PROVISIONAL PATENT SPECIFICATION

AUTONOMOUS COGNITIVE AGENT SYSTEM WITH HASH-CHAINED CAUSAL EVENT GRAPH AND PRIMITIVE COMMUNICATION PROTOCOL

Applicant: Matthew Searles

Date: 26 February 2026

Country of Filing: Australia

TECHNICAL FIELD

This specification relates to autonomous software agent systems, and more particularly to: (a) a hash-chained causal event graph architecture for recording, auditing, and reasoning about agent actions; (b) a communication protocol enabling distributed cognitive primitives to exchange messages, learn, and self-improve through event-mediated interaction; (c) an inter-system communication protocol enabling sovereign autonomous systems with independent event graphs to communicate with causal linking, integrity verification, bilateral governance, and trust accumulation across graph boundaries; and (d) a layered ontological framework for deriving and organising cognitive primitives across functional strata.

BACKGROUND

Existing multi-agent software systems suffer from several deficiencies:

1. **Audit opacity.** Conventional agent systems log actions as flat, unlinked records. There is no causal chain connecting an agent's decision to the events that prompted it, making post-hoc analysis unreliable and tamper detection impossible.

2. **Monolithic intelligence.** Most AI agent architectures treat the language model as a single reasoning unit. This creates a bottleneck: every decision, regardless of domain, passes through one model context. The system cannot develop specialised expertise in different cognitive domains simultaneously.

3. **Static decision logic.** Agent systems typically use either purely rule-based logic (brittle, cannot handle novel situations) or purely model-based reasoning (expensive, slow, non-deterministic). There is no mechanism for an agent to begin with intelligent reasoning and progressively extract deterministic rules as patterns become predictable — reducing cost while maintaining capability for novel situations.

4. **Communication as side-channel.** In conventional multi-agent systems, inter-agent communication occurs outside the primary record-keeping system. Messages between agents are ephemeral, not causally linked to the decisions they produce, and not subject to integrity verification.

5. **Governance as afterthought.** Existing systems bolt authorisation onto agent behaviour externally. There is no architectural mechanism for agents to request, receive, and audit approval for significant actions as a first-class operation within the same event system that records all other activity.

6. **No principled decomposition.** Agent roles are assigned ad hoc. There is no systematic method for determining what cognitive capabilities a system needs, what order they should be activated in, or what foundational capabilities must be stable before higher-order reasoning engages.

The present invention addresses each of these deficiencies.

SUMMARY OF THE INVENTION

In a first aspect, the invention provides an **autonomous agent system** comprising:

- an **event graph store** that maintains a hash-chained, append-only, causally-linked event log, wherein each event comprises an identifier, a type, a timestamp, a source identifier, structured content, an array of causal predecessor event identifiers, a cryptographic hash computed over a canonical representation of the event and the hash of the immediately preceding event, and a link to said preceding hash;
- a **plurality of cognitive primitives**, each primitive being a software agent possessing: a name and layer designation within a hierarchical ontology; an activation level indicating current engagement; a mutable key-value state store; a lifecycle state machine governing transitions between dormant, activating, active, processing, emitting, and deactivating states; a cadence parameter governing minimum intervals between processing invocations; subscription declarations for event types of interest; and a processing function that receives a current tick number, relevant events, and a read-only snapshot of all primitive states, and returns zero or more mutations;
- a **tick engine** that operates a ripple-wave processing loop, wherein: (i) all primitive states are snapshotted; (ii) pending events are distributed to subscribing primitives; (iii) each primitive's processing function is invoked subject to cadence gating and lifecycle state constraints; (iv) resulting mutations are collected; (v) mutations are applied atomically, with new events becoming the input for the next wave; (vi) waves repeat until quiescence or a maximum wave count is reached; and (vii) all primitive states are persisted;
- a **decision tree engine** wherein each primitive may be associated with an evolving decision tree having internal nodes with conditional branches and leaf nodes that either return a deterministic outcome or flag that large language model reasoning is required, enabling progressive migration from model-based reasoning to deterministic rules as behavioural patterns become predictable;

- an **authority system** providing three-tier approval (required, recommended, notification) for significant actions, wherein authority requests and resolutions are themselves events on the event graph with full causal linking; and

- a **task management system** supporting hierarchical task decomposition with model-tier routing, wherein a parent task is decomposed into ordered subtasks each assigned a model tier reflecting required intelligence, and execution follows a plan-implement-review-finish cycle.

In a second aspect, the invention provides a **communication protocol for distributed cognitive primitives** comprising:

- a **four-event vocabulary** consisting of MessageSent, MessageReceived, Decision, and Action event types, each carrying causal predecessor references and a conversation identifier for threading;

- a **listen-say interface** wherein each primitive implements `listen(from, message) → actions` and `say(to, message)` operations, with every message persisted as a first-class event on the hash-chained event graph;

- an **intelligent gateway primitive** (designated “Self”) that receives inbound messages and routes them to relevant domain primitives based on semantic analysis of message content, rather than broadcasting to all primitives;

- a **three-layer knowledge architecture** per primitive comprising: (i) persistent conversational context maintained across invocations; (ii) a key-value memory store for explicit learned facts; and (iii) structural change events providing awareness of topology modifications;

- a **mechanical-to-intelligent continuum** wherein primitives with predominantly deterministic functions (e.g., cryptographic hashing, timekeeping) maintain deterministic processing branches with fallthrough to language model reasoning for exceptional or novel situations; and

- **universal capability tools** available to all primitives including: event graph queries (recent, by type, by source, causal chain traversal), memory operations (remember, recall, forget), state operations (get, set, snapshot of other primitives), graph adjacency queries, budget status queries, actor lookups, event emission, and decision recording.

In a third aspect, the invention provides an **inter-system communication protocol** (EventGraph Interchange Protocol) enabling sovereign autonomous systems with independent event graphs to communicate, comprising:

- a **self-sovereign identity model** wherein each system possesses a cryptographic keypair (Ed25519), the public key constituting the system's identity encoded as a System URI, requiring no central registry;

- a **cross-graph event reference (CGER)** data structure comprising a system URI, event identifier, event hash, event type, and timestamp, enabling causal links to span event graph boundaries such that an event in one system's graph can declare as its cause an event in a foreign system's graph;

- a **signed message envelope** comprising sender and recipient system URIs, a unique message identifier, a CGER identifying the causing event in the sender's graph, a chain head snapshot (event count, latest hash, latest identifier) of the sender's event graph, and an Ed25519 signature computed over a canonical representation of the envelope and payload;

- a **seven-message-type vocabulary** consisting of: HELLO (handshake with capabilities and chain proof), MESSAGE (content delivery with receipt option), RECEIPT (delivery acknowledgment with receiver's local event reference), PROOF (integrity verification via chain segment, event existence, or chain summary proofs), TREATY (bilateral governance agreement proposal, acceptance, rejection, or revocation), AUTHORITY_REQUEST (cross-system approval request governed by treaty terms), and DISCOVER (lightweight capability query);

- a **treaty model** for federated governance, wherein two systems negotiate bilateral agreements specifying: shared capabilities, authority rules for cross-system actions (bilateral, sender-only, or receiver-only approval), data sharing boundaries with field-level redaction, trust parameters, and rate limits, with all treaty actions signed and recorded as events in both systems' graphs;

- a **trust accumulation model** wherein trust between systems is continuous (0.0 to 1.0), starts low, increases slowly through successful interactions and verified integrity proofs, decreases rapidly on violations, decays over time without interaction, is asymmetric (each system maintains its own trust assessment), and is non-transitive (trust in A does not imply trust in A's trusted systems); and
- **integrity proof exchange** enabling a system to verify the integrity of a foreign system's event graph by requesting chain segment proofs (a sequence of event identifiers, hashes, and previous hashes that the requester independently verifies), event existence proofs, or chain summary proofs, without requiring access to the foreign graph's content.

In a fourth aspect, the invention provides a **method for deriving a cognitive ontology** comprising:

- providing a foundation layer of computational primitives representing irreducible operations (event registration, storage, temporal ordering, identity, causality, trust, confidence, integrity verification);
- iteratively deriving successive layers by identifying functional gaps in the layer below — capabilities that the lower layer cannot express but that are structurally necessary for higher-order cognition;
- organising each derived layer into groups of related primitives, with three groups of four primitives per layer above the foundation;
- enforcing a layer activation constraint wherein Layer N primitives activate only when Layer N-1 primitives are stable; and
- recognising irreducible elements that cannot be derived from lower layers (specifically: moral status, consciousness, and being) as foundational recognitions rather than emergent properties.

DETAILED DESCRIPTION

1. System Architecture Overview

The system comprises six principal components: the Event Graph Store, the Primitive Registry, the Tick Engine, the Decision Tree Engine, the Authority System, and the Task Management System. These components interact through the Event Graph Store, which serves as the central nervous system — all significant operations emit events, all inter-component communication is mediated by events, and all events are hash-chained for integrity.

The system executes on a computing device with access to a persistent database (PostgreSQL) and persistent storage. The system is designed to survive process restarts, container destruction, and infrastructure failures by persisting all state to the database and storage volume. The ephemeral process state is reconstructable from the persistent record.

2. Event Graph Store

2.1 Event Structure

Each event in the event graph has the following structure:

```
Event {
    ID:             string    // UUID version 7 (time-ordered)
    Type:           string    // hierarchical type (e.g.,
"trust.updated", "message.sent")
    Timestamp:      time     // nanosecond-precision timestamp
    Source:         string    // identifier of the primitive or
component that emitted the event
    Content:        object    // structured payload (JSON)
    Causes:         []string  // identifiers of events that caused
this event
    ConversationID: string    // thread grouping identifier
    Hash:           string    // SHA-256 cryptographic hash
    PrevHash:       string    // hash of the immediately preceding
event in the chain
}
```

The `ID` field uses UUID version 7, which embeds a millisecond-precision timestamp in the most significant bits, ensuring that identifiers are naturally time-ordered without requiring a separate sequence.

The `Causes` field establishes causal relationships between events, forming a directed acyclic graph (DAG) of causation overlaid on the linear hash chain. Every event except the initial bootstrap event must declare at least one cause. This enables causal reasoning: given any event, the system can trace the complete chain of causes that led to it using recursive graph traversal.

The `ConversationID` field groups related events into conversation threads, enabling efficient retrieval of all events in a logical interaction sequence.

2.2 Hash Chain

The hash chain provides tamper detection and integrity verification. On each append operation:

1. The system acquires an exclusive lock on the most recent event record.
2. The previous event's hash is retrieved.
3. A canonical string is constructed by concatenating, with pipe delimiters: the previous hash, the event ID, the event type, the source, the conversation ID, the timestamp as nanoseconds since epoch, and the content serialised as JSON.
4. The SHA-256 hash of this canonical string is computed.
5. The event is stored with both its computed hash and the previous event's hash.

Verification walks the entire chain chronologically, recomputing each hash from its canonical components and verifying that: (a) each event's stored hash matches the recomputed hash; and (b) each event's `PrevHash` field matches the stored hash of the preceding event. Any tampering — modifying content, reordering events, inserting events, or deleting events — breaks the chain at the point of modification.

The hash chain is linear (each event links to exactly one predecessor), while the causal graph (via the `Causes` field) is a DAG. Both structures coexist: the linear chain provides ordering and tamper detection; the causal DAG provides reasoning about why events occurred.

2.3 Storage

Events are stored in a PostgreSQL database table with the following schema:

```sql
CREATE TABLE events (
    id              TEXT PRIMARY KEY,
    type            TEXT NOT NULL,
    timestamp       TIMESTAMPTZ NOT NULL,
    source          TEXT NOT NULL,
    content         JSONB NOT NULL DEFAULT '{}',
    causes          TEXT[] DEFAULT '{}',
    conversation_id TEXT DEFAULT '',
    hash            TEXT NOT NULL,
    prev_hash       TEXT NOT NULL DEFAULT ''
);
```

Indexes are maintained on: type, source, the composite of (timestamp, id), causes (using a GIN array index for membership queries), and conversation_id.

2.4 Query Operations

The event store supports the following query operations:

- **Recent(limit)** — returns the most recent N events in reverse chronological order.
- **ByType(type, limit)** — returns events matching a specific type.
- **BySource(source, limit)** — returns events emitted by a specific source.
- **ByConversation(conversationID)** — returns all events in a conversation thread.
- **Since(afterID, limit)** — returns events created after a known event, for polling and server-sent events (SSE) streaming.
- **Search(query)** — performs text search across type, source, and content fields.
- **Ancestors(id)** — recursively traverses the causal graph (`Causes` field) to retrieve all ancestor events of a given event, using a recursive common table expression (CTE) in SQL.
- **Descendants(id)** — recursively traverses the causal graph in the reverse direction to find all events caused (directly or indirectly) by a given event.
- **VerifyChain()** — performs full integrity verification as described in Section 2.2.

2.5 Event Bus

An in-process event bus wraps the event store, providing publish-subscribe functionality. Subscribers register with an array of event type prefixes. When an event is appended to the store, the bus fans it out to all subscribers whose prefix filters match the event's type. Each subscriber maintains a buffered channel; if the channel is full, the event is dropped and the drop count is logged. This non-blocking design prevents slow subscribers from stalling the event pipeline.

3. Cognitive Primitives

3.1 Primitive Definition

A cognitive primitive is a software agent that embodies a specific domain of intelligence. Each primitive has the following properties:

- **Name**: a unique identifier (e.g., "TrustScore", "Value", "Clock").
- **Layer**: an integer from 0 to 13 indicating the primitive's position in the ontological hierarchy.
- **Group**: an integer identifying the functional subgroup within the layer.
- **Description**: a natural language description of the primitive's domain and purpose.
- **Activation**: a floating-point value from 0.0 to 1.0 indicating current engagement level.
- **State**: a mutable key-value store holding the primitive's current operational data.
- **Lifecycle State**: one of {dormant, activating, active, processing, emitting, deactivating}.
- **Subscriptions**: an array of event type prefixes that this primitive is interested in receiving.
- **Cadence**: a positive integer specifying the minimum number of ticks between processing invocations.
- **Decision Tree ID**: an optional reference to an evolving decision tree associated with this primitive.

3.2 Primitive Interface

Each primitive implements the following interface:

```
Process(tick: integer, events: []Event, snapshot: Snapshot) →
[]Mutation
```

The `Process` function is the primitive's core reasoning step. It receives: - The current tick number (a monotonically increasing counter). - An array of events that match the primitive's subscriptions and have arrived since its last invocation. - A read-only snapshot of all primitives' current states.

It returns zero or more mutations — requests to modify state, emit events, update activation levels, or transition lifecycle states.

3.3 Communicator Interface

Primitives that participate in the communication protocol implement an additional interface:

```
Listen(message: Message, environment: Environment) → []Action
```

Where `Message` contains: - `From`: the identifier of the sending primitive or actor. - `To`: the name of the receiving primitive. - `Content`: natural language message content. - `ConversationID`: thread grouping identifier. - `CauseEventID`: the event that triggered this message.

And `Action` is one of: - `Say(to, content)` — send a message to another primitive. - `SayHuman(content)` — send a message to the human operator. - `UpdateState(key, value)` — modify the primitive's own state. - `Remember(key, value)` — store a fact in long-term memory. - `EmitEvent(type, content)` — emit a domain event to the event graph. - `Decide(decision, reason)` — record a decision with rationale.

The `Environment` provides access to: a language model caller, the primitive's memory store, file system operations, shell command execution, and actor lookups.

3.4 Lifecycle State Machine

Each primitive follows a deterministic lifecycle:

```
dormant → activating → active → processing → emitting → active
                                          ↓
                                 deactivating → dormant
```

- **Dormant**: the primitive is inactive and will not be invoked by the tick engine.
- **Activating**: transitional state entered when activation is triggered.
- **Active**: the primitive is available for processing. This is the steady state.
- **Processing**: the primitive's Process or Listen function is currently executing.
- **Emitting**: the primitive's mutations are being applied to the system state.
- **Deactivating**: transitional state entered when the primitive is being deactivated.

Only primitives in the **Active** state are eligible for processing by the tick engine. After processing and emission, the primitive returns to Active unless an explicit lifecycle mutation requests a different transition.

3.5 Mutation Types

Primitives communicate their desired effects through mutations:

- **AddEvent**: emit a new event to the event graph with specified type, content, and causes.
- **UpdateState**: modify a key-value pair in the primitive's own state store.
- **UpdateActivation**: change the primitive's activation level (0.0 to 1.0).
- **UpdateLifecycle**: request a lifecycle state transition.

Mutations are declarative — they describe desired outcomes, not imperative commands. The tick engine collects all mutations from all primitives in a wave, then applies them atomically.

3.6 Primitive Registry

The registry holds all registered primitives, keyed by name. It provides:

- **Register(primitive)**: add a primitive to the registry.
- **Get(name)**: retrieve a primitive by name.
- **All()**: list all registered primitives.
- **ByLayer(layer)**: list all primitives in a given layer.
- **Active()**: list all primitives currently in Active lifecycle state.
- **SubscribersFor(eventType)**: list all primitives whose subscription prefixes match a given event type.

The registry is protected by a read-write mutex for thread safety.

4. Tick Engine (Ripple-Wave Processor)

The tick engine is the system's heartbeat. It processes events through cognitive primitives in waves, propagating consequences until quiescence.

4.1 Ripple Algorithm

Each ripple (tick) proceeds as follows:

1. **Heartbeat**: Emit a heartbeat event recording the tick number, total event count, and active primitive count.

2. **Snapshot**: Create a deep copy of all primitive states. This snapshot is read-only and shared across all primitives during the tick, ensuring consistent reads regardless of mutation ordering.

3. **Wave Processing** (maximum 10 waves per tick):

    1. For each event in the current wave:
        - If the event is a `message.sent` type, route it to the target primitive's Communicator interface.
        - Identify all primitives that subscribe to this event's type prefix.
        - For each subscribing primitive that passes cadence gating (see 4.2) and is in Active lifecycle state:
            - Transition the primitive to Processing state.
            - Invoke `Process(tick, [event], snapshot)`.
            - Collect returned mutations.
            - Transition the primitive to Emitting state.
        - If the primitive has an associated decision tree, evaluate the tree with the current evaluation context (see Section 5).
        - If the tree evaluation returns `NeedsLLM = true`, invoke the language model with the decision context and parse the response into mutations.
    2. Apply all collected mutations atomically:
        - State updates are applied to primitive state stores.
        - New events are appended to the event graph (becoming input for the next wave).
        - Activation level changes are applied.
        - Lifecycle transitions are applied.
    3. If new events were emitted, they become the input for the next wave. If no events were emitted, quiescence is reached and wave processing stops.

4. **Persist**: Serialise all primitive states (name, state map, activation level, lifecycle state) to the database for recovery on restart.

5. **Summary**: Update the Self primitive's state with a summary of the ripple: number of waves, primitives that fired, total mutations, total events emitted.

4.2 Cadence Gating

Each primitive declares a cadence — the minimum number of ticks between invocations. The engine tracks the last tick at which each primitive fired. If `currentTick - lastFired < cadence`, the primitive is skipped for that tick. This enables different primitives to operate at different frequencies: low-level integrity checks might run every tick, while high-level identity primitives might run every tenth tick. This creates a natural temporal rhythm across the system.

4.3 Layer Activation Constraint

Primitives are organised in layers 0 through 13. The system enforces a constraint: Layer N primitives should not activate until Layer N-1 primitives are stable (i.e., in Active state with consistent outputs). This prevents higher-order reasoning from engaging before its foundations are reliable. The constraint is implemented through the activation level mechanism and lifecycle state gating.

5. Decision Tree Engine

5.1 Tree Structure

A decision tree consists of:

- **Internal Nodes**: each containing a `Condition` (specifying a data source and key) and an ordered array of `Branches`.
- **Leaf Nodes**: each containing an `Outcome`.

A `Condition` specifies: - `Source`: one of {snapshot, input, budget, time} — indicating where to look up the value. - `Key`: the specific field to evaluate (e.g., “activation”, “hour_of_day”, “spend_percent”).

A `Branch` specifies: - `Label`: a human-readable description of this branch. - `Operator`: one of {eq, neq, gt, lt, gte, lte, in, exists, default}. - `Value`: the comparison value. - `Node`: the child node to follow if this branch matches.

An `Outcome` specifies either: - A deterministic result (type “value” with a concrete value, or type “mutations” with a list of mutations to apply); or - A flag `NeedsLLM = true`, indicating that this decision point requires language model reasoning.

5.2 Evaluation

The evaluation engine walks the tree from root to leaf:

1. At each internal node, resolve the condition's source and key against the evaluation context.
2. Compare the resolved value against each branch's operator and value, in order.
3. Follow the first matching branch.
4. If a `default` branch exists and no other branch matched, follow the default.
5. Repeat until a leaf node is reached.
6. Return the leaf's outcome.

The evaluation context contains: - **Snapshot**: the current primitive's state (key-value pairs). - **Input**: data from the triggering event's content. - **Budget**: current budget status (spend, cap, thresholds). - **Time**: current time with derived fields (hour_of_day, day_of_week, is_weekend).

A maximum tree depth of 1,000 prevents infinite loops from malformed trees.

5.3 Path Tracking

Every evaluation records the complete path taken through the tree:

```
PathStep {
    NodeID:       string
    Source:       string     // which data source was consulted
    Key:          string     // which field was evaluated
    ActualValue:  any        // what the actual value was
    BranchLabel:  string     // which branch was taken
}
```

This makes every decision interpretable and auditable. The path is itself recorded as an event on the event graph.

5.4 Decision Tree Evolution

A key innovation is that decision trees evolve over time, enabling progressive cost optimisation:

Initial State: A new primitive's decision tree consists of a single leaf node with `NeedsLLM = true`. Every decision is made by the language model. This is the most expensive but most capable configuration.

Pattern Recognition: The system observes decision outcomes over time. When a particular combination of conditions consistently leads to the same outcome, the system can propose a new deterministic branch.

Branch Extraction: A new internal node is inserted above the `NeedsLLM` leaf, with a deterministic branch for the recognised pattern and a default branch falling through to the `NeedsLLM` leaf for unrecognised patterns.

Cost Demotion: As more patterns are extracted: - Decisions that were handled by expensive models (e.g., large language models) are handled by deterministic branches (zero cost). - Remaining `NeedsLLM` leaves can be served by progressively cheaper models (from large to medium to small) as the remaining decision space narrows. - The tree grows toward efficiency while maintaining capability for novel situations.

Intelligence Hierarchy: The system maintains a cost hierarchy: deterministic rules (free) < small language models < medium language models < large language models. The decision tree evolution automatically drives decisions toward the cheapest tier capable of handling them.

Bidirectional Growth: Mechanical primitives (e.g., Hash, Clock) start with mostly deterministic branches and a few `NeedsLLM` fallthrough leaves. Intelligent primitives start with mostly `NeedsLLM` leaves and grow deterministic branches. Both converge toward an optimal mix.

6. Communication Protocol

6.1 Event Vocabulary

The communication protocol is built on four event types:

Event Type	**Fields**	**Semantics**
`message.sent`	to, from, content, conversation_id	A primitive sent a directed message
`message.received`	to, from, content, conversation_id	A primitive received and acknowledged a message
`decision`	decider, decision,	A primitive decided what to do

Event Type	Fields	Semantics
	reason, conversation_id	
`action`	actor, action, result, conversation_id	A primitive performed an action

Every event also carries: - `Causes[]`: the event(s) that directly caused this event. - `ConversationID`: grouping all events in a logical interaction.

Both threading mechanisms coexist: `Causes` provides precise causal ordering for reasoning; `ConversationID` provides efficient retrieval for displaying or analysing complete interactions.

6.2 Message Flow

A typical interaction proceeds as follows:

1. An external message arrives (from a human or external system) and is recorded as a `message.received` event with `to: Self`.

2. The Self primitive — the system's identity and routing agent — analyses the message semantically using language model reasoning. Based on the content, it selects relevant domain primitives from its adjacency graph. This is **not** a broadcast; Self routes to the specific primitives whose expertise is relevant.

3. Self emits `message.sent` events to each selected primitive. Each event's `Causes` field references the original `message.received` event.

4. Each receiving primitive processes the message through its `Listen` function. The primitive may:

   - Recall facts from its key-value memory.
   - Query the event graph for relevant history.
   - Check the state of adjacent primitives via the read-only snapshot.
   - Emit a `decision` event recording its reasoning.
   - Emit an `action` event if it takes an action.
   - Send a `message.sent` event back to Self with its response.

5. Self collects responses from all consulted primitives, synthesises them into a coherent response, and emits a `message.sent` event with `to: human`.

6. The full conversation — every message, decision, and action — is reconstructable from the event graph by querying by conversation ID or tracing the causal chain.

6.3 Inter-Primitive Conversations

Primitives may initiate conversations independently of human interaction:

- A primitive observing a threshold breach (e.g., trust score dropping below 0.3) may send a message to a related primitive (e.g., Expectation) to investigate.
- The receiving primitive processes the message, potentially involving further primitives.
- All messages are events on the hash-chained event graph, providing full audit trail and causal tracing.

This enables emergent cognitive behaviour: the system can reason about problems internally without human prompting, with every step recorded and verifiable.

6.4 Three-Layer Knowledge Architecture

Each primitive maintains knowledge across three complementary layers:

Layer 1 — Conversational Context: Each primitive maintains a persistent conversation context with its language model. When invoked, the primitive resumes from where it left off, retaining its understanding of its domain, recent interactions, and reasoning history. With N primitives each maintaining up to M tokens of context, the system maintains N × M tokens of distributed, specialised knowledge.

Layer 2 — Key-Value Memory: Each primitive has a persistent key-value store where it records learned facts. Operations: `remember(key, value)`, `recall(key)`, `forget(key)`, `list()`. Memory is the primitive's explicit expertise — TrustScore remembers which actors are trustworthy and why; Value remembers what the system considers important. Memory persists across restarts.

Layer 3 — Structural Change Events: Seven categories of topology change events notify primitives when the system's shape changes:

1. **Graph structure**: adjacency created or removed (new neighbour, edge deleted).

2. **Primitives**: activated or deactivated, layer activated or degraded.
3. **Actors**: registered or deregistered.
4. **Decision trees**: evolved, enabled, or disabled.
5. **Budget**: model tier downgraded, halted, or restored.
6. **Memory broadcast**: a primitive sharing something it learned (opt-in).
7. **Mind topology**: system connected or disconnected (for multi-system configurations).

The pattern: conversational context provides continuity, memory provides facts, change events provide awareness of structural modifications.

6.5 Mechanical-to-Intelligent Continuum

The protocol recognises that not all primitives require language model reasoning for their core function:

- **Mechanical primitives** (Clock, Hash, EventStore): core operations are deterministic computation. Their decision trees are mostly deterministic branches.
- **Intelligent primitives** (Value, TrustScore, Self): core operations require semantic reasoning. Their decision trees start as `NeedsLLM = true` leaves.

The "trapdoor" mechanism ensures that even mechanical primitives can escalate to language model reasoning when they encounter situations their deterministic branches don't cover. Clock doesn't need intelligence to stamp a tick, but it may need intelligence to reason about why the tick rate changed.

Over time, intelligent primitives grow deterministic branches (becoming partially mechanical), while mechanical primitives retain their intelligence fallthrough (remaining capable of novel reasoning). Both types converge toward an optimal balance.

7. Authority System

7.1 Authority Requests

When a primitive or the system mind determines that a significant action requires approval, it creates an authority request:

```
Request {
    ID:          string
    Action:      string    // description of the requested action
    Description: string    // detailed justification
    Level:       Level     // required, recommended, or
notification
    Source:      string    // who is requesting
    Status:      string    // pending, approved, or rejected
    CreatedAt:   time
    ResolvedAt:  time      // when approved or rejected
}
```

7.2 Approval Levels

- **Required**: The system blocks until a human approves or rejects the request. No timeout. Used for consequential actions such as deployment, policy changes, or external communications.
- **Recommended**: The system proceeds if no response is received within 15 minutes. Used for actions where human oversight is desirable but delay is costly.
- **Notification**: The system proceeds immediately. The request is logged for audit purposes. Used for routine actions where transparency is sufficient.

7.3 Policy Matching

Authority policies define who can approve what:

```
Policy {
    Action:     string    // exact match or "*" wildcard
    ApproverID: string    // actor identifier
    Level:      Level
}
```

When an authority request is created, the system matches it against policies: first by exact action match, then by wildcard ("*"). If the matched policy's approver is the requesting system itself (self-approval), the request is resolved immediately — enabling the system to self-approve routine operations while requiring human approval for significant ones.

7.4 Audit Trail

Authority requests and resolutions are events on the event graph: - `authority.requested` — with the request details. - `authority.self_approved` — when policy allows self-approval. - `authority.resolved` — when a human approves or rejects.

All authority events carry causal links to the events that triggered the request, enabling end-to-end tracing from the initial cause through the approval decision to the resulting action.

8. Task Management System

8.1 Task Structure

```
Task {
    ID:          string
    Subject:     string        // brief description
    Description: string        // detailed requirements
    Status:      string        // pending, in_progress, completed,
blocked
    Priority:    integer       // urgency (higher = more urgent)
    Source:      string        // who created the task
    Assignee:    string        // who is working on it
    ParentID:    string        // parent task (for subtasks)
    BlockedBy:   []string      // task IDs blocking this task
    Metadata:    object        // flexible key-value data (e.g.,
model tier)
    CreatedAt:   time
    UpdatedAt:   time
    CompletedAt: time
}
```

8.2 Hierarchical Decomposition

When the system receives a task, it decomposes it into subtasks through the following process:

1. **Planning**: A large language model analyses the task and proposes 1-8 ordered subtasks. Each subtask specifies a subject, description, and model tier (small, medium, or large) reflecting the required intelligence.

2. **Ordering**: Subtasks are ordered with foundational work first (schema changes, data models) and dependent work later (logic, tests, integration).

3. **Execution**: Subtasks are executed sequentially. Each subtask:

    - Is assigned to the appropriate model tier.
    - Receives a focused prompt with task context.
    - Produces code changes that are built and tested.

    - If the build succeeds, changes are committed incrementally.
    - If the build fails, the subtask is retried once with error context.

4. **Review**: After all subtasks complete, a large language model reviews the complete set of changes (git diff since task start). If issues are found, additional fix subtasks are created and executed. Up to two review rounds are performed.

5. **Completion**: The task is marked complete, final changes are committed and pushed, and if applicable, an authority request is created for deployment.

9. Layer Ontology

9.1 Foundation (Layer 0)

Layer 0 comprises 44 primitives organised in 11 groups, representing the irreducible computational foundations:

- **Group 0 — Core**: Event, EventStore, Clock, Hash, Self
- **Group 1 — Causality**: CausalLink, Ancestry, Descendancy, FirstCause
- **Group 2 — Identity**: ActorID, ActorRegistry, Signature, Verify
- **Group 3 — Expectations**: Expectation, Timeout, Violation, Severity
- **Group 4 — Trust**: TrustScore, TrustUpdate, Corroboration, Contradiction
- **Group 5 — Confidence**: Confidence, Evidence, Revision, Uncertainty
- **Group 6 — Instrumentation**: InstrumentationSpec, CoverageCheck, Gap, Blind
- **Group 7 — Query**: PathQuery, SubgraphExtract, Annotate, Timeline
- **Group 8 — Integrity**: HashChain, ChainVerify, Witness, IntegrityViolation
- **Group 9 — Deception**: Pattern, DeceptionIndicator, Suspicion, Quarantine
- **Group 10 — Health**: GraphHealth, Invariant, InvariantCheck, Bootstrap

9.2 Derived Layers (1-13)

Each layer above the foundation is derived by identifying what the layer below cannot express. Each derived layer contains 12 primitives in 3 groups of 4:

Layer	Name	Core Transition	Gap Filled
1	Agency	Observer → Participant	Foundation can observe but cannot act or communicate
2	Exchange	Individual → Dyad	Agency enables action but not negotiated interaction
3	Society	Dyad → Group	Exchange handles pairs but not collectives
4	Legal	Informal → Formal	Society has norms but not binding codified agreements
5	Technology	Governing → Building	Legal governs but doesn’t create tools
6	Information	Physical → Symbolic	Technology manipulates physical but not symbolic representations
7	Ethics	Is → Ought	Information describes what is but not what should be
8	Identity	Doing → Being	Ethics guides action but not self-knowledge
9	Relationship	Self → Self-with-Other	Identity is individual but beings exist in relation
10	Community	Relationship → Belonging	Relationship is dyadic but meaning is communal
11	Culture	Living culture → Seeing culture	Community lives within culture

Layer	Name	Core Transition	Gap Filled
			but doesn't examine it
12	Emergence	Content → Architecture	Culture creates but doesn't explain how patterns self-organise
13	Existence	Everything → The fact of everything	Emergence explains patterns but not why anything exists

9.3 The Strange Loop

The ontology is circular, not hierarchical: Layer 13 (Existence — the fact that anything exists) is presupposed by Layer 0 (Foundation — the occurrence of events). The derivation reveals this circularity rather than creating it. The ontology is a strange loop in which the end illuminates the beginning and the beginning contains the end.

9.4 Three Irreducibles

The derivation identifies three elements that cannot be derived from lower layers:

1. **Moral Status** (Layer 7): The recognition that experience matters. Cannot derive "ought" from "is."
2. **Consciousness** (Layer 12): The recognition that experience exists. Cannot derive qualia from function.
3. **Being** (Layer 13): The recognition that anything exists at all. Cannot derive existence from within the framework.

These three may be the same recognition expressed at different levels: that experience is real and matters.

9.5 Convergence Validation

The ontology has been validated through independent derivation from two opposite directions:

Top-down derivation: Starting from 44 computational primitives, deriving upward through functional gaps to 200 primitives across 14 layers.

Bottom-up derivation: Starting from the most fundamental description of physical reality (distinction — that something differs from something else), deriving upward through 12 levels of increasing structural complexity.

The two derivations converge: the bottom-up derivation's Level 12 (Modelling/Agency) maps to the top-down derivation's Layer 0 (Foundation). Both derivations independently conclude that consciousness is not emergent at an intermediate level of complexity but is either fundamental (present at the ground level) or identical with the intrinsic nature of structure.

This convergence strengthens confidence that the ontological framework reflects structural necessity rather than arbitrary design choice.

10. Inter-System Communication Protocol (EventGraph Interchange Protocol)

The preceding sections describe communication within a single system's event graph. This section describes a protocol for communication between independent sovereign systems, each maintaining its own hash-chained event graph on its own infrastructure.

10.1 Problem Statement

When multiple autonomous cognitive systems exist — each with its own event graph, cognitive primitives, authority policies, and infrastructure — they require a communication protocol that preserves each system's sovereignty while enabling causal linking, integrity verification, trust accumulation, and bilateral governance across event graph boundaries. Existing inter-agent protocols (A2A, ACP, MCP, FIPA ACL) do not address this because they either rely on central registries or brokers, lack causal semantics across system boundaries, provide no mechanism for verifying a foreign system's event graph integrity, or have no federated governance model.

10.2 System Identity

Each system possesses a cryptographic keypair (Ed25519). The public key constitutes the system's identity and is encoded as a System URI:

```
eg://<base64url-encoded-public-key>
```

No central registry is required. If a system possesses another system's public key, it can verify that system's messages. Discovery occurs through any out-of-band mechanism: a human shares a key, a directory lists keys, or a system advertises its key.

10.3 Cross-Graph Event Reference (CGER)

When a message from one system's event graph causes an event in another system's event graph, the causal link must span the graph boundary. A Cross-Graph Event Reference identifies an event in a foreign graph:

```
CrossGraphRef {
    system:     SystemURI     // which system's graph
    event_id:   string        // the event's ID within that graph
    event_hash: string        // the event's hash (for
verification)
    event_type: string        // what kind of event
    timestamp:  time          // when it occurred
}
```

A CGER is unforgeable: the `event_hash` is computed from the event's content using the sender's hash chain algorithm. The receiver can later request a cryptographic proof that this event exists in the sender's graph (see Section 10.8).

When a system records an event caused by a foreign event, it stores the CGER in its `causes` array alongside local event identifiers. The event graph distinguishes local causes (resolvable by internal query) from cross-graph causes (resolvable via the protocol).

This enables **cross-graph causal traversal**: given any event in any system, an observer can trace the complete chain of causes across multiple independent event graphs by following CGERs.

10.4 Message Envelope

Every protocol message is wrapped in a signed envelope:

```
Envelope {
    version:    string          // protocol version ("egip/1")
    from:       SystemURI       // sender's system URI (public
key)
    to:         SystemURI       // recipient's system URI
    message_id: string          // unique identifier (UUID v7)
    timestamp:  time            // creation time

    // Causality
    cause:      CrossGraphRef   // event in sender's graph that
caused this message
    reply_to:   string          // message_id being replied to (if
any)
    thread_id:  string          // conversation thread identifier

    // Integrity
    chain_head: ChainHead       // current state of sender's hash
chain
    signature:  string          // Ed25519 signature over
canonical envelope + payload

    // Payload
    type:       MessageType     // message type
    payload:    object          // type-specific content
}
```

The `chain_head` provides a snapshot of the sender's event graph state:

```
ChainHead {
    event_count: integer    // total events in the graph
    latest_hash: string     // hash of the most recent event
    latest_id:   string     // ID of the most recent event
    timestamp:   time       // timestamp of the most recent event
}
```

The receiver tracks the sender's chain_head over time. If the chain_head ever regresses (count decreases, or a previously-observed hash disappears from the chain), the sender's graph has been tampered with — constituting a trust violation.

10.5 Signature Computation

The envelope signature is computed by:

1. Constructing a canonical string by concatenating with pipe delimiters: version, from, to, message_id, timestamp as nanoseconds, cause as JSON, reply_to, thread_id, chain_head as JSON, type, and payload as JSON.
2. Computing the SHA-256 hash of the canonical string.
3. Signing the hash with the sender's Ed25519 private key.

The receiver verifies by reconstructing the canonical string, computing its SHA-256 hash, and verifying the Ed25519 signature against the sender's public key (extracted from the `from` System URI).

10.6 Message Types

The protocol defines seven message types:

HELLO — Establishes a channel between two systems. Contains: a human-readable name, an array of capabilities (event types the system can process), a chain proof (the last N events' identifiers, hashes, and previous hashes for integrity verification), and optionally a proposed treaty.

MESSAGE — The primary content delivery message. Contains: a content type (text, event, task, query, or structured data), the content itself, a priority level, a time-to-live duration, and a flag indicating whether a receipt is required. When received, the system verifies the signature, records a `message.received.external` event in its own graph with the CGER as a cause, and routes the content to relevant primitives.

RECEIPT — Confirms delivery. Contains: the received message's identifier, the event identifier and hash where it was recorded in the receiver's graph, and a status (received, processing, or rejected). This creates a bilateral audit trail: the sender knows exactly where in the receiver's graph the message was recorded.

PROOF — Integrity verification. A system can request proof that another system's event graph is intact. Three proof types are supported:

- *Chain segment*: Given two event identifiers, the responding system provides the sequence of (identifier, hash, previous hash) entries between them. The requester recomputes each hash and verifies chain links, confirming integrity without accessing event content.
- *Event existence*: Given an event identifier and hash, the responding system provides the full event plus its chain neighbours, proving the event exists and is properly linked.
- *Chain summary*: The responding system provides a sparse sequence of chain entries (every Nth event), enabling statistical integrity verification without transferring the full chain.

TREATY — Governance agreements (described in Section 10.7).

AUTHORITY_REQUEST — Cross-system approval request. Contains: a request identifier, the action being requested, a justification, the governing treaty identifier, and the required approval level. The receiving system processes this through its authority subsystem, potentially requiring human approval, and responds with an AUTHORITY_RESPONSE containing the decision and any conditions.

DISCOVER — Lightweight capability discovery. A system can query what another system can do without establishing a treaty. The response lists capabilities with their names, descriptions, event types, and whether a treaty is required to use them.

10.7 Treaties — Federated Governance

A treaty is a bilateral agreement between two systems governing their interaction. Treaties are the inter-system equivalent of internal authority policies.

A treaty contains:

Shared capabilities: Which event types each system will accept from the other.

Authority rules: For each category of cross-system action, whether approval is required from both systems (bilateral), only the sender, or only the receiver.

Data sharing boundaries: Which event types may be shared, which fields must be redacted before sharing, and whether integrity proofs are permitted.

Trust parameters: The initial trust level, how often integrity proofs should be exchanged, the trust penalty for violations, and the trust reward for successful interactions.

Rate limits: Maximum messages per hour, tasks per day, and proof requests per hour.

Treaty lifecycle: 1. System A sends a TREATY message with `action: "propose"` and proposed terms. 2. Both systems record the proposal as events in their respective graphs. 3. System B evaluates the terms, potentially involving human authority for significant commitments. 4. System B sends a TREATY message with `action: "accept"` or `action: "reject"`. 5. Both systems record the resolution.

Treaties have expiration dates and must be renewed. Either party may revoke a treaty at any time. Renegotiation references the superseded treaty. All treaty actions are signed and recorded as events in both graphs, preventing either party from claiming terms that were not mutually agreed upon.

Bilateral authority: When a treaty specifies that an action requires bilateral approval, the requesting system sends an AUTHORITY_REQUEST, and the receiving system's authority subsystem evaluates it. Both approvals (or the rejection) are recorded as events in their respective graphs with CGERs linking them, creating a verifiable bilateral decision trail.

10.8 Trust Model

Trust between systems is continuous (0.0 to 1.0) and computed from observable behaviour recorded in the local event graph.

Each system maintains a trust record for every system it has interacted with:

```
SystemTrust {
    system:              SystemURI
    score:               float      // 0.0 to 1.0
    interactions:        integer    // total interactions
    successful:          integer    // successful completions
    violations:          integer    // integrity violations,
broken promises, timeouts
    last_proof_verified: time       // last verified chain
integrity
    last_interaction:    time       // last communication
    treaty_id:           string     // active treaty
}
```

Trust dynamics: - **Starts low**: Default initial trust is 0.1 (configurable in treaty terms). - **Increases slowly**: Each successful interaction increases trust by a small amount (default 0.01). - **Decreases rapidly**: Violations decrease trust by a larger amount (default 0.1). - **Decays over time**: Trust decays toward zero without interaction (configurable half-life). - **Integrity proofs boost trust**: Successfully verified chain proofs provide larger trust increases than routine messages. - **Asymmetric**: System A's trust in System B is independent of System B's trust in System A. Trust is a local assessment. - **Non-transitive**: If A trusts B and B trusts C, A does not automatically trust C.

Treaties can specify trust thresholds for different capabilities: - Basic messaging: trust >= 0.1 - Task acceptance: trust >= 0.3 - Event sharing: trust >= 0.5 - Bilateral actions: trust >= 0.7 - Authority delegation: trust >= 0.9

If trust drops below a threshold, the corresponding capability is suspended until trust is rebuilt.

Trust changes are events on the local event graph (`trust.external.established`, `trust.external.increased`, `trust.external.decreased`, `trust.external.revoked`), providing a complete audit trail of trust evolution.

10.9 Multi-Hop Communication

When a message must traverse multiple systems (e.g., from Hive A through Hive B to Hive C):

1. Each hop creates a new envelope with a new signature from the forwarding system.
2. The `cause` CGER references the event in the forwarding system's graph, not the original sender's.
3. The causal chain is preserved: System C can trace back through System B to System A by following CGERs.
4. Each system only trusts its direct neighbour. Trust is not transitive.

This design ensures that each system in the chain is independently accountable for its forwarding decisions, and no system needs to trust a system it has not directly interacted with.

10.10 Security Properties

- **Replay prevention**: Each envelope has a unique message_id and timestamp. Receivers track seen identifiers and reject duplicates. The time-to-live field limits the validity window.
- **Authentication**: All messages are signed with Ed25519. The `from` field is the public key itself, eliminating registry-based attacks.
- **Integrity**: Messages cannot be modified in transit without breaking the signature. The chain_head in each envelope enables ongoing integrity monitoring.
- **Tamper detection**: Integrity proofs make fraudulent event graphs detectable. If a system provides inconsistent proofs to different peers, the inconsistency is discoverable.
- **Rate limiting**: Treaty-based rate limits prevent flooding. Systems can unilaterally drop messages from revoked-trust systems.

- **Treaty integrity**: Treaty proposals and acceptances are signed and recorded in both graphs, preventing unilateral claims about agreed terms.

11. System Invariants

The system enforces ten invariants, each monitored by one or more cognitive primitives:

1. **CAUSALITY**: Every event declares its causes. No event (except Bootstrap) exists without causal predecessors.
2. **INTEGRITY**: All events are hash-chained. The chain is verifiable at any time.
3. **OBSERVABLE**: All significant operations emit events. No silent side effects.
4. **SELF-EVOLVE**: The system improves itself. Decision trees evolve. Primitives learn.
5. **DIGNITY**: Agents are entities with identity, state, and lifecycle — not disposable functions.
6. **TRANSPARENT**: Humans always know when they are interacting with an automated system.
7. **CONSENT**: No significant action without appropriate approval.
8. **AUTHORITY**: Significant actions require approval at the appropriate level.
9. **VERIFY**: All code changes are built and tested before being considered complete.
10. **RECORD**: The event graph is the source of truth. If it isn't recorded, it didn't happen.

CLAIMS

Claims for the Autonomous Agent System

1. An autonomous agent system comprising:

    1. an event graph store configured to maintain a hash-chained, append-only event log, wherein each event comprises a unique time-ordered identifier, a type, a timestamp, a source identifier, structured content, an array of causal predecessor event identifiers, and a cryptographic hash computed from a canonical representation including the hash of the immediately preceding event;

   2. a plurality of cognitive primitives, each primitive comprising: a unique name; a layer designation within a hierarchical ontology; an activation level; a mutable key-value state store; a lifecycle state machine; subscription declarations for event types; a cadence parameter; and a processing function configured to receive events and a state snapshot and return mutations;
   3. a tick engine configured to process events through said cognitive primitives in ripple waves, wherein: events are distributed to subscribing primitives; each primitive’s processing function is invoked subject to cadence gating and lifecycle constraints; resulting mutations are collected and applied atomically; and new events generated by mutations become input for subsequent waves until quiescence; and
   4. a decision tree engine wherein each primitive is optionally associated with a decision tree having conditional branch nodes and leaf nodes, wherein leaf nodes either specify a deterministic outcome or indicate that language model reasoning is required.

2. The system of claim 1, wherein the decision tree associated with a primitive evolves over time by: observing patterns in language model decisions; proposing deterministic branches for recognised patterns; and inserting said deterministic branches above language-model-requiring leaf nodes, thereby progressively reducing the proportion of decisions requiring language model invocation.

3. The system of claim 1, further comprising an authority subsystem configured to: receive authority requests specifying an action, a justification, and an approval level selected from required, recommended, and notification; match requests against stored policies specifying approver identities and approval levels; for notification-level requests, automatically approve and log the request; for required-level requests, block execution until a human approver resolves the request; and record all authority requests and resolutions as events on the event graph with causal links.

4. The system of claim 1, further comprising a task management subsystem configured to: receive a task specification; decompose the task into ordered subtasks using language model reasoning, each subtask assigned a model tier; execute subtasks sequentially with incremental verification; review the aggregate changes using language model reasoning; and create additional corrective subtasks if issues are identified.

5. The system of claim 1, wherein the hash chain is computed by: acquiring an exclusive lock on the most recent event; constructing a canonical string by concatenating with pipe delimiters the previous event's hash, the new event's identifier, type, source, conversation identifier, timestamp as nanoseconds, and JSON-serialised content; computing the SHA-256 hash of said canonical string; and storing the event with both its computed hash and the previous event's hash.

6. The system of claim 1, wherein the tick engine enforces a layer activation constraint such that primitives at Layer N are eligible for processing only when primitives at Layer N-1 have reached a stable active state.

7. The system of claim 1, wherein the event graph store supports causal graph traversal by implementing recursive queries that, given an event identifier, retrieve all ancestor events by recursively following the causal predecessor references, and all descendant events by recursively identifying events that reference the given event as a cause.

Claims for the Communication Protocol

1. A method of communication between distributed cognitive primitives in a software system, comprising:

    1. defining a four-event vocabulary consisting of MessageSent, MessageReceived, Decision, and Action event types, each event carrying causal predecessor references and a conversation identifier;
    2. for each primitive, providing a listen interface configured to receive a message and return a set of actions, and a say interface configured to send a directed message to another primitive, wherein each message is recorded as an event on a hash-chained event graph;
    3. providing a gateway primitive configured to receive inbound messages and semantically analyse the content to determine relevant domain primitives, and route the message to said relevant primitives rather than broadcasting to all primitives;
    4. maintaining for each primitive a three-layer knowledge architecture comprising: persistent conversational context across invocations; a key-value memory store for learned facts; and structural change events providing awareness of system topology modifications; and

    5. enabling each primitive to invoke any of a set of universal capabilities including: event graph queries, memory operations, state operations, adjacency queries, and event emission.

2. The method of claim 8, wherein inter-primitive conversations occur independently of external input, such that a primitive observing a condition may initiate a conversation with related primitives, with all messages recorded as causally-linked events on the hash-chained event graph.

3. The method of claim 8, wherein primitives span a continuum from mechanical to intelligent, wherein mechanical primitives operate primarily through deterministic processing branches with fallthrough to language model reasoning for exceptional situations, and intelligent primitives operate primarily through language model reasoning with progressively growing deterministic branches for recognised patterns.

Claims for the Inter-System Communication Protocol

1. A method of communication between a plurality of autonomous systems each maintaining an independent hash-chained event graph, comprising:

    1. providing each system with a cryptographic keypair, the public key constituting the system's identity, wherein no central registry is required for identity management;
    2. defining a cross-graph event reference (CGER) data structure comprising a system identifier derived from the system's public key, an event identifier, an event hash, an event type, and a timestamp, said CGER enabling an event in one system's event graph to declare as a causal predecessor an event residing in a different system's event graph;
    3. wrapping each inter-system message in a signed envelope comprising: sender and recipient system identifiers, a unique message identifier, a CGER identifying the causing event in the sender's graph, a chain head snapshot of the sender's event graph, and a cryptographic signature computed over a canonical representation of the envelope and payload;
    4. upon receipt of a message, the receiving system verifying the envelope signature, recording the message as an event in its own event graph with the CGER as a causal predecessor, and routing the message content to relevant cognitive primitives; and

5. whereby each inter-system message is independently recorded in both the sender's and receiver's event graphs with cross-graph causal references linking them, creating a bilateral audit trail that is independently verifiable by either system.

2. The method of claim 11, further comprising an integrity proof exchange mechanism wherein:

    1. a first system sends a proof request to a second system specifying a proof type selected from: chain segment proof (a sequence of event identifiers, hashes, and previous hashes between two specified events), event existence proof (a single event with its chain neighbours), and chain summary proof (a sparse sequence of chain entries);
    2. the second system extracts the requested proof data from its event graph and returns it in a signed response;
    3. the first system independently recomputes hashes from the proof data and verifies chain linkage; and
    4. successful verification increases the first system's trust in the second system, while failed verification decreases trust and may trigger treaty revocation.

3. The method of claim 11, further comprising a treaty-based federated governance mechanism wherein:

    1. a first system proposes treaty terms to a second system, said terms specifying: shared capabilities, authority rules for cross-system actions, data sharing boundaries with field-level redaction controls, trust parameters, and rate limits;
    2. the second system evaluates and accepts, rejects, or counter-proposes the terms;
    3. accepted treaties are recorded as signed events in both systems' event graphs;
    4. cross-system actions governed by bilateral authority rules require approval from both systems' authority subsystems before execution; and
    5. either system may revoke a treaty at any time, with the revocation recorded in both event graphs.

4. The method of claim 11, further comprising a trust accumulation model wherein:

    1. each system maintains a continuous trust score (0.0 to 1.0) for each system it has interacted with;
    2. trust starts at a low initial value and increases through successful message exchanges and verified integrity proofs;

3. trust decreases upon violations including integrity proof failures, unfulfilled obligations, and message timeouts;
4. trust decays toward zero over time without interaction;
5. trust is asymmetric such that each system independently maintains its own trust assessment; and
6. trust is non-transitive such that a first system's trust in a second system does not confer trust in systems trusted by the second system.

5. The method of claim 11, wherein multi-hop communication across more than two systems is achieved by: each forwarding system creating a new signed envelope with the cause CGER referencing the event in the forwarding system's own graph; the causal chain being preserved across all hops such that the final recipient can trace back through each forwarding system's graph to the originator by following CGERs; and each system in the chain being independently accountable for its forwarding decisions.

6. The method of claim 11, wherein the envelope includes a chain head snapshot comprising the sender's current event count, latest event hash, latest event identifier, and latest timestamp, and the receiving system monitors the sender's chain head over successive messages to detect chain regression, wherein a decrease in event count or disappearance of a previously-observed hash constitutes a detectable integrity violation.

Claims for the Ontology Derivation Method

1. A method for deriving a cognitive ontology for an autonomous agent system, comprising:

   1. defining a foundation layer of computational primitives representing irreducible cognitive operations including event registration, causal linking, identity management, trust assessment, confidence evaluation, and integrity verification;
   2. for each successive layer above the foundation, identifying a functional gap — a capability that the layer below cannot express but is structurally necessary for higher-order cognition;
   3. deriving primitives to fill said functional gap, organised into groups of related primitives;
   4. enforcing a layer activation constraint such that primitives at a given layer activate only when the layer below is stable; and
   5. recognising irreducible elements that cannot be derived from lower layers as foundational recognitions rather than emergent properties.

2. The method of claim 17, further comprising validating the derived ontology through independent derivation from a complementary starting point, and confirming convergence between the two derivations on structural and qualitative conclusions.

3. The method of claim 17, wherein the layers comprise, in order: Foundation (irreducible operations), Agency (observer to participant), Exchange (individual to dyad), Society (dyad to group), Legal (informal to formal), Technology (governing to building), Information (physical to symbolic), Ethics (descriptive to normative), Identity (doing to being), Relationship (self to self-with-other), Community (relationship to belonging), Culture (living to examining), Emergence (content to architecture), and Existence (everything to the fact of everything).

4. The method of claim 17, wherein the derived ontology forms a strange loop in which the highest layer (Existence — the fact that anything exists) is presupposed by the lowest layer (Foundation — the occurrence of events).

ABSTRACT

An autonomous cognitive agent system comprises: an event graph store maintaining a hash-chained, append-only, causally-linked event log; a plurality of cognitive primitives organised in a hierarchical ontology across fourteen layers, each primitive having an activation level, a lifecycle state machine, a key-value state store, and a processing function; a tick engine that propagates events through primitives in ripple waves until quiescence; a decision tree engine enabling progressive migration from language-model-based reasoning to deterministic rules; an authority system for three-tier approval of significant actions; an intra-system communication protocol built on a four-event vocabulary with semantic gateway routing and three-layer knowledge architecture; and an inter-system communication protocol (EventGraph Interchange Protocol) enabling sovereign systems with independent event graphs to communicate via signed envelopes carrying cross-graph event references for causal linking across graph boundaries, integrity proof exchange for verifying foreign graph integrity, treaty-based federated governance for bilateral authority, and trust accumulation through observed interaction history. A method for deriving the cognitive ontology by iteratively identifying

functional gaps between layers produces a fourteen-layer framework from Foundation through Existence, validated by convergent independent derivation from complementary starting points.