcement of the runtime rules.

8.3 Determinism and Replayability

LLM sampling and human judgment need not be repeatable. Accountability instead requires retaining what was decided and how work continued. Here replay means reconstruction of recorded history and artifacts, not re-execution of arbitrary black-box agents or external effects.

Recorded identity and attribution. Deterministic identity can make duplicate admission idempotent. One possible scheme hashes a canonical encoding of the run, trigger, producer, logical attempt, output index, type, and payload handle. The output index distinguishes intentional identical outputs in a batch. Duplicate delivery of the *same logical attempt*, with the same recorded output, reuses its identity; a *new attempt* has a new attempt identity. This needs collision handling and durable duplicate admission checks. Hashing alone neither enforces routing nor supplies missing producer metadata.

Retained payloads and decisions. Content-addressed handles identify payload bytes but do not preserve them: the referenced store must retain those bytes. Record model requests and responses and operator questions and answers against their originating obligation and attempt. A cache keyed only by model, prompt, and input must not conflate distinct sampled attempts. Retain original event bytes, including timestamps and identifiers, rather than regenerating them.

Admission checks. A conforming routing boundary validates subscriptions, agent emission sets, declared runtime control outputs, producer attribution, and causal parents. The prototype validates a first-line `EVENT:` convention; provider-constrained structured output and schema-locked payload admission are stronger design goals, not assumptions established for every current prototype path. Table 2 separates mechanisms from complete conformance.

Claim 2 (Retained-History Reconstruction)**.** Given a finite run's complete retained event bytes, producer and attempt metadata, referenced payload bytes, and recorded model responses and operator answers, the recorded events, causal edges, frontier, artifacts, and captured decisions can be reconstructed byte-for-byte without asking an agent or operator to decide again.

Reconstruction reads the retained bytes and follows their recorded links; it does not recover unrecorded actions. Re-executing preparation code or external tools would additionally require the relevant code, configuration, initial state, tool results, ordering, and all other variable inputs. No such general re-execution guarantee is made for the opaque transition functions of the model. Nor does reconstruction undo or repeat external side effects. Full prototype reconstruction remains proposed/partial.

8.4 Limitations

Conditional termination and completion. Theorem 8 requires once-only logical discharge, a finite aggregate child bound, ancestry guards, and terminal exhaustion with no internal subscribers. The prototype's ancestry mechanism does not alone establish these assumptions. External continuation needs a separately bounded scope. Eventual quiescence additionally requires eventual scheduling and finite processing or timeout; successful completion also requires a terminal frontier.

Quality gates. Gate-relative product correctness requires tree correctness but does not prove artifact correctness. Recorded producer and artifact references identify which check passed a defective artifact; this does not ensure that every artifact was checked adequately.

Retained history, not arbitrary re-execution. Reconstruction in Section 8.3 depends on retaining event and payload bytes and captured decisions. It does not reconstruct missing internal actions or reproduce model sampling, human judgment, or external effects.

Join patterns. The agent definition models single-event-triggered agents. Workflows that require an agent to wait for multiple correlated input events before firing—for example, aggregating parallel review results—require join patterns. The prototype implements a restricted form via a `compose_handles` pragma: an agent may declare a list of named handles; the runtime resolves each from Tether and concatenates them into a single composed input. This suffices for deterministic one-per-run joins (e.g., QA composing implementation and audit outputs) but does not generalize to fan-in over dynamic sets of events. A formal multi-input transition function would tighten the model and is deferred to future work.

Infrastructure as single point of failure. Removing a central orchestrating agent does not remove infrastructure dependencies. The NATS JetStream broker and the Tether content-addressed store are infrastructure-level SPOFs. High-availability configurations of both exist but are outside the scope of this paper.

Scale. We have not analyzed the model under large numbers of agents, high event throughput, or very long runs. JetStream's durable consumer model provides at-least-once delivery; exactly-once requires additional deduplication.

9 Related Work

Petri nets and workflow soundness. Van der Aalst's workflow nets [1] apply Petri net theory to business process modeling and establish a notion of *soundness*—reachability of a terminal marking, absence of dead transitions, and proper termination—that is directly analogous to our five static-analysis conditions (Theorems 2–6). A key difference is scope: workflow nets model a fixed schema of places and transitions, while our agents are typed black boxes with declared interface sets. We additionally model failure as a first-class routing event and provide content-addressed payload decoupling. Peltz [22] drew the early influential distinction between orchestration (a single coordinator directs others) and choreography (each participant declares its own behavior), which frames the central contrast of this paper.

Process calculi. CCS [16] and CSP [12] provide algebraic frameworks for concurrent process analysis. Both support formal reasoning about deadlock, liveness, and behavioral equivalence. Our agents are less expressive than arbitrary CCS processes—they cannot dynamically create channels or compute on channel names—but this restriction is deliberate:

static declaration of subscription and emission sets is precisely what enables the polynomial-time analysis in Section 4.

Choreographic programming. Choreographic programming languages [4, 17] take a top-down approach: the developer writes a global choreography from which each participant's local behavior is mechanically derived by *projection*, with static guarantees of deadlock freedom and progress across all participants. Our model inverts this: local interface declarations come first, and global topology is *derived* from the subscription graph. This inversion is necessary when agents are open-ended black boxes—LLMs, humans, external services—whose internal behavior cannot be fixed at choreography-design time. The tradeoff is that we lose the strong end-to-end protocol guarantees that projection provides; we recover structural weaker guarantees through static analysis.

Session types. Multiparty session types [10] give a type discipline to typed communicating processes in which a global protocol type specifies the interaction for all participants and is projected to local types checked against each process. Session types are strictly more expressive than our model: they support ordered sequences, branching, recursion, and full static verification of protocol conformance. Our agents are not session-typed—their interfaces are input/output *sets*, not ordered message sequences, and their behavior is nondeterministic. We do not claim session-type expressiveness; we claim suitability for the class of agents that *cannot* be session-typed.

Component-based coordination. The Reo coordination language [2] defines coordination as a compositional algebra of typed connectors between components, with formal analysis of connector behavior. BIP (Behavior-Interaction-Priority) [3] composes heterogeneous components through typed interaction ports with priority rules. Both provide richer compositional models than ours and are closer in spirit to session types and choreography. We differ in targeting heterogeneous, black-box AI agents whose behavior cannot be formally specified, relying instead on declared interface types and structural graph analysis.

Blackboard systems and shared-medium coordination. Blackboard systems [18]—originating with Hearsay-II—and Linda's generative tuple spaces [8] coordinate agents through a passive shared medium without direct communication. Our event bus plays an analogous role. Three structural differences separate our model from both: (1) events are immutable and causally linked, providing a durable audit trail absent from blackboard writes and Linda tuples; (2) typed subscription and emission declarations enable pre-execution analysis; (3) the event bus is ordered and provides at-least-once delivery guarantees.

Actor model. Hewitt's actor model [9] routes messages by address: actors may send messages to any known actor address, create new actors dynamically, and specify per-message behavior. Our agents are a restriction: they declare a fixed subscription set at configuration time and cannot send to arbitrary addresses. This restriction trades expressiveness for analyzability.

Event sourcing and the saga pattern. Event sourcing [6] treats the event log as the authoritative source of system state. Our contribution to this tradition is the formal characterization of the log as an occurrence net and the characterization of completion using its frontier and recorded processing obligations. The saga pattern [7] handles long-running transactions through sequences of compensating actions. Our failure mechanism generalizes the saga: failure events route through the same subscription graph as normal work, and the compensation logic is encoded in subscribing agents rather than in a separate compensating-

action specification.

LLM multi-agent systems. AutoGen [24] supports multi-agent conversation with customizable agent roles and group-chat routing but relies on orchestrator-mediated message passing without a formal coordination model. MetaGPT [11] assigns structured roles (product manager, engineer, QA) with standardized outputs exchanged through a shared environment, bearing the closest conceptual resemblance to our team model; it lacks formal analysis and causal accountability. ChatDev [25] structures LLM collaboration as a waterfall with role-playing phases. Park et al. [21] simulate social agent behavior in a sandbox environment. LangGraph [15] introduces graph structured execution flows over LLM agents, the closest existing system to our subscription-graph model; it does not provide formal static analysis or causal-event-tree accountability. None of these systems, to our knowledge, provide static decidability of coordination correctness or first-class causal treatment of failures.

10 Conclusion

We have presented a formal model of multi-agent AI coordination in which agents are typed transitions over event sets $\{X_1, \ldots, X_m\} \to \{Y_1, \ldots, Y_n\}$, execution history is an acyclic causal event tree (an occurrence net in the Petri net sense), and graph-structural coordination properties are statically decidable from a declarative configuration. The model unifies the handling of normal work, failure, and human intervention under a single mechanism—typed events in a shared stream—recording declared handoffs, failure handling, and captured decisions with their triggers and producers. It does not expose every internal action of an agent.

The key insight is that coordination and content can be separated. Graph-structural coordination properties are decidable from declarations; valid execution requires runtime enforcement. Finite history follows under explicit processing, branching, ancestry, and terminal-exhaustion assumptions, while eventual quiescence needs scheduling and timeout assumptions. Gate-relative success is not proof of artifact correctness. Retained history connects artifacts to recorded decisions and checks, allowing examination of failures without pretending to reproduce opaque reasoning or external effects.

The architecture generalizes to any system in which heterogeneous agents must coordinate over shared work. The model makes no assumptions about agent internals beyond the stated interfaces and runtime conditions. The conditional coordination properties apply equally to all of them.

References

[1] W. M. P. van der Aalst, "The application of Petri nets to workflow management," *Journal of Circuits, Systems, and Computers*, 8(1):21–66, 1998.

[2] F. Arbab, "Reo: A channel-based coordination model for component composition," *Mathematical Structures in Computer Science*, 14(3):329–366, 2004.

[3] A. Basu, M. Bozga, and J. Sifakis, "Modeling heterogeneous real-time components in BIP," in *Proc. 4th IEEE Int. Conf. on Software Engineering and Formal Methods (SEFM)*, pp. 3–12, IEEE, 2006.

[4] M. Carbone and F. Montesi, "Deadlock-freedom-by-design: Multiparty asynchronous global programming," *ACM SIGPLAN Notices*, 48(1):263–274, 2013.

[5] H. Chase, "LangChain," GitHub repository, 2022. https://github.com/langchain-ai/langchain

[6] M. Fowler, "Event sourcing," martinfowler.com, 2005.

[7] H. Garcia-Molina and K. Salem, "Sagas," *ACM SIGMOD Record*, 16(3):249–259, 1987.

[8] D. Gelernter, "Generative communication in Linda," *ACM Transactions on Programming Languages and Systems*, 7(1):80–112, 1985.

[9] C. Hewitt, P. Bishop, and R. Steiger, "A universal modular actor formalism for artificial intelligence," *IJCAI*, 1973.

[10] K. Honda, N. Yoshida, and M. Carbone, "Multiparty asynchronous session types," in *Proc. 35th ACM SIGPLAN-SIGACT Symp. on Principles of Programming Languages (POPL)*, pp. 273–284, 2008.

[11] S. Hong, X. Zheng, J. Chen, Y. Cheng, J. Wang, C. Zhang, Z. Wang, S. K. S. Yau, Z. Lin, L. Zhou, C. Ran, L. Xiao, and C. Wu, "MetaGPT: Meta programming for a multi-agent collaborative framework," *arXiv:2308.00352*, 2023.

[12] C. A. R. Hoare, "Communicating sequential processes," *Communications of the ACM*, 21(8):666–677, 1978.

[13] K. Jensen, "Coloured Petri nets," in *Petri Nets: Central Models and Their Properties*, pp. 248–299, Springer, 1987.

[14] L. Lamport, "Time, clocks, and the ordering of events in a distributed system," *Communications of the ACM*, 21(7):558–565, 1978.

[15] LangChain AI, "LangGraph," GitHub repository, 2024. https://github.com/langchain-ai/langgraph

[16] R. Milner, *A Calculus of Communicating Systems*, Springer, 1980.

[17] F. Montesi, "Introduction to choreographies," Cambridge University Press, 2023.

[18] H. P. Nii, "Blackboard systems: The blackboard model of problem solving and the evolution of blackboard architectures," *AI Magazine*, 7(2):38–53, 1986.

[19] M. Nielsen, G. Plotkin, and G. Winskel, "Petri nets, event structures and domains," *Theoretical Computer Science*, 13(1):85–108, 1981.

[20] OpenTelemetry Authors, "OpenTelemetry specification," https://opentelemetry.io/docs/specs/, 2024.

[21] J. S. Park, J. O'Brien, C. J. Cai, M. R. Morris, P. Liang, and M. S. Bernstein, "Generative agents: Interactive simulacra of human behavior," in *Proc. 36th ACM Symp. on User Interface Software and Technology (UIST)*, 2023.

[22] C. Peltz, "Web services orchestration and choreography," *IEEE Computer*, 36(10):46–52, 2003.

[23] C. A. Petri, *Kommunikation mit Automaten*, PhD thesis, University of Bonn, 1962.

[24] Q. Wu, G. Bansal, J. Zhang, Y. Wu, B. Li, E. Zhu, L. Jiang, X. Zhang, S. Zhang, J. Liu, A. H. Awadallah, R. W. White, D. Burger, and C. Wang, "AutoGen: Enabling next-gen LLM applications via multi-agent conversation," *arXiv:2308.08155*, 2023.

[25] C. Qian, X. Cong, C. Yang, W. Chen, Y. Su, J. Xu, Z. Liu, and M. Sun, "Communicative agents for software development," *arXiv:2307.07924*, 2023.