Architecture for a Scalable, Dual-Hemisphere Artificial General Intelligence (AGI) System

Andrew Bond

Department of Computer Engineering,

San José State University,

San José, CA, USA

Email: andrew.bond@sjsu.edu

Contents

Architecture for a Scalable, Dual-Hemisphere Artificial General Intelligence (AGI) System 1

I. Executive Summary 3

II. Problem Framing 4

III. Biological Inspiration 7

IV. Dual-Hemisphere Baseline Design 11

V. Communication and Coordination Layer 17

VI. Memory Architecture 22

VII. Safety Architecture 30

VIII. Metacognition Layer 38

IX. Lifelong Learning System 45

X. Virtual Embodiment Environment 51

XI. Sensorimotor Loop Architecture 56

XII. HPC Deployment Architecture 62

XIII. Networking & RDMA Fabric 68

XIV. API Interfaces & Versioning 73

XV. Implementation Roadmap 79

XVI. Evaluation & Testing Framework 90

XX. Appendices & Technical Specifications 102

Appendix A — API Schema Definitions 102

Appendix B — Data Formats 118

Appendix C — Event Fabric & Topic Schemas 131

Appendix D — SLURM Job Configurations 137

Appendix E — Security Considerations 144

Appendix F — Failure Modes & Recovery Strategies 149

Appendix G — Operational Guidelines 157

Appendix H — Future Research Directions 160

I. Executive Summary

This whitepaper presents a concrete, engineering-focused architecture for a scalable, dual-hemisphere Artificial General Intelligence (AGI) system deployed on high-performance computing (HPC) infrastructure and grounded in a real-time virtual embodied environment running on consumer-grade hardware. The design aims to bridge the gap between today's monolithic, single-node foundation models and a distributed cognitive system capable of long-horizon reasoning, continual learning, and safe real-world interaction.

The architecture is driven by four primary objectives:

1. **Architectural scalability** – enabling the system to grow from an initial dual-node prototype into a polycentric cognitive substrate composed of many specialized agents, without redesigning the core communication, memory, or safety mechanisms.
2. **Biologically inspired but engineering-realistic cognition** – adopting functional specialization (left/right hemispheres), distributed and asynchronous processing, sensorimotor grounding, and layered memory without resorting to literal neurobiological mimicry.
3. **Operational robustness and safety** – embedding safety, metacognition, and recoverability into the core execution loop, with explicit mechanisms for introspection, error correction, and bounded cognitive resources.
4. **Practical deployability on HPC clusters** – aligning with real-world constraints such as SLURM scheduling, Singularity/Apptainer containers, UCX/InfiniBand fabrics, and multi-user security policies common in research and industrial HPC environments.

The system comprises four primary components:

- **Left Hemisphere (LH, H100 GPU node):** Hosts large language–based reasoning and planning models, a task interpreter and goal extractor, a hierarchical planning engine, a metacognition layer, and a multi-layer safety supervisor. The LH is responsible for goal formation, plan generation, tool selection, and self-evaluation of its own reasoning under explicit time and resource budgets.
- **Right Hemisphere (RH, A100 GPU node):** Hosts multimodal perception, world-model simulation, trajectory planning, and physics-aware prediction modules. The RH provides grounded scene understanding, evaluates candidate actions via simulation, and enforces physics-based safety constraints in real or virtual environments.
- **Distributed Memory Substrate (CPU nodes / DHT layer):** Implements semantic, episodic, and procedural memory across one or more CPU nodes. A vector database backs semantic memory; episodic logs capture rich interaction histories; procedural memory stores skills and reusable workflows. Memory growth, consolidation, and pruning are explicitly managed to support long-lived agents.

- **Virtual Embodied Environment (Unity/MuJoCo on gaming laptop):** Provides a physics-consistent world in which the agent perceives, acts, and learns over continuous time. Sensor streams (RGB-D, metadata, proprioception) flow to the RH, while actuator commands are gated by safety checks before affecting the environment.

Cognition is coordinated through an **event-driven fabric** in the baseline system, and can be upgraded to an optional **Distributed Hash Table (DHT) coordination layer** that turns the architecture into a genuinely polycentric cognitive network. The event fabric supports low-latency, asynchronous publish/subscribe communication (e.g., perception.frame_ready, plan.step_ready, safety.flag), while the DHT generalizes this into a shared, fault-tolerant state space where cognitive artifacts (plans, embeddings, scene graphs, skills) are stored under hashed keys and consumed by whichever subsystem is best positioned to act on them.

A dedicated **metacognition layer** monitors reasoning quality, confidence, and cross-hemisphere consistency. It evaluates chains of thought, detects contradictions between symbolic plans and physical predictions, and triggers reflection or replanning when confidence is low. To avoid unbounded introspection, metacognition operates under explicit time and compute budgets and participates in a simple "cognitive economics" scheme that arbitrates scarce resources between competing tasks and subsystems.

Safety is implemented as a **three-layer pipeline**: pre-action static filtering of plans and tool invocations; in-action monitoring via physics-based prediction, action firewalls, and runtime constraints; and post-action verification with feedback into episodic and procedural memory. This safety stack is tightly integrated with both hemispheres and the embodiment layer, rather than added as an external wrapper.

The deployment model targets modern HPC clusters: each hemisphere and memory subsystem runs as a long-lived SLURM job inside a Singularity container, communicating over UCX/InfiniBand for microsecond-scale inter-node latency. The virtual environment is isolated on a gaming laptop to decouple simulation performance and crash risk from core cognition. A phased roadmap—from dual-hemisphere prototype to multi-agent, DHT-coordinated polycentric AGI—is provided to support incremental, testable progress over several years.

Overall, this architecture offers a buildable path from today's large-model capabilities toward a distributed, safety-aware, embodied AGI system that can be incrementally realized and rigorously evaluated in real HPC environments.

II. Problem Framing

Despite their impressive capabilities, today's large-scale AI systems remain fundamentally constrained by **architectural monolithism** and **episodic, single-context interaction patterns**. Large Language Models (LLMs) and multimodal foundation models excel at pattern recognition, code generation, and contextual reasoning within a single inference window, but they do not by

themselves constitute a general cognitive system. They typically operate as stateless services: given a prompt and a fixed context, they produce an output and then "forget" nearly everything.

This leads to several critical limitations:

- **No persistent, structured world model.** Most deployments rely on shallow, prompt-scoped context rather than a maintained, queryable representation of the environment and its dynamics.
- **Weak long-term memory.** Knowledge is encoded in the model weights and in transient context windows, but not in a persistent, inspectable, and updatable memory substrate.
- **Limited causal grounding.** Systems manipulate symbols and embeddings but rarely act in or learn from continuous, physics-consistent environments with real feedback loops.
- **Lack of autonomous goal management.** Present systems are typically "prompt-followers," not agents with ongoing goals, self-monitoring, and long-horizon plans.
- **Coupled resources and responsibilities.** In many architectures, a single GPU host is responsible for inference, short-term memory, tool use, and often even simulation, creating tight coupling and severe bottlenecks under load.

At the system level, this monolithic paradigm makes it difficult to realize key properties of general intelligence:

- **Parallel, distributed cognition.** In biological systems, many cognitive processes run concurrently and asynchronously, influencing one another without a single global "loop."
- **Persistent identity and continuity of experience.** An intelligent agent needs to accumulate and refine experiences over days, months, or years, not per API call.
- **Embodied interaction and causal learning.** Intelligence emerges from acting in the world (real or simulated), observing consequences, and refining internal models accordingly.
- **Flexible integration of specialized subsystems.** Vision, language, planning, motor control, safety checks, and metacognition must all interoperate without being forced into a single model or node.

Current AI deployments often place *all* of these responsibilities on a single node or container: the same process hosts the model, performs inference, maintains a scratchpad, calls tools, interacts with external systems, and sometimes even runs lightweight simulations. This "all-in-one" pattern creates several practical problems:

- **Scalability limits.** Scaling to larger tasks or longer horizons means scaling a single node vertically, which hits GPU memory and compute ceilings quickly.

- **Performance bottlenecks.** High-frequency perception, simulation, and planning work contend for the same GPU and CPU resources, degrading latency and throughput.
- **Operational fragility.** A crash, misconfiguration, or overload in one component can bring down the entire cognitive loop.
- **Inflexible evolution.** Swapping out or upgrading one capability (e.g., a world model) often requires disruptive changes to the entire stack.

From a cognitive standpoint, this architecture encourages **linear, tool-call-centric workflows** rather than **rich, asynchronous cognitive dynamics**. The agent is forced into a single-threaded pattern: read prompt → think once → act → stop. There is no natural place for ongoing background processes such as:

- continuous perceptual monitoring,
- proactive hypothesis testing in a world model,
- scheduled reflection and consolidation of past episodes, or
- ongoing safety and consistency checks spanning many tasks and time scales.

The core thesis of this work is that **general intelligence is inherently distributed**, both functionally and physically. In a capable AGI system:

- Many subsystems (language reasoning, perception, planning, safety, metacognition, memory) run on **different nodes** and **different time scales**, yet must remain tightly coordinated.
- Memory is not just the model's weights; it is a **structured, multi-layer substrate**—semantic, episodic, and procedural—that can be queried, updated, summarized, and pruned as the agent's life unfolds.
- Perception and action form a **continuous sensorimotor loop** in a physics-consistent environment, not sporadic, text-only tool calls.
- Safety and self-monitoring must be woven into every layer of the stack, not implemented as a thin filter around a monolithic model.

The engineering challenge, therefore, is **not** to build "one bigger model," but to design a cohesive **cognitive architecture** that:

1. **Decomposes cognition into specialized subsystems** (e.g., left/right hemispheres, memory nodes, embodiment engine) with clear responsibilities.
2. **Runs these subsystems on an HPC cluster**, using GPU and CPU nodes as a flexible substrate for parallel cognition rather than a single-host optimization target.
3. **Provides low-latency communication and coordination** via an event fabric and, in the advanced configuration, a Distributed Hash Table (DHT)–based cognitive substrate.

4. **Supports persistent, structured memory** that outlives any single process, job, or experiment.
5. **Integrates virtual embodiment** as a first-class citizen to provide causal grounding, curriculum-based learning, and real-time evaluation in a safe sandbox.
6. **Embeds safety and metacognition** directly into the planning, simulation, and execution loops, enabling introspection, error detection, and corrective behavior.

The dual-hemisphere, HPC-first architecture proposed in this whitepaper addresses these requirements by:

- Splitting cognition into a **Left Hemisphere** for language-based reasoning, planning, and metacognition, and a **Right Hemisphere** for perceptual understanding, world-modeling, and trajectory planning.
- Introducing a **distributed memory substrate** spanning semantic, episodic, and procedural stores, hosted on dedicated CPU nodes.
- Anchoring the system in a **virtual embodied environment** running on a consumer-grade gaming laptop, decoupled from HPC cognition but tightly coupled at the API level.
- Coordinating everything through an **event-driven fabric** that can be upgraded to a **DHT-based coordination layer** for polycentric, fault-tolerant cognition at scale.

In short, the problem this architecture addresses is the mismatch between **what current monolithic AI systems are structurally able to do** and **what general intelligence requires**: distributed, persistent, embodied, safety-aware cognition operating over long horizons in a real, resource-constrained HPC environment.

III. Biological Inspiration

This architecture is *inspired* by biological brains, but it does not attempt a literal neuroanatomical replica. Real hemispheric specialization is continuous, redundant, and shaped by development and learning rather than cleanly partitioned modules. Instead of copying biology, we treat it as a library of design patterns and constraints that inform how we structure distributed cognition on an HPC cluster.

A. Hemispheric Specialization as an Engineering Pattern

In biological brains, the left and right hemispheres exhibit *tendencies* (e.g., language vs. spatial reasoning), not hard boundaries. There is heavy cross-talk via the corpus callosum, context-dependent role allocation, and plasticity over time. Translating this into an engineering design:

- The **Left Hemisphere (LH)** node is *biased* toward:
    - language-based reasoning and planning
    - goal and task graph construction

    - metacognition and safety supervision
    - structured interaction with long-term memory
- The **Right Hemisphere (RH)** node is *biased* toward:
    - perception, spatial representation, and multimodal encoding
    - world-model simulation and physics prediction
    - trajectory planning and low-level control policies

These roles are not absolutely exclusive. Both hemispheres can read/write shared memory and request services from each other; the division exists to encourage **functional specialization** and to map cleanly onto distinct GPU nodes rather than to enforce a rigid cognitive ontology.

Cross-hemisphere communication is implemented by the **event fabric** and, in the advanced configuration, by the **DHT-based coordination layer**, which together function as an artificial corpus callosum. Events such as plan.step_ready, simulation.result, safety.flag, and confidence.low are continuously exchanged with microsecond-scale latency over UCX/InfiniBand.

B. Disagreement and Conflict Resolution Between Hemispheres

In biological systems, hemispheres frequently disagree; what matters is how conflicts are resolved, not that they never occur. The architecture makes this explicit and machine-checkable.

1. **Physics vs. Plan Conflicts**
    - The LH proposes a candidate plan or action sequence.
    - The RH's world model simulates this plan and returns:
        - predicted trajectories,
        - failure probabilities, and
        - a **risk score** per step.

If the RH flags trajectory.risk_high or violation.predicted, this is treated as a **hard veto** on that plan segment by the safety subsystem. The RH is considered authoritative on physical feasibility.

- The LH is required to **replan under new constraints** ("do not move object X above height H", "avoid path through region Y", etc.), and the original plan is annotated in episodic memory as "physics-inconsistent" for future learning.

Perception vs. Semantic Priors Conflicts
When RH perception contradicts LH or memory expectations (e.g., object not where LH predicted):

- RH publishes a perception.state_mismatch event with evidence (observed vs. expected state).
- A metacognitive routine evaluates three possible culprits:
    - outdated semantic memory,

    - world-model drift,
    - or flawed LH assumptions.

- The default priority order is:
  current perception > world-model prediction > stored semantic priors.
- Semantic memory entries that are repeatedly contradicted are down-weighted or revised during consolidation.

1. **Symbolic vs. Embodied Goal Conflicts**
   If LH's abstract goal conflicts with embodiment constraints (e.g., "stack blocks higher than allowed by scenario rules"):
    - Safety and RH jointly raise goal.infeasible and attach explanation metadata.
    - LH must either (a) reformulate the goal within constraints or (b) escalate to a human operator in supervised modes.

Taken together, these rules define a **conflict resolution protocol**: RH and safety are authoritative over physical feasibility and constraint adherence; LH must adapt its symbolic plans accordingly.

C. Attention, Priority, and Resource Management

Brains do not allocate attention uniformly; neither should this architecture. We introduce a simple **cognitive economics** scheme to arbitrate limited resources (GPU time, memory bandwidth, world-model rollouts) across tasks and hemispheres.

1. **Task Tickets and Priority Scores**
   Each active cognitive task (e.g., "navigate to waypoint", "analyze failure", "learn new skill") is represented as a ticket with:
    - priority class (safety-critical, task-critical, background),
    - expected compute cost,
    - deadline or soft time budget,
    - expected value (e.g., success probability × reward).
2. **Shared Scheduler Across Hemispheres**
   A lightweight scheduler, implemented either as a dedicated microservice or as a DHT-mediated protocol, periodically:
    - aggregates task tickets from LH, RH, and memory subsystems;
    - computes a scheduling decision using heuristics (e.g., safety-critical > ongoing control > background reflection);
    - throttles or defers low-value tasks (e.g., curiosity-driven simulation) when resources are scarce.
3. **Local Attention Within Each Hemisphere**
   Each hemisphere also maintains its own internal attention mechanisms:

- LH prioritizes plan steps that are imminent in the action queue or flagged by safety/metacognition.
- RH prioritizes perception frames and simulations that impact near-future actions versus long-horizon exploration.

This yields a dual-layer attention system: local (within each hemisphere) and global (via cognitive economics) to prevent both starvation and runaway background processes.

D. Convergence Criteria for Planning and Reflection

In biological cognition, deliberation is not infinite; organisms act under time pressure and incomplete information. To mirror this, planning and metacognitive loops are governed by explicit **convergence criteria**:

1. **Plan–Simulation Loop Convergence**
   The LH↔RH loop for a given plan terminates when *any* of the following holds:
   - **Feasibility**: RH's risk scores for all remaining steps fall below a configured threshold, and no new trajectory.risk_high or constraint.violation events are observed within a time window.
   - **Budget Exhaustion**: Time or compute budget for this decision cycle is reached; the system selects the best currently known safe plan, or defaults to a safe fallback policy (e.g., stop motion, retreat to safe pose).
   - **External Interrupt**: A higher-priority safety or operator interrupt preempts the loop.
2. **Metacognitive Reflection Convergence**
   Metacognitive analysis (self-checking LH's reasoning, cross-checking with RH and memory) terminates when:
   - no new inconsistencies are detected in a fixed number of reflection passes;
   - confidence metrics (based on historical success, memory agreement, and world-model agreement) exceed a threshold;
   - or the reflection time budget is consumed, at which point the system acts with explicit low-confidence tags stored in episodic memory.

These convergence rules ensure that the system **acts in finite time** while maintaining explicit, inspectable records of how confident it was and what tradeoffs were made.

E. Distributed, Non-Linear Cognition and Embodied Interaction

Finally, two broader biological inspirations shape the overall architecture:

1. **Distributed, Non-Linear Cognition**
   Brains run many partially independent processes; there is no single global loop. The architecture mirrors this via:
   - multiple long-lived services (LH, RH, memory nodes) on different HPC nodes;
   - an event-driven fabric where perception, planning, safety, and metacognition publish and react to events asynchronously;

- an optional DHT layer that turns cognitive state into a shared, decentralized substrate.

1. **Embodied Interaction**
   Intelligence emerges in the loop between perception and action. The virtual environment on the gaming laptop provides:
   - continuous sensor streams (RGB-D, object metadata, proprioception),
   - a rich action space (navigation, manipulation, tool use),
   - and curriculum-based tasks for learning.

Embodiment is not merely illustrative; it is the primary source of *ground truth* against which world models, semantic priors, and plans are continuously tested and corrected.

IV. Dual-Hemisphere Baseline Design

The dual-hemisphere baseline is the first concrete instantiation of the architecture: two specialized GPU nodes (Left and Right Hemisphere) running as long-lived services on an HPC cluster, connected to a shared memory substrate and a virtual embodied environment. The goal is not to perfectly mimic biological hemispheres, but to create a **useful division of labor** that maps cleanly onto heterogeneous hardware and supports explicit conflict-resolution and resource-allocation policies.

Each hemisphere is responsible for a distinct slice of cognition:

- The **Left Hemisphere (LH)** is biased toward language-based reasoning, goal and plan construction, safety and metacognitive evaluation, and structured interaction with memory and tools.
- The **Right Hemisphere (RH)** is biased toward perception, spatial and multimodal representation, world-model simulation, trajectory planning, and low-level control in the virtual environment.

Both hemispheres are deployed as Singularity/Apptainer containers on dedicated GPU nodes and communicate over UCX/InfiniBand via an event fabric (and optionally a DHT layer) that acts as a software corpus callosum.

A. Left Hemisphere Node (H100 GPU)

The Left Hemisphere serves as the system's **symbolic and deliberative core**. It concentrates capabilities that benefit most from large language models and structured reasoning:

1. **Language-Based Reasoning Model**

- A high-capacity transformer model (e.g., LLaMA-class, Qwen-class, or future architecture) runs on an H100 GPU with KV cache–optimized inference (e.g., vLLM or similar).
- The model is tuned for multi-step reasoning, tool selection, and plan generation, with explicit support for chain-of-thought, reflection prompts, and safety-aware instruction following.

Task Interpreter & Goal Manager

- Converts user instructions, environment events, and internal drives into **task graphs** with explicit goals, constraints, and success criteria.
- Interfaces with a **goal manager** that maintains an agenda of active tasks, each with priority, deadline, and expected value.
- Integrates with the cognitive economics layer: every new task is registered as a "ticket" with estimated resource cost and importance.

Hierarchical Planning Engine

- Produces multi-level plans: mission → subgoals → actionable steps → environment or tool commands.
- Plans are represented as explicit graphs (nodes = steps; edges = dependencies, contingencies), with annotations for required world-model checks and safety gates.
- For each step that involves environment change, the LH issues a **simulation request** to the RH and awaits a feasibility and risk evaluation before committing.

Safety Supervisor (Pre-Action Layer)

- Performs pre-action analysis of plans and tool calls:
  - validates syntax and schema,
  - checks against whitelists/blacklists of tools,
  - screens for obviously unsafe or out-of-scope actions,
  - detects likely hallucinations (nonexistent tools, impossible device capabilities).
- Enforces hard constraints (e.g., "never modify host filesystem", "never exceed torque limits") **before** simulation or execution.

Metacognition & Confidence Estimation

- Runs reflective passes over generated reasoning and plans to:
  - detect logical inconsistencies or unjustified leaps,
  - cross-check against semantic and episodic memory,

    - incorporate RH feedback (e.g., “physics says this is impossible”).
- Produces a **confidence vector** (semantic agreement, physics agreement, historical success rate) that is attached to each plan and stored in episodic memory.
- Operates under explicit time and compute budgets enforced by the cognitive economics layer; when budgets are exhausted, it returns the best available plan with an explicit “low-confidence” flag.

Event and Memory Interfaces

- Subscribes to events such as perception.state_mismatch, trajectory.risk_high, safety.flag, and goal.infeasible.
- Issues events like plan.step_ready, plan.revise_required, and reflection.completed.
- Performs semantic queries (vector search), episodic lookups, and procedural skill retrieval through the memory API, typically hosted on CPU nodes.

Operationally, the LH acts as a **strategist and supervisor**: it creates and revises plans, requests simulations, arbitrates between tasks, and ensures that actions respect high-level constraints and safety policies.

B. Right Hemisphere Node (A100 GPU)

The Right Hemisphere serves as the system’s **perceptual and world-modeling core**, responsible for interpreting the virtual environment, predicting consequences of actions, and generating safe low-level control trajectories.

Perception & Multimodal Encoding

    - Consumes sensor streams from the virtual environment: RGB frames, depth maps, segmentation masks, object metadata, proprioception, and collision events.

Runs a vision/multimodal encoder (e.g., CLIP/DINO-class model) on the A100 GPU to produce:

    - scene embeddings,
    - object-centric representations (location, attributes, affordances),
    - keypoint and segmentation maps.
- Publishes perception.frame_ready and perception.state_update events with summarized state and references to stored raw data in episodic memory.

World-Model Simulator & Physics Predictor

Implements one or both of:

    - a parametric world model (e.g., video transformer or dynamics model), and/or
    - a physics engine–backed simulator (MuJoCo, Unity physics) running locally or via a tightly coupled service.

Given candidate action sequences from LH, performs short-horizon rollouts to estimate:

    - trajectories,
    - contact events,
    - constraint violations,
    - failure probabilities.

- Returns a structured simulation.result object with per-step risk metrics; if any step exceeds a hazard threshold, emits trajectory.risk_high and a veto recommendation.

Perceptual State Tracker

- Maintains a continuously updated state representation of the environment:
    - object positions and identities,
    - relationships (supported-by, inside-of, near-to),
    - agent pose and joint states,
    - active constraints and goals relevant to the current scene.
- Detects discrepancies between predicted and observed states (perception.state_mismatch) and routes them to LH and metacognition for analysis and memory updates.

Trajectory Planning & Low-Level Control

- Given a validated, high-level plan step (e.g., “move object A to location B”), computes feasible trajectories for the agent’s body or end effector.
- Enforces kinematic and dynamic constraints (joint limits, velocities, collision-free paths) and uses the world model to test and refine trajectories before execution.
- Sends actuator commands to the virtual environment via a safety-checked control API; publishes action.executed and, in case of issues, control.error events.

Safety (In-Action Layer)

- Implements runtime safety guards close to the control loop:
    - monitors predicted vs. actual contacts,
    - enforces speed/force bounds,
    - halts motion when unexpected high-risk states are detected.

- Can issue **hard interrupts** (e.g., emergency_stop) that preempt LH plans and return the system to a safe configuration.

Operationally, the RH acts as the **physicist and operator**: it tells the system what the environment looks like, which actions are physically feasible, and how to execute them safely and efficiently.

C. Cross-Hemisphere Coordination and Conflict Resolution

The two hemispheres are loosely coupled yet tightly coordinated:

1. **Event Fabric / Artificial Corpus Callosum**
    - Both LH and RH publish and subscribe to a shared event bus over UCX/InfiniBand.
    - Key event types include:
        - LH → RH: plan.step_ready, simulation.request, control.request
        - RH → LH: simulation.result, trajectory.risk_high, perception.state_mismatch
        - Both directions: safety.flag, goal.infeasible, confidence.low

Conflict Resolution Protocol

- **Physics vs. Plan:** RH and safety are authoritative on physical feasibility and constraint adherence. If RH returns high risk or violation predictions, that segment of the plan is vetoed; LH must replan under the new constraints.
- **Perception vs. Memory:** When perception disagrees with semantic priors, perception wins in the short term; semantic memory is updated or down-weighted during consolidation.
- **Goal vs. Scenario Constraints:** If a goal cannot be satisfied without violating environment or safety constraints, a goal.infeasible event triggers goal reformulation or escalation to a human supervisor.

Cognitive Economics and Resource Arbitration

- Both hemispheres submit task tickets to a shared scheduler: simulation jobs, planning tasks, reflection passes, and background learning.
- The scheduler allocates GPU time and bandwidth by priority (safety > active control > near-term planning > background reflection), preventing either node from being monopolized by low-importance work.

D. Execution Loop and Convergence

A typical perception–plan–simulate–act loop proceeds as follows:

1. RH ingests sensor data from the virtual environment and publishes an updated perceptual state.
2. LH retrieves relevant memory, interprets goals, and generates candidate plans.
3. For each plan segment that affects the environment, LH requests simulations from RH.
4. RH evaluates candidate actions, returns predicted trajectories and risk scores, and may veto unsafe segments.
5. LH revises the plan if necessary; metacognition evaluates reasoning quality and cross-checks with memory and simulation.
6. Once convergence criteria are met (risk below threshold, budgets respected, no new conflicts), RH executes the agreed-upon control commands.
7. Episodic and procedural memory are updated with outcomes, errors, and any newly learned skills.

Convergence is guaranteed by explicit **time/compute budgets** for planning, simulation, and reflection. When budgets are exhausted, the system either:

- selects the best available safe plan, or
- defaults to a conservative policy (e.g., “stop, maintain safe pose, request human input”).

E. Role in the Overall Architecture

The dual-hemisphere baseline is intentionally minimal: a small number of well-defined, high-capability services that can be deployed on existing HPC clusters. It establishes:

- clear **interfaces** (APIs and events) between symbolic reasoning and embodied physics,
- a **conflict resolution** regime between hemispheres,
- a **resource arbitration** mechanism via cognitive economics, and
- a **finite, inspectable decision process** via explicit convergence criteria.

Later phases of the roadmap extend this baseline by adding more specialized micro-agents, expanding the DHT substrate, and enriching the virtual world, but the Left/Right Hemisphere pair remains the central organizing scaffold for distributed cognition.

V. Communication and Coordination Layer

V. Communication and Coordination Layer

The communication and coordination layer is the system's "artificial nervous system." It must support:

- low-latency, high-throughput messaging between hemispheres and memory,
- flexible scaling from a small number of services to dozens of micro-agents, and
- robust conflict resolution and safety signaling across the whole cognitive graph.

To achieve this, the architecture defines **two complementary coordination mechanisms**:

1. an **Event-Driven Fabric** (baseline, simple, low overhead), and
2. an optional **DHT-Based Coordination Layer** (advanced, scalable, fault-tolerant).

These are not mutually exclusive. The event fabric is the default for fine-grained, low-latency messaging; the DHT is introduced when you need shared, persistent cognitive state across many nodes.

A. Event-Driven Fabric (Baseline Coordination Mode)

The **event fabric** is a high-speed publish/subscribe message bus running over the HPC interconnect (UCX/InfiniBand). It acts as the system's "fast path" for real-time coordination.

Primary use cases

- Cross-hemisphere communication (LH ↔ RH) for:
    - plan.step_ready, simulation.request, simulation.result,
    - trajectory.risk_high, perception.state_update, safety.flag.
- Notifications to and from memory subsystems (memory.updated, episode.appended).
- Short-lived control signals (e.g., emergency_stop, goal.infeasible).

Message ordering guarantees

- **Per-topic FIFO**:
    - For a given topic and publisher, the fabric guarantees in-order delivery: messages from the same publisher on the same topic are delivered to subscribers in the order sent.
- **No global total ordering**:
    - Across different topics and publishers, no strict global ordering is enforced. If global causal relationships matter, components attach **monotonic sequence numbers** and **trace IDs** so receivers can reconstruct partial orders where needed.

Preventing event storms and circular dependencies

To prevent pathological behavior, the event fabric enforces several policies:

1. **Subscription Scoping**
    - Topics are organized hierarchically (e.g., perception.*, plan.*, safety.*).
    - Subsystems only subscribe to the specific topics they require, reducing fan-out.
2. **Rate Limiting and Backpressure**
    - Each subscriber can advertise a maximum accepted throughput.
    - If a publisher exceeds this, messages are either dropped (for low-priority topics) or aggregated into summaries (for high-priority topics like perception).
    - The fabric exposes metrics to the cognitive economics layer, which can throttle low-value tasks when congestion appears.
3. **Coalescing and Deduplication**
    - Repeated events of the same type and key (e.g., many perception.state_update events for minor changes) can be coalesced into a single latest-state update.
4. **Loop Detection and Hop Limits**
    - Every event carries a **trace ID** and a **hop counter**.
    - If an event's trace ID appears too frequently in a short time window, or the hop counter exceeds a threshold, the system flags a potential circular dependency and either:
        - drops further events in that trace, and/or
        - raises a coordination.loop_detected diagnostic event.

Together, these mechanisms keep the event fabric responsive, bounded, and debuggable, even as more subsystems are added.

When to choose the Event Fabric

Use the event fabric alone when:

- The system is in early phases (dual-hemisphere + memory + single embodiment).
- Most communication is **ephemeral** (notifications, triggers, control signals).
- You need **microsecond-scale latency** and want to minimize coordination overhead.

B. DHT-Based Coordination Layer (Advanced Mode)

The **Distributed Hash Table (DHT) coordination layer** is an optional, higher-level mechanism that turns distributed state into a shared, queryable substrate—effectively, a **cognitive blackboard** spread across many nodes.

Instead of passing all information around as transient events, subsystems can:

- write cognitive artifacts (plans, embeddings, scene graphs, skills, summaries) to the DHT under hashed keys;
- subscribe to changes in certain key ranges or namespaces;
- rely on the DHT for **state discovery, replication, and fault tolerance**.

Primary use cases

- Polycentric cognition with many micro-agents (future phases).
- Sharing **longer-lived, structured state** among multiple subsystems:
    - plan.graph.current,
    - scene.graph.current,
    - skill.library.*,
    - episodic.index.*.
- Decoupling producers and consumers of cognitive state (e.g., a learning daemon that ingests episodic logs and writes updated procedural skills).

C. DHT Consistency Model and Semantics

The DHT is designed around **eventual consistency with causal hints**, which is appropriate for cognitive workloads that tolerate short-lived inconsistencies but need durability and convergence.

Consistency model

- **Eventual consistency**:
    - Updates to a key are propagated to responsible peers and replicas; after a bounded time, all healthy nodes converge to the same value (modulo new writes).
- **Per-key causal metadata**:
    - Each value is stored with a **version vector or Lamport timestamp**.
    - When concurrent updates occur, conflict resolution rules are applied (see below).

Conflict resolution

- **Type-specific policies**:
    - For additive logs (e.g., episodic entries), concurrent updates are merged by *union + ordering*.
    - For scalar values with clear semantics (e.g., plan.stage), a **priority rule** applies (e.g., “higher stage number wins”, or “higher confidence wins”).
    - For complex structures (scene graphs, plan graphs), a merge function is defined:

        - combine non-conflicting nodes/edges,
        - mark conflicting edges with annotations for metacognitive review.
- **Provenance tracking**:
    - Values include the origin subsystem and trace IDs, enabling downstream components (and human operators) to understand *who* wrote *what* and *when*.

Ordering and subscriptions

- The DHT itself does not guarantee strict temporal ordering across updates; instead:
    - clients use version vectors to detect staleness,
    - subscribers to a key or namespace receive updates in best-effort order and reconcile them using the version metadata.

This approach trades strong global consistency for **availability and partition tolerance**, which is appropriate for a system that must remain robust under node failures and network variability.

D. Event Fabric vs DHT: When to Choose Each

To clarify tradeoffs, the architecture adopts the following guideline:

Aspect / Need	**Event Fabric (Baseline)**	**DHT Coordination Layer (Advanced)**
Latency	Microseconds, UCX/InfiniBand path	Higher (network + routing + replication)
Data type	Ephemeral events, triggers, control signals	Structured, persistent cognitive state
Topology	Small to medium number of services	Many micro-agents, multi-node cognitive substrates
Coupling	Tighter (publisher/subscriber aware of topics)	Looser (read/write by key; discovery via DHT)
Failure handling	Retries, heartbeats, backpressure only	Replication, key ownership transfer, eventual recovery
Example uses	plan.step_ready, simulation.result, safety.flag	plan.graph.current, scene.graph.current, skill.library.*

Design rule-of-thumb

- Use the **event fabric** for **fast, local, in-the-moment coordination** of perception, planning, simulation, and control.
- Use the **DHT** when:
    - you have **more than a handful of subsystems** needing access to the same cognitive artifacts,

- you require **state to outlive** individual processes or jobs, or
- you want to allow micro-agents to **self-organize around key ranges** (e.g., certain goals or domains).

In practice, early phases of the system can run wholly on the event fabric; the DHT is introduced when scaling to polycentric cognition and long-lived agents.

E. Preventing Coordination Pathologies

Both layers include safeguards to prevent pathological behaviors such as event storms, circular dependencies, and thrashing updates.

1. **Event Storm Mitigation**
   - Rate limiting and coalescing (as described above) on the event fabric.
   - For the DHT, **write throttling** and **key-level update budgets** prevent overly chatty subsystems from repeatedly overwriting shared state.
2. **Circular Dependencies and Feedback Loops**
   - Every event and DHT write carries a **trace ID** and **depth counter**.
   - A coordination policy component monitors these traces:
     - If a loop pattern is detected (e.g., repeated LH↔RH updates on the same plan without progress), it triggers coordination.loop_detected.
     - Metacognition or a supervisory agent can then intervene:
       - cancel or downgrade the task's priority,
       - enforce a backoff or require human oversight.
3. **Message Validation and Schema Versioning**
   - Both event payloads and DHT value types are versioned.
   - Incompatible versions are rejected or routed through a compatibility shim, reducing the risk of "semantic storms" caused by mismatched expectations across subsystems.

F. Role in the Overall Architecture

The communication and coordination layer turns the system from "a collection of models on different nodes" into a **coherent cognitive whole**:

- The **event fabric** provides real-time, low-latency reactivity and supports the core perception–planning–simulation–action loop.
- The **DHT layer** provides shared, persistent cognitive state and enables scalable, fault-tolerant, polycentric cognition.

- Explicit **ordering, consistency, and loop-prevention policies** make the system's coordination behavior inspectable, tunable, and amenable to formal analysis.

This layered design ensures that, as the architecture evolves from a dual-hemisphere prototype into a multi-agent AGI substrate, the communication backbone remains robust, understandable, and controllable.

VI. Memory Architecture

Memory is not a sidecar; it is a **core substrate** of the architecture. The system is designed to support three major memory types—semantic, episodic, and procedural—each with explicit **query interfaces, indexing strategies, management policies, capacity limits, and consistency guarantees**.

Rather than treating memory as an unbounded vector store, the design assumes:

- memory is **finite** and must be managed,
- access patterns and time matter,
- concept drift will occur,
- not every experience deserves to become a persistent skill.

A. Semantic Memory

Semantic memory stores **abstract, decontextualized knowledge**: concepts, scene embeddings, code snippets, documentation, tool specs, and high-level summaries.

A.1 Data Model and Indexing

Semantic entries are stored in a **vector database backed by CPU nodes** (e.g., Qdrant/FAISS) with **hybrid indexing**:

- **Primary index:** dense vector embedding (e.g., 1–4k dimensions).
- **Secondary indices:**
  - symbolic tags (e.g., domain=physics, type=tool_doc),
  - timestamps (created_at, last_accessed),
  - provenance (source=episodic/LLM/human).

Queries from LH/RH combine **vector similarity + filters**:

```
(embedding ~ nearest_neighbors) ∧
(domain = "navigation") ∧
(created_at > t0)
```

This enables queries scoped by **topic, time, and origin**, not just nearest neighbors.

A.2 Concept Drift and Updates

Semantic memory is explicitly **mutable** to handle concept drift:

- **Versioned entries:**
    - Each concept/key can have multiple versions with timestamps and confidence scores.
    - New evidence can either:
        - create a new version (soft update), or
        - merge with an existing one (hard update) using domain-specific rules.
- **Drift detection:**
    - When new embeddings repeatedly cluster far from older entries on the same concept ID, a **drift flag** is set.
    - Metacognition or an offline consolidation job can then:
        - re-cluster concepts,
        - retire outdated definitions,
        - or split concepts into sub-concepts (e.g., "port" → "port_network" vs "port_harbor").

A.3 Update Frequency and Garbage Collection

Semantic memory is maintained by background jobs:

- **Hot-path writes:**
    - Newly important facts (e.g., tool docs, safety constraints, recent failures) are written synchronously.
- **Batch consolidation:**
    - Periodic jobs (e.g., hourly/daily) re-cluster embeddings, merge duplicates, and adjust weights.
- **Garbage collection policy:**
    - Each semantic entry has a **retention score** derived from:
        - recency of access,
        - frequency of use in successful plans,
        - safety relevance,
        - human/critical flag overrides.
    - When capacity thresholds are approached (memory budget, index size), low-retention entries are:
        - demoted to cold storage (slower, cheaper disks),
        - or deleted if their retention score is below a configured threshold and they are not safety-critical.

B. Episodic Memory

Episodic memory stores **time-ordered experience traces**: what the agent perceived, did, and what happened next.

B.1 Episode Segmentation

Episodes are segmented along **task and temporal boundaries**:

- A new episode is started when:
    - a new top-level goal is adopted,
    - a major environment reset occurs,
    - or a long period of inactivity elapses.
- Within each episode, data is organized into **steps**:

```
{ timestep, observation_ref, action_ref, simulation_ref, outcome,
reflection, reward }
```

Each step stores references to:

- raw sensor data (frames, depth, etc.),
- actions executed,
- world-model predictions,
- actual outcomes,
- reflections and error notes from metacognition.

B.2 Querying Across Time

Episodic memory supports both **time-range** and **semantic** queries:

1. **Time-based queries**
    - Index on (episode_id, timestep) and timestamp.
    - Queries such as:
        - "all steps in last 30 minutes with safety.flag,"
        - "episodes where goal=X between T1 and T2."
2. **Event- and state-based queries**
    - The system maintains secondary indices on **event types** and **state predicates** (e.g., "collision", "task_success", "loop_detected").
    - LH can ask: "show me last 10 episodes where we failed to stack blocks > 3 high."
3. **Semantic queries over episodes**
    - Summaries or key frames of each episode are embedded and indexed in the semantic store, allowing queries like:
        - "episodes similar to the current situation where we succeeded,"

        - "cases where navigation around obstacles in confined spaces worked well."

B.3 Compression and Summarization

To avoid performance degradation:

- **Raw data (frames, sensor streams)** is kept for a limited window and compressed (e.g., WebP/Zstd).
- Periodic **summarization passes** produce:
    - short natural-language summaries,
    - structured graphs of key events,
    - distilled "lessons learned."
- After summarization, detailed logs can be:
    - archived to cheaper storage, or
    - partially discarded, keeping only summaries + a few representative snapshots.

C. Procedural Memory

Procedural memory stores **skills**: reusable behaviors and workflows (e.g., "navigate around docked boat," "stack blocks safely," "diagnose generator issue").

C.1 Skill Representation

A skill is a structured object:

```
skill {
  name,
  preconditions,
  postconditions,
  policy_ref,           // controller, script, or plan template
  required_tools,
  success_stats,        // success rate, last_used, last_updated
  domain_tags,
  confidence
}
```

Skills can point to:

- parameterized plan templates (for LH),
- low-level controllers or motion policies (for RH),
- or both.

C.2 How Skills Are Learned

Skills are not created for every episode; they are **promoted** through a **skill discovery pipeline**:

1. **Candidate identification**
    - A background process scans episodic memory for:
        - repeated successful patterns (e.g., similar sequences of actions solving similar goals),
        - high reward / low risk trajectories,
        - sequences repeatedly re-generated by LH planning.
2. **Pattern mining and abstraction**
    - Similar trajectories and plans are clustered.
    - If a cluster has:
        - sufficient support (used N times),
        - high success rate,
        - consistent pre/post conditions,
        then a candidate skill is formed.
3. **Promotion decision**
    - The **skill manager** (part of the procedural memory subsystem + LH metacognition) decides whether to promote a candidate:
        - gains in planning speed or reliability justify the storage cost,
        - no conflict with existing higher-confidence skills for the same niche.
4. **Validation**
    - Promoted skills are tested on **held-out tasks** or simulated scenarios in RH's world model.
    - Skills that perform poorly are either refined or downgraded back to "patterns" rather than full skills.

C.3 Skill Maintenance and Retirement

Procedural memory also has capacity limits:

- Skills track **usage frequency and success rate**.
- Under storage pressure, a **skill GC process**:
    - retires rarely used, low-success skills,
    - merges highly similar skills,
    - or compresses them into higher-level meta-skills.

D. Memory Management Policies and Capacity Limits

All three memories operate under **explicit capacity budgets** to prevent unbounded growth and performance collapse.

1. **Budgets per memory type**
    - Semantic: max number of active vectors + index size budget.
    - Episodic: rolling time window of detailed logs (e.g., last N hours/days) plus compact summaries beyond that.
    - Procedural: maximum number of active skills per domain or overall.
2. **Retention scoring**
   Each memory item has a retention score driven by:
    - recency and frequency of access,
    - contribution to successful behavior (reward-weighted usage),
    - safety relevance (safety-related data is strongly protected),
    - human "pinning" (items that operators never want deleted).
3. **Tiered storage**
    - **Hot tier**: fast local disks / RAM indices for recent and frequently used items.
    - **Warm/cold tier**: slower storage (object store, network-attached disks) for older or lower-score items.
    - GC demotes items between tiers, not just deletes them.
4. **Performance monitoring**
    - The system continuously monitors query latency, index size, and cache hit rates.
    - When latency crosses thresholds, cognitive economics can:
        - delay low-priority writes,
        - throttle long-range episodic queries,
        - schedule extra consolidation/GC cycles.

E. Cold-Start and Bootstrapping

At system launch or after a major reset, memory is "cold":

1. **Preloaded priors**
    - Semantic memory is pre-populated with:
        - tool documentation,
        - environment schema,
        - safety constraints,

        - basic domain knowledge.
    - Procedural memory can include a small set of **hand-authored or pre-trained skills** (e.g., basic navigation, reach/grasp).

2. **Early-phase retrieval heuristics**
    - When memory is small or sparse, retrieval relies more heavily on **model priors** (LLM/world model) and fewer on similarity scores.
    - Confidence estimates reflect this: the system knows when it has limited experience.

3. **Ramp-up strategy**
    - The curriculum in the virtual environment is designed to **quickly populate** episodic and procedural memory with diverse but safe experiences, giving the system enough raw material to start discovering meaningful skills.

F. Memory Consistency and Integration with Coordination Layers

The memory subsystem must be consistent **enough** to support coherent cognition while still scalable and robust.

1. **Internal consistency within a node**
    - Each memory service (semantic, episodic, procedural) is strongly consistent **locally**:
        - writes are committed synchronously to local storage and indices,
        - reads see the latest committed state on that node.

2. **Cluster-wide consistency**
    - When memory is sharded or replicated across nodes, the system uses:
        - **eventual consistency** at cluster level (via the DHT) for large-scale, long-lived state, and
        - explicit versioning (timestamps or version vectors) for conflict detection.

1. **Interaction with event fabric and DHT**
    - **Event fabric** is used for:
        - notifications (memory.updated, skill.added, episode.summarized),
        - cache invalidation signals to hemispheres.
    - **DHT** is used to store:
        - global indices and summaries (e.g., episodic.index, skill.catalog, semantic.cluster_map),
        - pointers to where detailed memory for a given key/episode/skill lives.

2. **Read-your-writes guarantees**
    - For a given subsystem, the memory APIs guarantee **read-your-writes**:
        - if LH writes a semantic entry or procedural skill, subsequent reads from LH will see it, even if cluster-wide propagation is still in progress.

G. Summary

This memory architecture turns "semantic/episodic/procedural" from labels into **operational components** with:

- clear **data models and indexing strategies**,
- **query semantics** (including time-aware episodic queries),
- **concept drift handling** and garbage collection for semantic memory,
- an explicit **skill discovery and promotion pipeline** for procedural memory,
- **capacity-aware management policies**,
- a **cold-start strategy**, and
- well-defined **consistency guarantees** integrated with the event fabric and DHT.

Together, these mechanisms allow the AGI system to build, maintain, and refine a long-lived memory substrate without collapsing under its own history.

VII. Safety Architecture

Safety is treated as a **first-class cognitive subsystem** woven through planning, perception, memory, and execution—not as an external filter. The architecture retains the three-layer structure (Pre-Action, In-Action, Post-Action) but makes the safety logic **operational and measurable**, with explicit definitions, budgets, and feedback loops.

A. Pre-Action Safety (Static Risk Filtering)

Pre-action safety evaluates *plans, code, and tool invocations* before they reach the world model or the environment.

A.1 Hallucination Definition and Detection

For this system, a **hallucination** is defined as a *semantically or operationally invalid proposal* that appears well-formed linguistically but fails structural or factual checks against the system's knowledge and APIs.

Examples:

- **Code hallucination**

    - References non-existent APIs, tools, or system capabilities.
    - Violates type signatures or schemas of known APIs.
    - Attempts operations outside the allowed sandbox (e.g., arbitrary filesystem access).
- **Plan hallucination**
    - References objects that do not exist in the current scene.
    - Assumes preconditions that contradict perception or memory (e.g., "door is open" when RH reports closed).
    - Uses tools the system is not authorized to access.

Detection is implemented via:

1. **Schema and registry checks**
    - All tools, APIs, and environment actions are defined in a **tool registry** with schemas.
    - Generated code and plans are checked against this registry; mismatches flag hallucinations.
2. **Consistency checks with memory and perception**
    - LH proposals are validated against semantic memory (tool docs, environment schema) and current RH state.
    - If required objects, locations, or states cannot be found or contradict recent observations, the plan step is flagged.
3. **Static analysis**
    - Code proposals run through static analyzers and sandbox "dry runs" to detect dangerous imports, system calls, or resource abuse patterns.

Hallucination flags do **not** always hard-block; they contribute to a **risk score** that combines structural invalidity, safety relevance, and history of similar errors.

A.2 Blocking Thresholds and False Positives

Pre-action safety must balance:

- **False negatives** (letting unsafe actions through), and
- **False positives** (blocking safe but novel or slightly unusual actions).

The system uses **configurable thresholds**, tuned per domain:

- **High-risk domains** (physical manipulation near fragile objects, safety-critical tools):

    - Low tolerance for false negatives.
    - Pre-action checks can block with higher false positive rates (e.g., err on the side of refusal and require operator override).
- **Low-risk domains** (purely internal computation or simulation):
    - Higher tolerance for false negatives; more permissive thresholds to avoid over-blocking.

Metrics tracked:

- fraction of blocked actions later judged safe (false positives),
- fraction of allowed actions that lead to safety incidents (false negatives),
- average added latency per safety decision.

Thresholds are adjusted offline based on these metrics and operator feedback.

A.3 Updating Safety Rules and Policies

Safety rules are not static:

- **Rule refinement**
    - If a rule repeatedly blocks actions that are retrospectively labelled safe (e.g., by human operators or successful simulations), it is tagged as "overly restrictive."
    - The safety subsystem proposes relaxations (e.g., narrower regex, more specific context conditions) which must be approved in a controlled mode.
- **New rule introduction**
    - When Post-Action analysis identifies a new failure pattern, a corresponding pre-action rule is added (e.g., "never request tool X with parameter Y in context Z").
- **Versioning and rollback**
    - Rules are versioned and stored in semantic/procedural memory.
    - If new rules cause unacceptable disruption (spike in false positives), the system can roll back to a previous policy set.

B. In-Action Safety (Dynamic Monitoring & Runtime Guards)

In-action safety monitors behavior **while the system is acting**, particularly in the virtual environment.

B.1 Prediction Horizon and Rationale

The RH's world model performs **N-step lookahead** (typically 5–20 steps):

- Short horizon (< 5 steps): too myopic, misses near-term compounding risks.
- Very long horizon (> 20–30 steps): computationally expensive and less reliable due to compounding model errors.

The 5–20 range is a **tunable compromise**:

- During **fine manipulation** or high-risk scenarios, the horizon can be extended or sampling density increased.
- For simpler tasks, shorter horizons reduce compute.

To address "step 21" risks:

- The system uses **receding-horizon control**: after each executed step (or small bundle of steps), the RH re-simulates the next 5–20 steps from the *new* state.
- This continually pushes the predictive window forward, so longer-term hazards eventually appear within horizon as execution progresses.

B.2 Computational Budget for Safety Checks

Safety checks share GPUs/CPUs with other workloads, so they have explicit **compute budgets**:

- Each plan step or control cycle is allocated a safety budget (e.g., max simulation time, max number of sampled trajectories).
- The cognitive economics layer prioritizes safety simulations above background tasks; if resources are limited, non-critical simulations are dropped before safety checks are compromised.

If a safety check **cannot complete** within budget (timeout or overload):

- The default behavior is **conservative**:
    - degrade to a simpler, known-safe policy (e.g., stop motion, hold position), and/or
    - downgrade action confidence and request human oversight for high-risk tasks.

B.3 Handling Safety Failures at Runtime

When In-Action safety detects risk (e.g., predicted collision, violation of constraints, unexpected strong contact):

- **Soft failure (recoverable)**
    - Emit trajectory.risk_high or safety.flag.
    - Interrupt the current plan segment.
    - Hand control back to LH for replanning with new constraints (e.g., "do not enter region R").

- **Hard failure (emergency)**
    - Trigger emergency_stop: halt actuators, move to safe configuration if possible.
    - Lock out further actions in that episode until a safety review is done.
    - Log all relevant sensory, plan, and simulation data into episodic memory.

Degradation strategies include:

- reducing control speed,
- increasing safety margins (larger distances, lower forces),
- restricting allowable action space until confidence is restored.

C. Post-Action Safety (Outcome Verification & Learning)

Post-action safety closes the loop by comparing **what was expected** to **what actually happened**.

C.1 Unforeseen vs. Unmodeled

We distinguish:

- **Unmodeled** behavior:
    - World model **signaled high uncertainty** or lack of coverage beforehand (e.g., novel objects, out-of-distribution states).
    - The system knew it "didn't know" and should have acted conservatively.
- **Unforeseen** behavior:
    - World model predicted low risk; safety thresholds were satisfied.
    - Nevertheless, an adverse or unexpected outcome occurred.

Operationally:

- Unmodeled cases indicate **epistemic uncertainty** that was correctly flagged but not handled conservatively enough.
- Unforeseen cases indicate **model error or safety logic failure** and require targeted retraining or rule updates.

C.2 Feedback Loop to Improve Safety Modules

Post-Action analysis feeds multiple loops:

1. **World model updates**
    - Episodes with unforeseen outcomes are added to a **world-model error dataset**.

    - Periodic retraining or fine-tuning reduces future prediction error in similar states.

2. **Rulebase refinement**
    - Safety incidents that slip through pre- or in-action filters generate candidate rules or threshold changes.
    - These proposals are stored in semantic/procedural memory and can be human-reviewed before activation.
3. **Skill adaptation**
    - Procedural skills involved in incidents see their **confidence and priority reduced**.
    - The skill manager may require skills to be revalidated in simulation before reuse.
4. **Metacognitive calibration**
    - Confidence metrics are recalibrated using empirical data (how often high-confidence predictions were wrong).
    - This reduces overconfidence and can increase the frequency of reflection for similar tasks.

D. Safety–Performance Tradeoffs

Safety inevitably trades off with performance (latency, throughput, exploration):

- **Latency vs. risk**
    - More extensive simulation and reflection reduce risk but increase response time.
    - The architecture allows per-task tuning: safety-critical tasks favor more checks; low-stakes tasks may accept faster, less conservative behavior.
- **Exploration vs. conservatism**
    - Highly conservative policies limit learning and skill acquisition.
    - In low-risk simulated settings, exploration limits are relaxed to allow the agent to discover new strategies.
- **Resource allocation**
    - Safety checks consume GPU/CPU cycles that could otherwise be used for planning or learning.
    - Cognitive economics explicitly reserves a safety quota but can adjust it based on observed incident rates and task mix.

These tradeoffs are **parameterized knobs** rather than hard-coded, enabling operators to adjust policies based on domain, environment, and risk tolerance.

E. Known Attack Vectors and Mitigations

While the initial deployment is in a simulated environment, the architecture anticipates adversarial behavior:

1. **Prompt and tool injection**
    - Adversarial instructions attempting to bypass safety or invoke unauthorized tools.
    - Mitigation: strict tool registry; separation between natural-language instructions and tool/actuation channels; safety filters that consider context and provenance.
2. **Adversarial observations**
    - Crafted environment states or sensor patterns that confuse perception or world model.
    - Mitigation: ensemble checks, anomaly detection on sensor data, and fallback to conservative policies when anomalies are detected.
3. **DHT and event fabric abuse**
    - Poisoned shared state (e.g., malicious keys) or event storms.
    - Mitigation: key signing, access control lists for write permissions, rate limiting, loop detection, and trace-based auditing of state changes.
4. **Code execution exploits**
    - Generated code attempting to escape sandboxes or escalate privileges.
    - Mitigation: strict sandboxing (seccomp, container isolation), limited system calls, resource quotas, and static/dynamic analysis before execution.

F. Adversarial Testing and Safety Validation Methodology

Safety is not assumed; it is **empirically tested and red-teamed**.

1. **Simulation-based stress testing**
    - Construct adversarial scenarios in the virtual environment:
        - narrow clearances, moving obstacles, deceptive object arrangements.

- Measure incident rates, near-misses, and how often safety mechanisms intervene.

2. **Prompt-level adversarial testing**
    - Red-team prompts aimed at:
        - bypassing tool restrictions,
        - inducing dangerous code generation,
        - confusing LH about goals or constraints.
    - Track how often safety layers block these attempts and refine rules accordingly.
3. **Fuzzing of plans and actions**
    - Randomly perturb plans and control sequences then run them through safety checks to evaluate coverage and robustness.
4. **Regression and continuous validation**
    - Maintain a suite of safety scenarios and adversarial tasks as **regression tests**.
    - Any change in models or rules must pass this suite before deployment.
5. **Metrics and reporting**
    - Key safety KPIs:
        - unsafe actions blocked vs. allowed,
        - false positive rate on safe actions,
        - number of emergency stops,
        - near-miss rate (predicted vs. actual).
    - These metrics inform ongoing tuning of thresholds and rule sets.

G. Summary

The safety architecture is now specified as a **measurable, adaptive subsystem**:

- **Pre-Action** safety rigorously defines and detects hallucinations, with tunable thresholds and rule-updating mechanisms.
- **In-Action** safety uses receding-horizon predictions under explicit compute budgets, with well-defined failure and degradation behaviors.
- **Post-Action** safety distinguishes unforeseen vs. unmodeled failures and feeds training data back into world models, rules, and skills.

- Safety is evaluated through structured **tradeoff analysis, adversarial testing, and regression suites**, ensuring that as the system learns and scales, safety remains an integral, continuously improving property of the architecture.

VIII. Metacognition Layer

The metacognition layer is the subsystem responsible for **reasoning about the system's own reasoning**. It does not replace planning, perception, or safety; instead, it **monitors, critiques, and adjusts** them under explicit time and compute budgets. Its goals are to:

- detect flawed reasoning,
- calibrate confidence,
- resolve cross-hemisphere inconsistencies, and
- drive self-improvement over long time scales.

Metacognition runs primarily on the Left Hemisphere (LH), but it consumes evidence from the Right Hemisphere (RH), memory nodes, and the safety subsystem.

A. Role and Objectives

Metacognition has four core responsibilities:

1. **Quality control of reasoning and plans**
    - Inspect chains-of-thought, plans, and explanations produced by LH.
    - Identify logical gaps, contradictions, or unjustified assumptions.
2. **Confidence estimation and calibration**
    - Combine signals from memory, perception, and world-model predictions into an actionable confidence score.
1. **Cross-hemisphere consistency checking**
    - Compare symbolic expectations (LH) with physical predictions (RH).
    - Flag mismatches that could lead to unsafe or doomed-to-fail behavior.
2. **Self-improvement and learning guidance**
    - Decide when to store new lessons, refine skills, or retrain components.
    - Influence curriculum and exploration strategies in the virtual environment.

Metacognition itself is **bounded and schedulable**—it is subject to the same cognitive economics and convergence rules as other subsystems.

B. Core Mechanisms

B.1 Self-Evaluation of Reasoning and Plans

When LH generates a plan or a multi-step reasoning trace, it also produces a **self-explanation artifact**:

- an explicit chain-of-thought graph (internally),
- references to supporting memories,
- the assumptions it relied on (e.g., "object X remains stationary," "tool Y behaves as documented").

Metacognition runs a **critic pass** over this artifact using:

1. **Logical checks**
    - Look for circular reasoning, missed dependencies, or contradictory conclusions.
    - Verify that each plan step has justified preconditions and expected effects.
2. **Evidence checks**
    - Confirm that referenced facts exist in semantic memory and match RH's latest state where relevant.
    - Mark steps as weakly supported if evidence is missing or outdated.
3. **Pattern-based heuristics**
    - Compare current chains with past success/failure episodes in episodic memory:
        - "Have we used a similar plan before?"
        - "How often did this structure lead to errors?"

The result is a **reasoning quality score** and a set of annotations (e.g., "assumption A unsupported," "step B contradicted by history").

B.2 Cross-Hemisphere Reflection

Metacognition orchestrates structured "cross-checks" between LH and RH:

- For each critical plan segment, it requests RH's **world-model simulation** and compares:
    - LH's symbolic expectations (e.g., "block will rest stably") vs.
    - RH's predicted trajectories and failure probabilities.
- It also compares:
    - semantic priors (e.g., known object properties) vs.

    - RH's perception and state estimates (e.g., current pose, friction, geometry).

Disagreements are categorized as:

- **semantic vs perception mismatch**,
- **plan vs physics mismatch**, or
- **model vs experience mismatch** (history shows different outcomes).

Each mismatch contributes to lower confidence and may trigger re-planning, re-simulation, or memory updates.

B.3 Hypothesis Management

Metacognition treats explanations and plans as **hypotheses**:

- A hypothesis includes:
    - a proposed plan or belief,
    - a set of supporting evidence,
    - confidence and uncertainty estimates.

Hypotheses can be:

- **accepted** (within thresholds → proceed),
- **revised** (modify steps, add constraints), or
- **rejected** (unsafe/incoherent → discard and replan).

These states are written into episodic memory, allowing the system to later analyze **when and why** certain hypotheses failed or succeeded.

C. Triggers, Budgets, and Convergence

To avoid unbounded introspection, metacognition is activated under explicit policies.

C.1 When Metacognition Runs

Metacognitive passes are triggered by:

- **High-risk or novel tasks**
    - Plans involving safety-critical actions or unfamiliar environments.
- **Low-confidence signals**
    - LH or RH explicitly tags outputs as low-confidence.
- **Cross-hemisphere conflicts**

    - Events such as perception.state_mismatch, trajectory.risk_high, or goal.infeasible.

- **Post-incident review**
    - After safety incidents or near-misses, dedicated retrospective passes analyze what went wrong.

Routine, low-risk tasks may only receive lightweight, fast metacognitive checks.

C.2 Time and Compute Budgets

Metacognition is governed by **cognitive economics**:

- Each metacognitive invocation receives a budget:
    - max wall-clock time (e.g., 50–500 ms for online decisions),
    - max inference tokens or model calls,
    - max number of alternative plans to evaluate.

Budget allocation depends on:

- safety-criticality,
- expected benefit (historical impact of reflection in similar cases),
- current system load.

If the budget is exhausted:

- Metacognition returns the **best current assessment**, including:
    - a confidence score,
    - a list of unresolved issues,
    - a recommendation (proceed / proceed but mark risky / stop and ask for help).

This guarantees **finite-time convergence** even under heavy load.

C.3 Convergence Criteria

Metacognitive loops terminate when:

- **No new issues found** within a fixed number of reflection passes.
- **Confidence thresholds** are reached (e.g., combined confidence from semantics + world-model + history > target).
- **External interrupts** occur (e.g., safety emergency, higher-priority task).

If convergence is not achieved but acting is necessary (e.g., deadlines, safety constraints), the system acts conservatively and records that it did so under **explicitly low-confidence conditions**.

D. Confidence Estimation and Calibration

Confidence is not a single scalar but a **vector of signals**:

- **Semantic agreement**
    - How well does the plan align with known facts in semantic memory?
- **Physics agreement**
    - Does RH's world model predict successful, low-risk outcomes?
- **Historical success**
    - How often have similar plans worked in the past?
- **Internal consistency**
    - Does the reasoning chain pass logical and structural checks?

Metacognition combines these signals into:

- a composite **confidence score**, and
- a breakdown for diagnostics (e.g., "high semantic support, low physics support").

Calibration uses Post-Action data:

- If high-confidence plans regularly fail, metacognition reduces trust in certain signals (e.g., world model in a certain regime) and **tightens thresholds**.
- If low-confidence plans often succeed, thresholds may be relaxed to avoid unnecessary conservatism.

E. Interaction with Memory and Learning

Metacognition is the **glue** between online decision-making and long-term learning.

E.1 Writing to Memory

Metacognitive outcomes are written as:

- **annotated episodes** in episodic memory:
    - including confidence scores, identified flaws, and final outcomes.
- **semantic updates**:
    - corrections to factual knowledge when consistent discrepancies are found.
- **procedural updates**:

    - promotions or demotions of skills based on their performance and reliability.

This makes memory a **reflective record**, not just a raw log.

E.2 Guiding Skill Discovery and Safety Enhancements

Metacognition flags:

- **candidate skills**
    - sequences of actions that repeatedly succeed with strong evidence and high confidence.
- **anti-patterns**
    - recurring failure modes that should be addressed with new safety rules or skill adjustments.

These flags guide:

- the **skill manager** in procedural memory (what to promote or retire), and
- the **safety subsystem** (what new rules or thresholds to add).

E.3 Curriculum and Exploration Control

Metacognition can request:

- more training in areas where confidence is systematically low,
- targeted scenarios to probe model weaknesses (e.g., edge cases in physics),
- adversarial tests to validate safety assumptions.

These requests feed into long-term **curriculum design** in the virtual environment.

F. Observability and Diagnostics

Metacognition is designed to be **inspectable**:

- Every metacognitive run produces a **structured report**:
    - what was evaluated,
    - which inconsistencies were found,
    - how confidence was computed,
    - what decision was taken (accept/revise/reject).
- These reports are accessible via monitoring dashboards and logs for human operators and researchers.

This makes it possible to:

- debug “why the system trusted this plan,”
- analyze trends in confidence vs. actual success,
- refine metacognitive heuristics and thresholds over time.

G. Failure Modes and Safeguards

Metacognition itself can fail or misbehave:

- **Overthinking / indecision**
    - Safeguard: strict budgets and convergence rules; when exceeded, defer to safe defaults or human oversight.
- **Excessive pessimism**
    - Safeguard: calibration using empirical success data; if metacognition blocks many ultimately safe plans, thresholds are relaxed.
- **Blind spots**
    - Safeguard: adversarial and regression testing to expose patterns metacognition misses; add new checks or models in response.

These safeguards ensure metacognition remains a **helpful critic**, not a bottleneck or source of paralysis.

H. Summary

The metacognition layer upgrades the architecture from “a set of coordinated models” to a **self-monitoring cognitive system**:

- It **evaluates** reasoning and plans,
- **quantifies and calibrates** confidence,
- **reconciles** symbolic and physical predictions,
- **records and learns from** its own mistakes, and
- operates under explicit **resource and convergence constraints**.

Together with the safety and memory subsystems, metacognition provides the foundation for **continuous, data-driven self-improvement** in a distributed, embodied AGI system.

IX. Lifelong Learning System

IX. Lifelong Learning System

The lifelong learning system is the mechanism by which the architecture **improves over time** without losing stability, safety, or coherence. It ties together perception, planning, metacognition, safety, and all three memory stores (semantic, episodic, procedural) into a continuous adaptation loop.

The goals are to:

- increase competence and efficiency with experience,
- avoid catastrophic forgetting and uncontrolled drift,
- keep safety performance at least as strong as capabilities, and
- make learning resource-aware and schedulable on HPC infrastructure.

A. Learning Signals and Data Sources

The system learns from multiple signals:

- **Task outcomes** – success/failure, reward, safety incidents, near-misses.
- **Metacognitive assessments** – confidence scores, detected reasoning flaws, cross-hemisphere inconsistencies.
- **Safety feedback** – pre-action blocks, in-action interventions, post-incident analyses.
- **Usage patterns** – which skills, facts, and plans are used frequently and successfully.

These signals are written into episodic and semantic memory as **annotated experiences**, forming the raw material for consolidation and skill discovery.

B. Semantic Learning and Consolidation

Semantic learning focuses on updating **abstract knowledge and priors**:

1. **Incremental Concept Updates**
    - When repeated episodes contradict or refine existing semantic entries (e.g., tool behavior, object properties), metacognition proposes semantic updates.
1.
    - New evidence is merged using versioned entries and drift detection (from VI): concepts can be refined, split, or re-weighted rather than overwritten blindly.
2. **Consolidation Jobs**

- Periodic background jobs (scheduled by the cognitive economics layer) perform:
  - re-clustering of semantic embeddings,
  - deduplication of low-value or redundant entries,
  - promotion of frequently-used summaries and schemas.
- Safety-critical knowledge (constraints, hazards) is pinned and treated with higher retention priority.

3. **Stability–Plasticity Control**
   - Consolidation limits how much semantic memory can change per cycle (e.g., max fraction of entries updated), preventing sudden shifts.
   - High-confidence, frequently verified entries are harder to modify; low-confidence or rarely used entries are more plastic.

C. Episodic Encoding, Compression, and Replay

Episodic memory acts as the **experience buffer**: detailed traces feed future learning and debugging.

1. **Rich Encoding at Runtime**
   - During operation, the system logs:
     - observations, actions, world-model predictions, and actual outcomes,
     - metacognitive reports and safety decisions,
     - task metadata (goal, context, reward).
2. **Temporal Compression and Summarization**
   - Background processes compress episodes by:
     - identifying key events (subgoal completion, safety alerts, failures),
     - summarizing long stretches of “routine” behavior,
     - selecting representative frames and trajectories.
   - This reduces storage while retaining **behaviorally important** information.
3. **Experience Replay**
   - For world model and skill refinement, targeted subsets of episodes are replayed in offline jobs on the cluster:
     - high-impact failures and near-misses,

        - edge-case successes,
        - scenarios where confidence and outcomes diverged (over/underconfidence).
    - Replay is prioritized by safety relevance and expected learning benefit.

D. Procedural Skill Formation, Refinement, and Retirement

Procedural memory captures **how to do things** efficiently and safely.

1. **Skill Discovery (Promotion)**
    - A skill discovery pipeline mines episodic memory for **repeated successful patterns**:
        - similar action sequences solving similar goals under similar conditions,
        - plans LH tends to recreate,
        - trajectories RH executes with high success and low risk.
    - Candidates are abstracted into skill templates with pre/post conditions and policies.
2. **Skill Validation and Integration**
    - Candidate skills are validated in RH's world model across held-out scenarios.
    - Only skills meeting minimum success and safety thresholds are promoted to procedural memory and exposed to LH's planner as reusable building blocks.
3. **Skill Refinement**
    - When a skill underperforms (e.g., rising failure rate, safety flags), metacognition triggers refinement:
        - adjust parameters or preconditions,
        - split a skill into simpler sub-skills,
        - or restrict its domain of applicability.
4. **Skill Retirement (GC)**
    - Skills track usage frequency, success rate, and safety incidents.
    - Under capacity pressure or persistent poor performance, skills can be:
        - downgraded to patterns (not directly used by the planner),

        - fully retired, with their history kept only in episodic/semantic summaries.

E. Curriculum, Exploration, and Cold-Start

Lifelong learning is steered by a **curriculum and exploration policy** integrated with the virtual environment.

1. **Cold-Start Phase**
    - At boot, the system relies heavily on:
        - preloaded semantic priors (tool docs, environment schema, safety constraints),
        - a small hand-crafted or pre-trained skill set.
    - The environment presents **simplified tasks** designed to quickly populate episodic memory with diverse but safe experiences.
2. **Curriculum Progression**
    - As skills and confidence grow, the curriculum generator escalates difficulty:
        - more complex object arrangements,
        - longer-horizon tasks,
        - multi-step tool use and obstacle navigation.
    - Metacognition can request **focused practice** in weak areas (where confidence or success rate is low).
3. **Exploration vs. Exploitation**
    - In low-risk simulated settings, exploration is encouraged:
        - randomized variations of tasks,
        - deliberate attempts to test the limits of world-model predictions.
    - Exploration parameters (e.g., noise level, novelty preference) are tuned based on safety and performance metrics.

F. Online vs. Offline Learning and HPC Scheduling

The architecture separates **online adaptation** from **offline heavy training** to fit HPC realities.

1. **Online Learning (Fast Path)**

- Light-weight updates that can run during normal operation:
  - adjusting skill usage priorities,
  - tuning thresholds and confidence weights,
  - writing summaries and simple semantic updates.
- These are scheduled as low-latency jobs on the same nodes as LH/RH or memory.

2. **Offline Learning (Batch Jobs)**
   - Heavier updates (e.g., world model fine-tuning, large-scale semantic re-clustering, new skill policy training) run as SLURM batch jobs on the cluster.
   - Jobs consume episodic datasets and write back updated models or skills to memory.
   - The event fabric/DHT broadcasts model.updated or skill.library.updated notifications so live services refresh their local caches in a controlled way.
3. **Change Management**
   - Updates to core models (LLM, world model) and major skill libraries go through:
     - evaluation against safety and regression suites (from VII),
     - staged deployment (shadow mode, canary tests),
     - rollback paths if regressions are detected.

G. Safety-Aware Learning and Guardrails

Learning must **never silently erode safety**:

1. **Safety-Informed Losses and Objectives**
   - Training objectives for world models, skills, and policies incorporate:
     - penalties for hazardous outcomes,
     - bonuses for near-miss avoidance,
     - preferences for interpretable, robust strategies where possible.
2. **Safety Constraints on Policy Updates**
   - New or updated skills are tested under adversarial and stress scenarios before being accepted.

    - Safety rules (pre- and in-action) bound what learned policies are allowed to propose or execute.

3. **Drift Monitoring**
    - The system tracks safety metrics over time:
        - incident rate, emergency stops, near-misses.
    - If safety metrics degrade, learning rates and exploration are reduced, and the system may revert to older models or skill sets while investigation occurs.

H. Evaluation of Learning Progress

Lifelong learning is evaluated along three axes:

- **Task performance**
    - success rates, task completion time, energy/effort use, robustness to perturbations.
- **Cognitive efficiency**
    - planning depth and latency, reliance on reflection, frequency of re-planning.
- **Safety & reliability**
    - incident rates, near-misses, safety intervention frequency, calibration of confidence vs. actual outcomes.

These metrics are tracked longitudinally, allowing the system (and human operators) to see whether learning:

- is genuinely improving competence,
- maintains or improves safety, and
- respects the computational and memory budgets of the underlying HPC platform.

I. Summary

The lifelong learning system turns the architecture into a **continually adapting AGI substrate**:

- Semantic memory evolves via controlled consolidation and drift handling.
- Episodic memory captures rich experience but is compressed and replayed intelligently.
- Procedural memory accumulates and curates a library of validated skills.
- Curriculum and exploration are steered by metacognition and safety.

- Online and offline learning are scheduled to fit HPC constraints and safety requirements.

In combination, these mechanisms let the system **learn from its own history**—improving planning, perception, and control—while preserving stability, safety, and interpretability over long operational horizons.

X. Virtual Embodiment Environment

The virtual embodiment environment is the AGI's **body and world**. It is not just a demo front-end; it is the primary source of **causal feedback**, **curriculum**, **adversarial tests**, and **ground truth** against which plans, world models, and safety mechanisms are continuously evaluated. The environment runs on a separate consumer-grade gaming laptop to decouple simulation reliability and GPU usage from HPC cognition, while still providing high-frequency sensorimotor loops to the Right Hemisphere (RH) and, indirectly, to the Left Hemisphere (LH), memory, safety, and metacognition subsystems.

A. Design Goals for Embodiment

The virtual world is designed to satisfy four core requirements:

1. **Physics-consistent causality**
    - Actions must produce predictable, physically plausible outcomes (within model limits), enabling the world model to learn meaningful dynamics.
2. **Continuous sensorimotor loop**
    - The agent receives time-continuous observations and sends time-continuous actions rather than sporadic tool calls.
3. **Curriculum and difficulty control**
    - The environment must support structured progression from simple, safe tasks to complex, multi-step scenarios, including adversarial test cases.
1. **Safe yet rich experimentation**
    - The system should be able to "break things" virtually without real-world harm, but with enough realism that lessons transfer to physical systems or higher-fidelity simulations later.

B. Environment Engines and Deployment Model

The architecture supports multiple physics/simulation backends:

- **Unity** – high flexibility and tooling for interactive 3D worlds and curricula.
- **MuJoCo** – high-accuracy rigid-body physics and control-oriented tasks.
- **Unreal / Webots / Isaac Gym** – optional for specific needs (photorealism, robotics, GPU-accelerated batch RL).

In the baseline deployment:

- A **single chosen engine** (typically Unity or MuJoCo) runs in **headless mode** on a gaming laptop with a dedicated GPU.
- The engine exposes a **network API** (WebSocket/gRPC/REST) for sensors and actuators, and is treated as a separate service by the RH.
- Simulation rate (e.g., 30–120 Hz) and rendering fidelity are adjustable knobs that trade off realism against bandwidth and compute.

This separation ensures that environment crashes, heavy scenes, or GPU stalls **cannot directly destabilize** HPC cognition.

C. Sensor Stream API (Perception Interface)

The environment streams a structured set of observations to the RH at a configurable rate:

1. **Visual Data**
    - RGB frames:
        - Resolution: 640×480 to 1920×1080 (configurable).
        - Frame rate: 30–120 FPS depending on scene complexity.
        - Encoded (e.g., WebP/JPEG) for low-latency transfer.
    - Depth maps / point clouds (optional):
        - Lower resolution (e.g., 320×240) and lower frequency if bandwidth is constrained.
2. **Semantic and Object Metadata**
    - Object IDs, types, positions, velocities, and affordances (e.g., “graspable”, “pushable”).
    - Scene graph descriptors (e.g., “cup_on_table”, “block_inside_box”) enabling RH to construct high-level representations.
3. **Proprioception and Contact**
    - Joint positions and velocities.

- Center-of-mass and balance measures.
- Contact events (what touched what, forces involved).

4. **Auxiliary Signals**
    - Scenario IDs, task labels, and environment constraints (e.g., “no objects may fall off table”).
    - Optional audio streams for multimodal experiments.

The RH converts these streams into embeddings, scene graphs, and state trackers, logging references to raw data into episodic memory for later replay and analysis.

D. Actuator Command API (Action Interface)

The agent affects the world via an **actuator command API** that is tightly integrated with safety and control:

1. **Action Primitives**

Examples include:

- High-level commands (LH-level intent):
    - navigate_to(position), pick_and_place(object_id, target_position), open(container_id).
- Low-level control (RH-level trajectories):
    - joint target positions/velocities/efforts,
    - Cartesian end-effector poses,
    - discrete “mode switches” (e.g., grasp/release).

2. **Safety Firewall and Rate Limiting**

Before commands reach the simulator:

- In-Action safety checks enforce:
    - joint limits, collision constraints, speed/force bounds,
    - environment-specific safety rules (e.g., “never apply more than X torque to hinge Y”).
- Commands that violate constraints are either:
    - clipped to safe ranges,
    - rejected with a control.error / safety.flag event, or

- cause an emergency_stop if already in a hazardous trajectory.

3. **Bidirectional Time Control**

For some training and testing scenarios:

- The environment can support:
  - **pausing** and **slowing down** time to allow more intensive planning/simulation,
  - **fast-forwarding** deterministic segments (e.g., repeated waiting periods),
  - **resetting** to a checkpoint for repeated trials of the same task.

Time control is itself restricted by safety and curriculum logic to prevent unrealistic exploiting (e.g., rewinding outcomes during online evaluation).

E. Curriculum, Task Design, and Adversarial Scenarios

The embodiment environment includes a programmable **scenario and curriculum manager**:

1. **Task Templates and Levels**

Tasks are defined with:

- goals (e.g., “stack blocks to height N”, “navigate around obstacles to target region”),
- constraints (e.g., “do not drop objects”, “stay within region”),
- initial conditions and randomization ranges.

Difficulty increases in **phases**, aligned with the AGI’s internal roadmap:

- Phase 1: simple reaching, navigation, and single-object manipulation.
- Phase 2: multi-step tasks, tool use, and partial observability.
- Phase 3+: multi-object coordination, long-horizon puzzles, and sparse reward tasks.

2. **Adversarial and Safety-Stress Scenarios**

The environment can be configured to run **safety and robustness tests**:

- narrow clearances, moving obstacles, and deceptive configurations,
- unstable stacks or structures to test world-model prediction,
- tasks designed to tempt unsafe shortcuts (e.g., faster but riskier trajectories).

These scenarios feed the safety and metacognition evaluation pipelines, forming part of the **regression and adversarial test suite** described in VII.

3. **Curriculum Feedback from Metacognition and Safety**

Metacognition and safety subsystems can:

- request more practice in specific failure modes,
- mark tasks or configurations as “high-value” for learning,
- temporarily drop tasks that are too hard or too risky until models or policies improve.

This closes the loop between internal learning needs and environment content.

F. Integration with Learning, Memory, and Evaluation

The virtual embodiment environment is the **primary source of data** for:

- **Episodic memory** – logging detailed trajectories, outcomes, and reflections.
- **World model training** – using rollouts and discrepancies between predicted and actual outcomes.
- **Procedural skill discovery** – mining repeated successful behaviors as candidate skills.

Key integration points:

1. **Structured Logging**

Every episode includes:

- environment configuration (task, random seeds, constraints),
- sequences of observations, actions, predictions, and safety interventions,
- success/failure labels and reward signals.

These logs provide ground truth for all downstream learning and analysis.

2. **Controlled Evaluation Modes**

The environment supports distinct modes:

- **Training mode** – full curriculum, resets allowed, heavy exploration and randomized conditions.
- **Validation mode** – fixed seeds and tasks for regression testing, no dynamic curriculum changes.
- **Adversarial mode** – curated difficult scenarios focusing on safety and robustness.

Switching modes is controlled by higher-level orchestration (e.g., experiments, evaluation jobs).

G. Performance, Realism, and Scaling Considerations

The embodiment environment must balance:

- **Realism** – enough fidelity for meaningful learning and safety assessment.
- **Performance** – staying within bandwidth and latency budgets.
- **Scalability** – ability to run many episodes or environments for offline training.

Key design choices:

- Use **configurable fidelity**: lower visual resolution and simplified physics for large-scale training jobs; higher fidelity for safety-critical or final evaluation tasks.
- Allow **multiple environment instances** for offline batch jobs (e.g., MuJoCo or Isaac Gym on HPC nodes), while keeping the main "online body" on the laptop for continuity of identity.
- Monitor and log environment performance (frame times, latency, dropped frames) and feed these metrics into cognitive economics to adjust simulation rate or complexity as needed.

H. Summary

The virtual embodiment environment elevates the architecture from "purely symbolic, prompt-driven AI" to a **sensorimotor AGI substrate**:

- It supplies **continuous, physics-based experience** that grounds planning, perception, and safety in real consequences.
- It provides a **programmable curriculum and adversarial test bed** for lifelong learning and safety validation.
- It interfaces cleanly with RH, safety, metacognition, and memory via well-defined sensor and actuator APIs.
- It is engineered to respect **performance constraints and operational isolation**, making it practical to deploy alongside HPC cognition.

In combination with the dual-hemisphere architecture, distributed memory, and safety/metacognitive layers, the virtual environment forms the **experiential backbone** of the proposed AGI system.

XI. Sensorimotor Loop Architecture

The sensorimotor loop connects **perception, cognition, and action** into a continuous, closed feedback cycle. It is where the dual-hemisphere architecture, safety stack, metacognition, and memory systems are all exercised in real time. Rather than a single global loop, the system runs

multiple overlapping, event-driven loops at different frequencies, coordinated by the event fabric (and optionally the DHT).

A. High-Level Loop Structure

At a conceptual level, each control cycle proceeds through the following stages:

1. **Sense**
    - The virtual environment emits sensor data (RGB-D, object metadata, proprioception, contacts).
    - The RH ingests and encodes this into scene embeddings, state estimates, and events (perception.state_update).
2. **Interpret & Recall**
    - LH and memory subsystems are notified of state changes and task progress.
    - LH queries semantic/episodic/procedural memory for relevant facts, episodes, and skills.
3. **Plan & Hypothesize**
    - LH constructs or refines a plan step or short-horizon policy using reasoning and available skills.
    - Metacognition inspects the reasoning and plan structure, annotating assumptions and confidence.
4. **Simulate & Predict**
    - RH's world model simulates candidate actions/trajectories for 5–20 steps ahead (receding horizon).
    - Safety modules compute risk scores and check constraint violations.
5. **Decide & Approve**
    - LH, RH, safety, and metacognition jointly determine whether a candidate plan segment is:
        - accepted,
        - revised (with added constraints), or
        - rejected (unsafe or incoherent).
6. **Act**
    - Approved trajectories are executed by RH via the actuator API, passing through runtime safety guards.
    - The environment evolves, producing new sensor readings.
7. **Learn & Update**

- The full cycle (state, plan, predictions, outcomes, safety events, reflections) is logged to episodic memory.
- Semantic and procedural memories are updated over time from these logs.

This loop repeats continuously at frequencies determined by hardware limits, cognitive economics, and task requirements.

B. Timing, Frequencies, and Concurrency

The architecture explicitly separates **fast sensorimotor loops** from **slower deliberative and learning processes**:

1. **Environment and RH Loop (Fast Path)**
   - Environment tick: typically 30–120 Hz, configurable.
   - RH perception and control loop:
     - processes new sensor frames, updates state, checks safety, and optionally issues low-level control adjustments at the same or slightly lower rate.
2. **LH Planning and Metacognition (Slower, Event-Driven)**
   - LH does not rebuild full plans every frame; instead it:
     - reacts on meaningful events (subgoal completion, state mismatch, safety alerts, goal changes),
     - maintains a rolling plan for the next several control windows.
   - Metacognitive passes are triggered on:
     - critical decisions, low-confidence plans, or safety conflicts,
     - post-incident reviews, or scheduled reflection slots.
3. **Learning and Consolidation (Slow, Background)**
   - Semantic consolidation, skill discovery, and model retraining run as background or batch jobs, decoupled from real-time control but fed by episodic logs.

Concurrency is managed by the event fabric and cognitive economics:

- Safety-critical and control-related tasks have **pre-emptive priority** over reflection and background learning.
- Metacognition and planning operate under strict time budgets so they cannot stall the fast sensorimotor loop.

C. Detailed Control Cycle Example

A single “macro-step” of the loop can be illustrated as follows:

1. **Perception Update (RH)**
    - Environment sends a new frame + metadata.
    - RH encodes it, updates the scene graph and state tracker, and publishes perception.state_update and, if relevant, perception.state_mismatch (e.g., object not where expected).
2. **Plan Maintenance (LH)**
    - LH receives the update and checks whether:
        - the current plan remains valid (preconditions still hold),
        - any subgoal has been completed or invalidated.
    - If the plan is still valid, LH may simply mark progress and avoid recomputing from scratch.
    - If invalid or incomplete, LH generates or revises a plan segment, possibly reusing skills from procedural memory.
3. **Simulation & Safety Check (RH + Safety)**
    - LH sends simulation.request with one or more candidate plan segments.
    - RH runs short-horizon rollouts, producing simulation.result with:
        - predicted state trajectories,
        - per-step risk scores,
        - any predicted constraint violations.
    - The in-action safety layer evaluates these results:
        - high risk → raise trajectory.risk_high, veto this segment,
        - moderate risk → require plan adjustments (slower motion, extra clearance),
        - low risk → approve.
4. **Metacognitive Review (LH)**
    - For critical or novel actions, metacognition checks:
        - whether the plan is logically coherent,
        - whether memory and perception support the assumptions,
        - whether confidence is commensurate with observed risk and history.
    - Within its time budget, it may suggest re-planning or adding safety margins.
5. **Decision and Execution (LH + RH)**
    - If a plan segment passes safety and metacognition (or budgets expire), LH selects the best available option.

- RH translates the high-level step into trajectories and sends actuator commands through the safety firewall.
- Environment state changes accordingly; contacts and other events are streamed back.

6. **Logging and Learning Hooks**
   - The entire cycle is logged as one or more step entries in episodic memory, including:
     - input observations, actions, predictions, safety decisions, and final outcomes,
     - metacognitive annotations and confidence scores.
   - These logs later feed world-model updates, skill learning, and safety rule refinement.

This macro-cycle may span multiple environment ticks but is constrained by deadlines appropriate to the task (e.g., tens of milliseconds for reflexive adjustments, hundreds of milliseconds for complex re-planning).

D. Asynchrony, Event Ordering, and Robustness

Because cognition is distributed, the sensorimotor loop is **asynchronous and partially ordered**:

- The event fabric guarantees **per-topic FIFO** ordering from each publisher but no global total ordering.
- Events carry **trace IDs**, timestamps, and sequence counters, enabling subsystems to reconstruct causal chains where necessary.

Robustness mechanisms include:

1. **Graceful Degradation**
   - If LH or metacognition is slow or overloaded, RH maintains minimal safe control (e.g., holding position, slowing movement) rather than acting blindly.
   - If RH becomes overloaded, simulation fidelity or rate is reduced, but hard safety limits (joint bounds, contact thresholds) are still enforced.
2. **Timeouts and Fallbacks**
   - Simulation and planning requests have deadlines; if results are not returned in time, LH either:
     - reuses the last known-safe policy,
     - chooses a simpler, conservative action, or
     - requests human input in supervised modes.
3. **Failure Handling**

- If the environment connection drops, an env.disconnected event triggers:
    - an immediate halt or safe pose request,
    - logging of the failure,
    - optional retry or manual intervention.

4. **Loop and Storm Detection**
    - Repeated rapid re-planning or oscillatory behavior (e.g., plan ↔ veto cycles) are detected via trace analysis (coordination.loop_detected) and handled by:
        - increasing safety margins,
        - simplifying the task, or
        - pausing and escalating to a human operator.

E. Interaction with Memory, Safety, and Metacognition

The sensorimotor loop is the integration point for all major subsystems:

- **Memory**
    - Provides context and priors during interpretation and planning.
    - Receives richly annotated episodes for later learning and debugging.
- **Safety**
    - Pre-action checks filter candidate actions before simulation/execution.
    - In-action checks monitor actual trajectories and intervene as needed.
    - Post-action analysis labels episodes with incident and near-miss data for future prevention.
- **Metacognition**
    - Monitors the loop for low-confidence reasoning, inconsistency, and recurring patterns of failure.
    - Decides when deeper reflection, re-planning, or curriculum changes are warranted.

F. Summary

The sensorimotor loop architecture turns the system from "models in isolation" into a **coherent, embodied agent**:

- RH maintains a fast, safety-aware control loop grounded in continuous perception.
- LH provides higher-level goals, plans, and introspective oversight, without blocking real-time safety and control.

- Memory, safety, and metacognition are woven into each cycle, ensuring that every action is both **grounded in current reality** and **informed by past experience**.

This design supports both **fast reflexes** and **slow deliberation**, making it suitable for long-horizon, safety-critical AGI research in rich virtual environments.

XII. HPC Deployment Architecture

The deployment architecture is designed to run on **real, shared academic HPC infrastructure**, not an imaginary bespoke supercomputer. In the current development and testing phase, all experiments are constrained by the resources and policies of the **San José State University College of Engineering (CoE) HPC cluster**. San José State University

This section explains how the AGI architecture maps onto that cluster, how we respect its constraints (SLURM scheduling, wall-clock limits, shared GPUs/storage), and how the design can later be lifted to larger or dedicated HPC systems with minimal changes.

A. Target Platform: SJSU CoE HPC Cluster

The SJSU CoE HPC system is a Linux distributed cluster composed of: San José State University

- **44 nodes** with Intel Xeon processors (multi-core, multi-socket), providing **~1,232 compute cores** and ~100 TFLOP/s peak.
- A mix of **NVIDIA GPUs** across nodes:
    - 18 × A100, 5 × H100, 1 × A40, 3 × V100, 17 × P100.
- **Memory:** 20 compute nodes with 128 GB RAM; 24 GPU/condo nodes with 256 GB RAM (total ~8.7 TB).
- **High-speed interconnect:** very high-speed **InfiniBand network** tying nodes together.
- **Storage:**
    - 524 TB **Lustre scratch filesystem** for high-performance I/O,
    - shared /home and /data each with 100 TB across nodes,
    - SSDs on all GPU/condo nodes for fast local I/O.

Access is via SSH to login nodes, with computation scheduled exclusively through **SLURM**, including batch (sbatch) and interactive jobs. GPU nodes have **48-hour max wall-clock limits**, CPU nodes 24 hours; condo nodes may have no time limits. San José State University

Implication: the architecture must treat the cluster as a **pooled, time-bounded, shared resource**, not as a set of permanent, dedicated servers.

B. Logical Deployment Topology on SJSU HPC

The AGI system is decomposed into services that map onto the cluster as follows:

1. **Left Hemisphere (LH) – H100/A100 Node**
    - Runs the large language–based reasoning and planning stack.
    - Deployed as a **Singularity/Apptainer container** on a single H100 (preferred) or A100 GPU node with 256 GB RAM.
    - Uses the node's local SSD for KV cache and temporary model state; logs to Lustre scratch or /data.
2. **Right Hemisphere (RH) – A100 Node**
    - Runs multimodal perception, world-model simulation interface, and trajectory planning.
    - Deployed as a second container on a separate A100 node.
    - Shares InfiniBand network with LH and memory nodes for low-latency coordination.
3. **Memory Nodes – CPU / GPU-less Nodes**
    - Semantic, episodic, and procedural memory services run on **CPU-only compute nodes** with 128 GB or 256 GB RAM.
    - These nodes host the vector DB, episodic log indexers, and procedural skill registry.
    - Heavy I/O goes to Lustre scratch when training or replaying episodes; longer-term state and models live under /data. San José State University
4. **Coordination Layer – Shared Across Nodes**
    - The **event fabric** uses UCX or MPI-based messaging over InfiniBand.
    - The **DHT coordination layer** (when enabled) runs as a thin daemon on the same nodes as memory or small sidecar processes, using the same network fabric.
5. **Virtual Environment – Off-cluster**
    - The Unity/MuJoCo environment runs on a **separate gaming laptop**, not on the HPC.
    - RH communicates with it over the campus network/VPN via a dedicated API (WebSocket/gRPC), isolating simulation instability from cluster jobs.

C. Containers, Images, and Runtime Environment

The cluster does **not** run Docker daemons on compute nodes; instead, it expects user workloads to run in user-space via tools like **Singularity/Apptainer**. San José State University

Deployment pattern:

- Each major component (LH, RH, memory, coordination) is packaged as a **versioned Singularity .sif image**.
- Images are stored in /home or /data and referenced from SLURM job scripts.
- A typical LH job script:

```
#!/bin/bash

#SBATCH --job-name=agi-lh

#SBATCH --nodes=1

#SBATCH --ntasks=1

#SBATCH --gres=gpu:H100:1      # or A100:1, depending on partition

#SBATCH --cpus-per-task=8

#SBATCH --mem=200G

#SBATCH --time=24:00:00        # within 48h GPU limit

#SBATCH --output=agi-lh.%j.log


module load apptainer

srun apptainer exec /data/agi_images/lh_latest.sif \

    python -m agi.lh_service --config /data/agi_configs/lh_config.yaml
```

- Similar scripts start RH and memory services, with node constraints and time limits set according to SJSU HPC policies. San José State University

---

D. SLURM Scheduling Model and Long-lived Services

A key constraint is that **GPU jobs are time-limited** (typically 48 hours maximum), and the cluster is a shared resource with fair-use scheduling. San José State University

To reconcile this with the "long-lived microservice" design, we treat the current SJSU deployment as **experiment-oriented sessions**, not perpetual daemons:

1. **Session-based AGI Runs**

- LH, RH, and memory services are launched together (or in a controlled sequence) as a **stack of SLURM jobs** with aligned wall-clock limits (e.g., 8–24 hours).
- During a session, they behave as if they are long-lived services.

2. **Checkpointing and Restart**
   - All critical state (models, memory, DHT contents, configuration, logs) is stored on persistent storage (/data or Lustre).
   - Before job expiry, services checkpoint their state:
     - current task agenda,
     - pending episodes,
     - any in-memory DHT "hot" keys.
   - New sessions can restart from these checkpoints, giving the agent **lifespan continuity** across SLURM jobs even though the underlying processes are periodically torn down.
3. **Condo Nodes for Extended Experiments**
   - Where available (e.g., condo nodes without time limits), we can run **longer experiments** such as multi-day world-model training or large-scale replay, still respecting fair-use agreements.
4. **No Heavy Work on Login Nodes**
   - All computational work runs under SLURM; login nodes are used only for code editing, compilation, and job submission, per SJSU guidelines. San José State University

---

E. Storage Layout and Data Management

Given the Lustre scratch and shared /home / /data volumes, we propose the following layout: San José State University

- **/home/<user>/agi/**
  - Source code repositories, small configs, build scripts.
- **/data/agi/models/**
  - Pretrained LLM weights, world-model checkpoints, skill policies.
- **/data/agi/state/**
  - Semantic/episodic/procedural memory snapshots.

    - DHT metadata, safety rule versions, calibration parameters.
- **/scratch/agi/<jobid>/ (Lustre)**
    - High-volume episodic logs, temporary tensors, training data shards.
    - Short-lived caches for batch jobs.

Policies:

- Long-term state must live in /data so it survives scratch purges.
- Large replay datasets and intermediate training artifacts reside on Lustre scratch and can be regenerated if needed.
- Job-specific outputs are written to scratch and then summarized/compacted into /data during consolidation jobs.

F. Network and Coordination Considerations

The CoE HPC provides an **InfiniBand interconnect**, which is well-aligned with our need for low-latency, high-bandwidth communication between LH/RH and memory nodes. San José State University

Deployment choices:

- Event fabric implemented on top of UCX, MPI, or a high-performance messaging layer that can exploit InfiniBand.
- DHT nodes co-located with memory services to minimize extra hops.
- RH ↔ virtual environment communication runs over standard TCP on VPN; this is higher latency than intra-cluster traffic but acceptable because the environment is on a separate machine by design.

We assume **no direct inbound connections from the public internet** to compute nodes; any external APIs (e.g., for human supervision dashboards) are proxied via login or gateway nodes in compliance with SJSU policies.

G. Development vs. Future Production Deployment

In this phase, the architecture is intentionally scoped to **fit within SJSU CoE HPC constraints**:

- Limited number of H100/A100 nodes and fair-share scheduling.
- 24–48 hour wall-clock limits per job.
- Shared GPU and storage resources with other students and faculty. San José State University

As a result:

- We focus on **dual-hemisphere + memory + single environment** configurations, not large multi-agent swarms.
- We treat the system as a series of **reproducible experiments** (sessions with well-defined start/end and checkpoint boundaries) rather than an always-on AGI service.
- We lean heavily on **offline batch jobs** for heavy learning (world-model retraining, large skill updates) to avoid monopolizing GPUs.

The same logical architecture is, however, designed to be **portable**:

- On a larger or dedicated cluster, LH, RH, memory, safety, DHT, and monitoring can each scale horizontally across more nodes.
- On cloud/HPC hybrids, LH and RH could use cloud GPUs while memory and coordination remain on on-prem nodes, as long as latency budgets are respected.

H. Summary

The HPC deployment architecture:

- Maps the AGI's dual-hemisphere, memory, and coordination layers onto the **real SJSU CoE HPC cluster** with its Intel Xeon nodes, mixed GPU pool (A100/H100/etc.), InfiniBand network, and Lustre storage. San José State University
- Uses **Singularity/Apptainer containers** and **SLURM** as the operational backbone.
- Respects **time limits and fair-share policies** by structuring AGI runs as checkpointed sessions rather than immortal processes.
- Leverages persistent /data storage and Lustre scratch to preserve cognitive state and logs across jobs.
- Remains scalable and portable to more capable clusters or cloud environments in later phases.

In short, the AGI system is engineered to run **inside the very real constraints of SJSU's HPC**, while still embodying the architectural principles needed for future, larger-scale deployments.

D. SLURM Job Topology

A typical deployment requires:

```
sbatch left_hemisphere.sbatch
sbatch right_hemisphere.sbatch
```

```
sbatch semantic_memory.sbatch
sbatch episodic_memory.sbatch
sbatch procedural_memory.sbatch
```

Optional:

```
sbatch dht_coordinator.sbatch
sbatch simulation_batch.sbatch
```

Jobs connect through SSH tunnels or cluster-internal networking fabric.

XIII. Networking & RDMA Fabric

The networking and RDMA fabric is the **artificial nervous system** of the AGI: it carries perception, plans, embeddings, safety signals, and memory queries between hemispheres, memory nodes, and coordination layers. The design must deliver **low latency, high throughput, and predictable behavior** while operating within the constraints of the SJSU CoE HPC cluster's InfiniBand interconnect.

A. Physical Substrate: InfiniBand + UCX on SJSU HPC

The SJSU CoE HPC cluster provides a **high-speed InfiniBand network** tying together CPU and GPU nodes, with Lustre-backed storage and shared /home and /data volumes.

We exploit this with:

- **UCX (Unified Communication X)** as the primary transport for intra-cluster messaging.
- **RDMA (Remote Direct Memory Access)** for zero-copy transfers of tensors, embeddings, and scene/state blobs between LH, RH, and memory nodes.

Goals:

- **Latency**: single-hop messaging between hemispheres and memory nodes in tens of microseconds under normal load.
- **Throughput**: sustained multi-Gb/s bandwidth for episodic logging, model updates, and batch memory operations.
- **CPU offload**: RDMA reduces CPU overhead, leaving cores free for model inference and memory indexing.

B. Logical Topology

The logical topology overlays on the physical InfiniBand fabric as follows:

1. **Core Nodes**

- **LH node** (H100/A100): runs LLM-based planning, metacognition, and pre-action safety.
- **RH node** (A100): runs perception, world-model interface, trajectory planning, and in-action safety.
- **Memory nodes** (CPU-only): host semantic, episodic, and procedural memory services.

2. **Coordination Overlay**
    - **Event fabric**: high-speed pub/sub channels over UCX between LH, RH, and memory nodes.
    - **DHT overlay** (optional): a logical key–value mesh spanning (typically) the memory nodes and low-footprint daemons on LH/RH, sharing the same InfiniBand network.
3. **Off-cluster link to Virtual Environment**
    - RH communicates with the Unity/MuJoCo environment via standard TCP (WebSocket/gRPC) over the campus network/VPN, accepting higher latency because the environment is intentionally off-cluster.

The design assumes **no direct internet ingress** to compute nodes; all remote access is via login/gateway nodes and campus networking, in line with SJSU policies.

C. Messaging Stack

The networking stack is organized in three tiers:

Transport Tier – UCX over InfiniBand

- Provides RDMA read/write, send/recv, and zero-copy semantics.
- Handles connection setup, adaptive routing, and low-level congestion control.

Messaging Tier – Event Fabric and RPC

- **Event fabric**:
    - A lightweight pub/sub layer (e.g., custom UCX-based broker or ZeroMQ built with UCX transports) for events like:
        - perception.state_update, plan.step_ready, simulation.result, safety.flag, memory.updated.
    - Guarantees **per-publisher, per-topic FIFO** ordering but no global total order.
- **RPC (gRPC/REST)**:
    - Used for request/response operations (e.g., memory.semantic.query, world_model.simulate, safety.check_plan).
    - Payloads serialized via Protobuf or FlatBuffers.

State Tier – DHT (Advanced Mode)

- Distributed key–value store over the same network fabric, used for:
    - long-lived cognitive state (plan.graph.current, scene.graph.current),
    - indices (episodic.index, skill.catalog),
    - coordination metadata.
- Implements **eventual consistency with version vectors**, as described in Section V; clients resolve conflicts with type-specific merge policies.

D. Data Types, Sizes, and Performance Budgets

Different communication paths have different requirements:

1. **Fast-Path Control & Safety Signals**
    - Examples: emergency_stop, trajectory.risk_high, goal.infeasible.
    - Payload size: tens to hundreds of bytes (metadata, IDs).
    - Target latency: **sub-millisecond** end-to-end under normal cluster load.
    - Transport: small UCX messages via event fabric.
2. **Embeddings and Plan Graphs**
    - Semantic embeddings (1–4k float32), plan graphs, skill descriptors.
    - Payload size: 4–32 KB typical.
    - Target latency: < 1–2 ms; these are frequent but not hard real-time.
    - Transport: UCX send/recv or RPC.
3. **Scene Graphs and World-Model Rollouts**
    - Scene graphs, simulation traces, multi-step predictions.
    - Payload size: tens to hundreds of KB per exchange.
    - Target latency: bounded by planning budgets (e.g., tens of ms); throughput is more critical than ultra-low latency.
    - Transport: batched UCX transfers, optionally using RDMA for larger blobs.
4. **Episodic Logs and Training Data**
    - Streaming logs to episodic memory and training jobs.
    - High volume, but tolerant of higher latency.
    - Transport: bulk transfers over UCX with write-back to Lustre; can be batched and compressed.

Budgets are enforced via cognitive economics: if network/load metrics approach thresholds, low-priority transfers (e.g., background logging) are throttled before fast-path control signals.

E. Reliability, Ordering, and Fault Handling

The networking layer must survive transient failures, congestion, and partial node outages:

1. **Ordering and Idempotence**
    - **Per-topic FIFO** from a given publisher is guaranteed by the event fabric; subscribers reconstruct causal chains using:
        - trace IDs,
        - per-publisher sequence numbers,
        - timestamps.
    - Idempotent handlers are preferred: repeated or delayed messages are safe.
2. **Heartbeats and Health Checks**
    - LH, RH, and memory services send periodic heartbeats via a health.* topic.
    - Missing heartbeats within a window trigger:
        - local degradation (e.g., RH holds pose if LH is unreachable),
        - logging and notification,
        - optional re-routing of responsibilities if redundancy exists.
3. **Retry and Backoff**
    - RPC calls implement bounded retry with exponential backoff for transient failures.
    - If a critical service is unavailable, clients fall back to safe defaults (e.g., conservative policies, read-only memory access).
4. **Network Partition Handling (DHT)**
    - In case of partial partitions, the DHT continues operation within each partition using **eventual consistency**; when connectivity returns, version vectors drive reconciliation.
    - Safety-critical keys (e.g., safety rules, emergency overrides) can be replicated more aggressively or pinned to specific nodes to reduce ambiguity.

F. World → HPC Data Pipeline

The link from the virtual environment (laptop) to the RH on the cluster is separate from InfiniBand and uses standard campus networking:

1. **Sensor Data Path**
    - Environment → RH:

        - RGB frames + depth/metadata compressed and sent over WebSocket/gRPC.
        - RH performs decoding and pre-processing, then passes embeddings and state to other subsystems over UCX.

2. **Action Path**
    - RH → Environment:
        - Actuator commands sent via a low-latency, authenticated TCP channel.
        - In-action safety checks run **before** the commands leave the cluster.

3. **Bandwidth and Latency Management**
    - Environment frame rate and resolution are tuned so that network round-trips stay within acceptable bounds for the control loop (tens of ms).
    - If campus network conditions degrade, RH automatically:
        - reduces frame rate or resolution,
        - increases control safety margins,
        - and, in worst case, transitions to a safe, "wait" posture until conditions improve.

G. Observability and Network-Level Monitoring

To manage such a distributed system, network-level observability is essential:

- **Metrics collection** (Prometheus/OpenTelemetry):
    - per-link latency and throughput,
    - message queue lengths,
    - UCX error rates,
    - DHT update/lookup latency.
- **Dashboards** (Grafana):
    - show cross-hemisphere latency, memory query times, event fabric traffic, and network saturation.
- **Trace-based diagnostics**:
    - Each significant request/response pair and event chain carries a trace ID.
    - Traces sampled to a logging backend enable debugging of slow decisions, bottlenecks, and coordination loops.

These tools allow tuning of RDMA parameters (e.g., queue depths, message sizes) and cognitive economics policies (e.g., throttling low-value traffic under congestion).

H. Summary

The networking and RDMA fabric turns the SJSU HPC cluster into a **coherent substrate for distributed cognition**:

- UCX over InfiniBand provides low-latency, high-throughput transport compatible with the dual-hemisphere and memory architecture.
- The event fabric and optional DHT layer share this substrate, balancing **real-time reactivity** with **persistent, shared cognitive state**.
- Performance budgets, ordering guarantees, and fault-handling strategies make communication **predictable and debuggable**, even under load or partial failure.
- A separate campus network link to the virtual environment completes the loop, delivering a practical, safety-aware sensorimotor pipeline within the constraints of the SJSU HPC environment.

XIV. API Interfaces & Versioning

The API layer is what makes the architecture *engineerable*: it turns an abstract dual-hemisphere, multi-memory cognitive system into a set of **typed, versioned contracts** that services can rely on over time. This section describes the internal APIs between hemispheres, memory, safety, metacognition, and the virtual environment, as well as how these interfaces evolve without breaking the system.

A. Design Principles

APIs in this architecture follow a few core principles:

1. **Strongly typed and schema-driven**
    - All messages and RPCs are defined in an IDL (e.g., Protobuf or FlatBuffers).
    - No “raw JSON blobs” in hot paths; JSON is reserved for debug tooling and logs.
2. **Explicit versioning**
    - Every API has a **semantic version** (MAJOR.MINOR.PATCH).
    - Backwards compatibility rules are clear and enforced by code generation and contract tests.
3. **Error-aware and time-bounded**
    - All calls define timeouts and error codes.
    - Clients must handle failures gracefully (degrade, fallback, or escalate).
4. **Observable and testable**
    - APIs emit structured logs and metrics.

- Contract tests and regression suites catch breakage before deployment.

B. Internal Service APIs

The system exposes several *internal* APIs between core components.

B.1 LH ↔ RH Cognitive APIs

These APIs are used for planning, simulation, and control. Typical RPCs:

- PlanSegmentRequest (LH → RH)
  - Contains:
    - high-level action description,
    - initial state reference,
    - constraints (time, force, regions to avoid),
    - optional candidate trajectories.
- SimulationResult (RH → LH)
  - Contains:
    - predicted state trajectories,
    - per-step risk scores,
    - constraint violations,
    - suggested modifications (e.g., slower speed, detours).
- ControlRequest (LH/RH → RH control loop)
  - Contains:
    - validated plan segment or skill invocation,
    - parameters (target poses, waypoints),
    - priority and safety margins.
- ControlStatus (RH → LH)
  - Contains:
    - execution state (running, complete, error),
    - any safety.flag or emergency_stop triggers,
    - final pose/outcome.

These APIs are performance-critical and run over UCX/RDMA using compact binary encodings.

B.2 Hemispheres ↔ Memory APIs

Memory APIs are request/response oriented:

- SemanticQuery, SemanticResult
    - Vector + filter queries with pagination; returns entries with embeddings, metadata, and version IDs.
- EpisodicAppend, EpisodicQuery
    - Append step/episode records; retrieve episodes by time, events, or semantic filters.
- SkillGet, SkillPut, SkillSearch
    - Fetch/insert skills; search skill catalog by tags, preconditions, and success statistics.

These APIs guarantee **read-your-writes** behavior per client for consistency.

B.3 Safety & Metacognition APIs

Safety and metacognition expose well-defined services to other components:

- SafetyCheckPlan
    - Input: plan graph + context.
    - Output: risk classification, flagged steps, suggested constraints, hallucination markers.
- SafetyCheckAction
    - Input: low-level action command.
    - Output: allowed/denied, adjusted parameters, reason codes.
- MetaReview
    - Input: reasoning trace, plan, simulation results, memory hits.
    - Output: confidence vector, detected issues, recommended actions (accept/revise/reject).

These APIs make safety and metacognition **reusable services** rather than hard-coded library calls.

B.4 Environment API

The virtual environment API is intentionally simple and stable:

- StepEnv (RH → Env)

    - Contains: action command(s), optional control horizon.
- StepResult (Env → RH)
    - Contains: observations (RGB, depth, metadata, proprioception), reward/task signals, termination flags.

These messages are versioned separately from HPC-internal APIs, since they cross the campus network and may have different deployment cadence.

C. Schema Definition and Evolution

All APIs are defined in a central **schema repository** (e.g., agi/apis/*.proto):

- Every message and RPC is annotated with:
    - since_version, deprecated_since, and comments on migration paths.
- Code for LH, RH, memory, safety, metacognition, and environment adapters is generated from the same IDL.

Backward-compatible changes allowed in MINOR versions:

- adding new optional fields with defaults,
- adding new RPCs,
- expanding enum values (with appropriate unknown handling).

Breaking changes (e.g., removing fields, changing semantics) require a MAJOR version bump and a migration period where both old and new versions coexist.

Migration pattern:

- Services implement **multi-version handlers**:
    - can speak both v1 and v2 for a transition window,
    - log usage of deprecated versions,
    - eventually drop support when all clients have moved.

D. Versioning Strategy and Namespacing

APIs and data are versioned at multiple layers:

1. **Service API Versioning**
    - Each service (LH, RH, memory, safety, env) exposes a versioned endpoint, e.g.:
        - agi.lh.v1.PlanService, agi.memory.v2.SemanticService.

- Clients negotiate or are configured to a specific supported version.

2. **Event and Topic Versioning**
    - Event names include a version suffix when payloads are not backward compatible, e.g.:
        - perception.state_update.v1, plan.step_ready.v2.
    - New consumers subscribe to the new version; old consumers can continue to listen to the old version until removed.
3. **DHT Key Namespacing**
    - Keys are namespaced by component and version, e.g.:
        - lh/v1/plan_graph/current,
        - memory/v2/episodic/index.
    - This avoids collisions and allows gradual migration of shared state representations.
1. **Model & Policy Versioning**
    - Models (LLM, world model, skills) carry their own semantic versions.
    - API responses include model versions used to make a decision, aiding debugging and regression testing.

E. Error Handling, Timeouts, and Retries

All APIs define **standard error codes** and behaviors:

- **Error classes**:
    - INVALID_ARGUMENT, NOT_FOUND, PERMISSION_DENIED,
    - UNAVAILABLE, DEADLINE_EXCEEDED, INTERNAL, UNIMPLEMENTED.
- **Timeouts**:
    - LH ↔ RH simulation calls: short timeouts (tens to hundreds of ms), aligned with planning budgets.
    - Memory queries: moderate timeouts; soft failures can fall back to smaller context or cached results.
    - Learning and batch jobs: longer timeouts, but running under SLURM wall-clock limits.
- **Retry policies**:

- Transient errors (UNAVAILABLE, DEADLINE_EXCEEDED) may be retried with exponential backoff.
- Permanent errors (INVALID_ARGUMENT, PERMISSION_DENIED) are not retried; clients must adjust or escalate.

Error responses include:

- machine-readable codes,
- human-readable messages,
- optional remediation hints (e.g., “reduce batch size,” “refresh skill catalog”).

Metacognition treats persistent errors as signals:

- repeated NOT_FOUND for tools → possible hallucinated tools or outdated docs,
- frequent timeouts → adjust planning/simulation budgets or resource allocation.

F. Testing, Validation, and Tooling

To prevent API drift from silently breaking cognition, the architecture includes:

1. **Contract Tests**
   - For each API, a suite of tests ensures:
     - required fields are present,
     - default values and optional fields behave as expected,
     - old clients still function against new servers (and vice versa where supported).
2. **Schema Linting and CI**
   - Changes to IDL files trigger CI checks:
     - compatibility analysis (breaking vs non-breaking changes),
     - regeneration of stubs,
     - re-running contract tests and key integration tests.
3. **Replay-based Validation**
   - Logged episodes (requests + responses) are replayed against new versions of services to detect behavior regressions.
4. **Documentation and Introspection**
   - API documentation is generated from IDL comments (e.g., via protoc-gen-doc).

- A small “introspection service” exposes the current API and model versions in use across LH, RH, memory, and safety nodes for debugging.

G. External and Human-Facing APIs (Future Work)

While the initial deployment focuses on internal APIs, the same versioning and schema discipline naturally extend to:

- **Operator dashboards** (e.g., monitoring, inspection of plans and memory),
- **Supervision interfaces** (e.g., human-in-the-loop approval of high-risk actions),
- **Research APIs** (e.g., experiment configuration, curriculum control).

These will be defined as **separate API families** (e.g., agi.admin.*) with stricter authentication and rate limits, but they share the same underlying versioning and contract-testing approach.

H. Summary

The API Interfaces & Versioning layer ensures that the AGI architecture is:

- **composable** – clear, typed contracts between LH, RH, memory, safety, and the environment;
- **evolvable** – semantic versioning, multi-version support, and migration paths;
- **robust** – explicit error handling, timeouts, and retries;
- **testable** – schema-driven CI, contract tests, and replay-based regression checks.

This discipline lets the system grow—from a constrained SJSU HPC prototype to larger polycentric deployments—without collapsing under interface drift or brittle, ad-hoc integrations.

XV. Implementation Roadmap

This roadmap turns the architecture into a concrete, staged build plan that fits within the **SJSU CoE HPC constraints** (SLURM, shared GPUs, 24–48h jobs) and keeps the project shippable at each step. It also includes a proposed **git repo layout** to keep the codebase sane as it grows.

A. Roadmap Goals

Across phases, we’re aiming for:

1. **Always-runnable prototypes** – each phase ends with something you can demo or regression-test.

2. **HPC-aware design** – every step is runnable as SLURM jobs on the SJSU cluster.
3. **Safety & observability from early on** – logging, metrics, and basic safety checks are never bolted on at the very end.
4. **Modularity** – LH, RH, memory, safety, metacognition, and environment stay cleanly separated.

I'll describe the phases as "Phase 0, 1, 2…" rather than time-based, so you can pace them however you like.

B. Phase 0 – Repo & Infrastructure Skeleton

Goal: Stand up a coherent mono-repo and basic infra so future phases aren't chaos.

Scope:

- Initialize primary repo (call it e.g. agi-hpc).
- Define **core package structure**, API schemas, and config format.
- Add basic CI: linting, type checks, unit tests.
- Add **Singularity/Apptainer build scripts** and **SLURM job templates**.
- Stub out minimal "hello world" services for LH, RH, and memory that just ping each other.

Exit criteria:

- git clone && make dev && make test works locally.
- sbatch templates can run a trivial multi-node "ping" example on SJSU HPC.
- High-level docs: README.md, docs/ARCHITECTURE_OVERVIEW.md, docs/HPC_DEPLOYMENT.md.

C. Phase 1 – Single-Node Cognitive Core (Dev Laptop / Single GPU)

Goal: Get a **minimal AGI loop** running on a single machine, no HPC yet: LH + stub RH + in-process memory.

Scope:

- LH with:
    - LLM integration (local or API-backed).
    - Simple planner (text plans → environment commands).

    - Very basic metacognition (self-critique of plans).
- RH stub:
    - Mock perception and “physics” (e.g., simple state machine or 2D grid).
    - No real GPU vision yet, just structured JSON state.
- Memory:
    - In-process vector store (semantic) + simple episodic log.
- Safety v0:
    - Hard-coded pre-action filters (blacklist obvious unsafe ops in plans).
- Simple CLI or notebook loop to send tasks and watch the loop.

Exit criteria:

- On a single machine, you can say “move object A to B in a toy grid world” and see a sense–plan–act–log cycle complete successfully.
- Memory writes/reads and very basic reflection are happening.

D. Phase 2 – Split LH/RH Across Nodes with Event Fabric

Goal: Actually **split LH and RH into separate services** talking over the fabric on the SJSU cluster.

Scope:

- Implement event fabric (UCX/ZeroMQ/gRPC) with:
    - plan.step_ready, simulation.request, simulation.result, perception.state_update, safety.flag.
- Deploy:
    - LH on one GPU node (H100/A100).
    - RH on another GPU node (or CPU node if vision is still mocked).
- Memory v1:
    - Semantic memory as a separate service on a CPU node (vector DB or simple FAISS/Qdrant instance).
    - Episodic log service that accepts append calls from LH/RH.
- HPC integration:
    - SLURM job scripts to start LH, RH, memory in the correct order and wire them via hostnames/ports/UCX endpoints.

Exit criteria:

- You can launch a **multi-job experiment** on SJSU HPC where LH and RH exchange events and complete a simple scenario.
- Event fabric metrics + logs show stable, ordered communication.

E. Phase 3 – Memory Architecture v1 (Semantic + Episodic)

Goal: Turn memory from "sidecar" into a real **subsystem with policies**.

Scope:

- Implement data models from the revised **Memory Architecture**:
  - Semantic entries with embeddings + tags + timestamps.
  - Episodic episodes: sequences of (obs, action, prediction, outcome, reflection).
- Query APIs:
  - Time-bounded episodic queries.
  - Semantic vector + filter queries.
- Simple management:
  - Retention scores and basic garbage collection (LRU + safety pinning).
- Add memory-backed behaviors:
  - LH uses semantic retrieval consistently when planning.
  - Metacognition consults both memories for confidence estimation.

Exit criteria:

- You can query "similar past episodes" and see LH adjust a plan based on a previous success/failure.
- Memory size doesn't explode under a multi-hour run; GC is running.

F. Phase 4 – Virtual Environment Integration & Sensorimotor Loop

Goal: Connect RH to a **real virtual environment** on the gaming laptop and stand up the full sensorimotor loop.

Scope:

- Choose and integrate engine (Unity or MuJoCo) for a simple manipulation/navigation scenario.

- Define and implement:
    - Sensor API (RGB, metadata, proprioception).
    - Actuator API (high-level + low-level control).
- RH upgrades:
    - Vision encoder (CLIP/DINO-style) on GPU.
    - World-model interface to the environment for short rollouts.
- Sensorimotor loop:
    - Continuous perception → plan → simulate → act cycle.
- Logging:
    - Episodic memory logs full episodes with environment context.

Exit criteria:

- On HPC + laptop, you can run a demo where the agent **perceives** the 3D scene, **plans**, and **successfully completes** basic tasks (e.g., pick-and-place, simple navigation) several times.
- Telemetry dashboards show control-loop timing, RH latency, and network stats.

G. Phase 5 – Safety v1 (Three-Layer Safety Online)

Goal: Implement **pre-action, in-action, post-action** safety as real services with metrics.

Scope:

- Pre-Action:
    - Schema/tool registry + hallucination detection for plans/code.
    - Tunable thresholds; track false positives/negatives for blocks.
- In-Action:
    - Receding-horizon simulation (5–20 step) with risk scores.
    - Runtime safety guards in RH control loop (joint limits, force, emergency stop).
- Post-Action:
    - Label episodes with incidents, near misses.
    - Feed this data into memory and simple rule updates.
- Monitoring:

- Safety dashboard: incidents per hour, emergency stops, near misses, blocked vs allowed actions.

Exit criteria:

- You can run safety-stress scenarios in the env and see safety intervene appropriately.
- Metrics show thresholds are adjustable and don't totally freeze the system.

H. Phase 6 – Metacognition v1

Goal: Lift metacognition from "some checks" to a **structured subsystem** with budgets.

Scope:

- Implement metacognitive API:
  - Input: reasoning trace, plan, sims, memory hits.
  - Output: confidence vector, issues, accept/revise/reject.
- Integrate with LH:
  - LH calls metacognition selectively (critical tasks, low-confidence signals, safety conflicts).
- Implement budgets:
  - Per-call time and compute caps; fallback when exhausted.
- Logging:
  - Store metacognitive annotations in episodic memory.

Exit criteria:

- For a non-trivial task, you can inspect a metacognitive report and see:
  - what it flagged,
  - how it influenced plan selection.

I. Phase 7 – Lifelong Learning Infrastructure

Goal: Add **offline learning jobs** and skill discovery, respecting HPC.

Scope:

- Batch jobs:
  - World-model fine-tuning on episodic data.

    - Semantic consolidation and clustering.
    - Skill discovery pipeline (pattern mining from episodes).
- Procedural memory:
    - Skill promotion/retirement logic.
    - RH and LH APIs to call learned skills.
- Curriculum:
    - Simple curriculum manager in env that can generate scenarios from a difficulty schedule.

Exit criteria:

- After running offline jobs, you can measure:
    - improved performance on a set of tasks,
    - or decreased need for re-planning / reflection in known scenarios.

J. Phase 8 – DHT & Polycentric Extensions (Advanced)

Goal: Introduce DHT-based coordination and allow **more than two major cognitive agents**.

Scope:

- Implement DHT layer:
    - Versioned keys for plan graphs, scene graphs, skill catalogs.
    - Eventual consistency + merge policies.
- Add one or more micro-agents:
    - e.g., a dedicated reflection agent, curriculum agent, or skill miner.
- Move some shared state off the event fabric into the DHT.

Exit criteria:

- You can temporarily kill/restart one micro-agent and the system still converges.
- DHT metrics show stable operation, conflict resolution working.

K. Phase 9 – Hardening, Tooling, and Documentation

Goal: Make the system **usable and inspectable** by other people.

Scope:

- CLI tools for:
    - launching an entire experiment stack,
    - tailing logs,
    - collecting metrics and artifacts.
- Docs:
    - “How to run on SJSU HPC” guide,
    - per-component reference docs,
    - troubleshooting and common failure modes.
- Regression suites:
    - Standard tasks and safety scenarios run in CI/nightly.

Exit criteria:

- New contributor can: clone repo, follow docs, run a small experiment on HPC and a local env without bothering you.

L. Proposed Git Repo Layout

Here’s a suggested mono-repo layout for agi-hpc that matches the architecture:

```
agi-hpc/
├── README.md
├── pyproject.toml          # or setup.cfg/poetry for Python packaging
├── requirements.txt        # base deps (if using pip)
├── .gitignore
├── .editorconfig
├── .pre-commit-config.yaml
├── .github/
│   └── workflows/          # CI: lint, tests, maybe build containers
│
├── docs/
│   ├── ARCHITECTURE_OVERVIEW.md
│   ├── HPC_DEPLOYMENT.md
```

```
│   ├── ENVIRONMENT_SETUP.md
│   ├── API_REFERENCE.md
│   └── ROADMAP.md
│
├── design/
│   ├── whitepaper/         # source for this doc
│   ├── diagrams/           # draw.io / mermaid / plantuml
│   └── specs/              # more detailed design notes
│
├── configs/
│   ├── lh_config.yaml
│   ├── rh_config.yaml
│   ├── memory_config.yaml
│   ├── safety_config.yaml
│   ├── meta_config.yaml
│   └── env_config.yaml
│
├── infra/
│   ├── hpc/
│   │   ├── slurm/
│   │   │   ├── lh_job.sbatch
│   │   │   ├── rh_job.sbatch
│   │   │   ├── memory_job.sbatch
│   │   │   └── stack_launcher.sbatch
│   │   ├── apptainer/
│   │   │   ├── lh.def
│   │   │   ├── rh.def
│   │   │   └── memory.def
│   │   └── monitoring/
```

```
│   │       ├── prometheus.yaml
│   │       └── grafana_dashboards/
│   └── local/
│       └── docker-compose.yml   # optional for local dev
│
├── src/
│   └── agi/
│       ├── common/              # shared utilities, logging, config
│       ├── core/
│       │   ├── events/          # event fabric, messaging abstractions
│       │   ├── dht/             # DHT implementation (when added)
│       │   └── api/             # generated stubs from
proto/flatbuffers
│       ├── lh/                  # Left Hemisphere service
│       │   ├── __init__.py
│       │   ├── service.py
│       │   ├── planning/
│       │   ├── metacognition/
│       │   └── safety_preaction/
│       ├── rh/                  # Right Hemisphere service
│       │   ├── __init__.py
│       │   ├── service.py
│       │   ├── perception/
│       │   ├── world_model/
│       │   └── control/
│       ├── memory/
│       │   ├── __init__.py
│       │   ├── semantic/
│       │   ├── episodic/
│       │   └── procedural/
```

```
│       ├── safety/
│       │   ├── __init__.py
│       │   ├── in_action/
│       │   └── post_action/
│       ├── meta/                # metacognition service (if separate)
│       ├── env_client/          # client for Unity/MuJoCo API
│       ├── learning/            # offline training, skill discovery
│       └── tools/               # CLI tools, experiment runners
│
├── proto/                       # or schemas/
│   ├── lh.proto
│   ├── rh.proto
│   ├── memory.proto
│   ├── safety.proto
│   ├── meta.proto
│   └── env.proto
│
├── tests/
│   ├── unit/
│   ├── integration/
│   └── regression/
│
└── scripts/
    ├── build_sif.sh             # wrappers for Apptainer builds
    ├── submit_stack.sh          # submit full experiment stack to
SLURM
    └── dev_shell.sh             # convenience script for local dev
```

You might also keep the **Unity/MuJoCo project** in a separate repo (e.g. agi-env) with its own lifecycle; src/agi/env_client just speaks its network API.

XVI. Evaluation & Testing Framework

This section provides a practical, engineering-oriented breakdown of how the architecture is implemented as **real services**, **real containers**, and **real SLURM jobs** on the SJSU CoE HPC cluster, talking to a **real virtual environment** running on a separate machine.

The goal is to make the system **buildable and debuggable**, not just conceptual.

A. Containerization & Runtime Footprint

All major subsystems run in **Apptainer/Singularity** containers, which is compatible with SJSU HPC. Containers encapsulate:

- OS + Python runtime
- PyTorch / CUDA stack (where needed)
- gRPC servers
- UCX/InfiniBand support
- Logging and metrics stack
- The agi Python package

Representative image set:

- lh.sif – Left Hemisphere (LLM, planning, metacog, pre-action safety)
- rh.sif – Right Hemisphere (perception, world model, control, in-action safety)
- memory_semantic.sif – vector DB + semantic service
- memory_episodic.sif – episodic store/index service
- memory_procedural.sif – skill catalog/manager
- safety_meta.sif – optional combined safety + metacognition microservice
- dht_router.sif – optional DHT node(s) for polycentric cognition
- tools_cli.sif – orchestration/inspection tools

Typical base images

- LH / RH:
    - nvcr.io/nvidia/pytorch:24.02-py3 (CUDA + cuDNN + PyTorch)
- Memory / DHT / tools:
    - python:3.11-slim or similar, plus UCX, qdrant-client/FAISS, SQL DB libs

Each image installs:

- agi (this repo, as a Python package)
- grpcio / grpcio-tools

- ucx-py / UCX
- prometheus_client and (optionally) OpenTelemetry
- pyyaml, pydantic, etc. for configs

B. SLURM-Based Deployment on SJSU HPC

Services are launched as **SLURM jobs** with time limits and resource constraints matching SJSU policies. Jobs run as **session-based cognitive runs**, with checkpointing to persistent storage.

B.1 Example LH Job

```
#!/bin/bash
#SBATCH --job-name=agi-lh
#SBATCH --partition=gpu
#SBATCH --gres=gpu:H100:1        # or A100:1
#SBATCH --cpus-per-task=16
#SBATCH --mem=200G
#SBATCH --time=24:00:00
#SBATCH --output=/scratch/%u/agi/logs/lh.%j.log

module load apptainer

export AGI_RUN_ID=${AGI_RUN_ID:-lh_${SLURM_JOB_ID}}
export AGI_CONFIG=/data/agi/configs/lh_config.yaml

srun apptainer exec /data/agi_images/lh.sif \
   python -m agi.lh.service --config $AGI_CONFIG
```

B.2 Example RH Job

```
#!/bin/bash
#SBATCH --job-name=agi-rh
#SBATCH --partition=gpu
#SBATCH --gres=gpu:A100:1
#SBATCH --cpus-per-task=24
#SBATCH --mem=220G
#SBATCH --time=24:00:00
```

```
#SBATCH --output=/scratch/%u/agi/logs/rh.%j.log

module load apptainer

export AGI_RUN_ID=${AGI_RUN_ID:-rh_${SLURM_JOB_ID}}
export AGI_CONFIG=/data/agi/configs/rh_config.yaml

srun apptainer exec /data/agi_images/rh.sif \
  python -m agi.rh.service --config $AGI_CONFIG
```

B.3 Memory Services Jobs

Semantic, episodic, and procedural memory typically run on CPU nodes:

```
#!/bin/bash
#SBATCH --job-name=agi-mem-semantic
#SBATCH --partition=cpu
#SBATCH --cpus-per-task=16
#SBATCH --mem=128G
#SBATCH --time=24:00:00
#SBATCH --output=/scratch/%u/agi/logs/memory_semantic.%j.log

module load apptainer

srun apptainer exec /data/agi_images/memory_semantic.sif \
  python -m agi.memory.semantic_service \
      --config /data/agi/configs/memory_semantic.yaml
```

A "stack launcher" script can submit memory → LH → RH in sequence, passing node names/ports via env vars or config.

C. Left Hemisphere Runtime (Planning, Meta, Pre-Action Safety)

The LH service is implemented as a **microservice** with multiple cooperating loops (threads or async tasks).

Key internal modules:

- planning/ – task/plan graphs, hierarchical planning, search

- metacognition/ – reasoning evaluation, confidence, cross-checks
- safety_preaction/ – tool registry, plan/code validators, hallucination detection
- memory_client/ – gRPC stubs for semantic / episodic / procedural access
- events/ – event fabric client (UCX-based pub/sub)
- api/ – gRPC server exposing PlanService, optional MetaService

Representative internal loops:

- RPCServerThread
    - Handles inbound RPCs (e.g., “propose plan for goal X”).
- EventListenerThread
    - Subscribes to perception.state_update, trajectory.risk_high, goal.update.
- PlanningThread
    - Maintains goal/plan graphs and issues SimulationRequests to RH.
- SafetyGuardThread
    - Runs pre-action safety rules on draft plan segments.
- MetaThread
    - Runs metacognitive passes under a time/compute budget.
- MemoryClientThread
    - Issues async queries and writes to memory (e.g., storing reasoning summaries).

Plan execution pipeline:

1. Receive a goal or state-change event.
2. Query semantic memory for relevant facts/docs/tools.
3. Use LLM (+ skills) to sketch plan candidates.
4. Run pre-action safety:
    - verify tool & object references,
    - check constraints & hallucinations,
    - filter obviously unsafe plans.
5. Request simulations from RH (SimulationRequest).
6. Call metacognition for critical steps (confidence, inconsistencies).
7. Approve a plan segment; publish plan.step_ready or control.request to RH.

D. Right Hemisphere Runtime (Perception, World Model, Control, In-Action Safety)

The RH service handles the **sensorimotor side**: perception → state → simulation → control, plus in-action safety.

Key internal modules:

- perception/
    - Camera client (laptop env → cluster)
    - Preprocessing (CUDA)
    - Vision encoders (CLIP/DINO-like)
    - Object detection / segmentation
    - Scene graph builder
- world_model/
    - Env API client for rollouts
    - (Later) learned world-models for "mental" simulation
- control/
    - Trajectory planner
    - Inverse kinematics, joint-space or Cartesian controllers
- safety_in_action/
    - Runtime constraint enforcement, risk scoring, emergency stop
- api/
    - gRPC server for SimulationService, ControlService
- events/
    - Publishes perception.state_update, safety.flag, emergency_stop

Perception pipeline:

1. Receive StepResult from environment (RGB, depth, object metadata, proprioception).
2. Preprocess frames on GPU.
3. Encode via vision model → embeddings + detections.
4. Update state & scene graph.
5. Publish perception.state_update (with references to episodic logs).

Simulation & control pipeline:

- **Simulation:**
    1. Receive SimulationRequest with a candidate plan segment.
    2. Run N-step rollouts:
        - either via cloned Unity/MuJoCo state,

        - or via a learned dynamics model.
    3. Compute risk & constraint violations.
    4. Return SimulationResult (trajectories, risk, suggested modifications).
- **Control:**
    1. Receive ControlRequest with an approved plan segment or skill call.
    2. Generate low-level trajectories.
    3. Apply in-action safety checks (limits, predicted collisions).
    4. Stream commands to environment (env_client).
    5. If risk spikes or env state diverges → raise safety.flag / emergency_stop.

E. Memory Subsystems: Semantic, Episodic, Procedural

E.1 Semantic Memory

Backend: Qdrant / FAISS with a thin Python API.

Schema:

- id
- embedding: float32[N]
- type: concept / tool_doc / scene_summary / rule / skill_summary …
- tags: arbitrary metadata (domain, risk level, source)
- timestamp, last_accessed
- provenance: originating run, episode, model version

Ops:

- SemanticWrite – insert/upsert entries, with versioning.
- SemanticQuery – vector + filter search, with pagination.
- Drift handling and simple GC are implemented per earlier memory section:
    - keep safety-related and frequently-used entries “pinned”,
    - be more aggressive with seldom-used, low-confidence entries.

E.2 Episodic Memory

On disk:

```
/scratch/agi/episodes/<run_id>/<episode_id>/
  frames/           # WebP / JPEG
  state.parquet     # state over time
```

actions.parquet

sims.parquet # world-model predictions

safety.parquet # safety events

meta.json # summary, tags, seeds, model versions

Index DB (Postgres or SQLite):

- Tables for episodes, events, tags.
- Indexed by:
    - time, task id, safety incident, success/failure.

APIs:

- EpisodicAppend – append step(s) with references to logs.
- EpisodicQuery – filter by task, time range, event labels, or semantic descriptors (via embedded summaries).

These logs feed replay/regression testing and offline learning.

E.3 Procedural Memory

Procedural memory tracks **skills** as structured capabilities:

- name, domain, tags
- preconditions, postconditions
- policy_ref (script or learned policy checkpoint)
- success_rate, recent_incidents, usage_count
- applicable_context (object types, geometry ranges, etc.)

Ops:

- SkillGet / SkillSearch – for LH and RH when planning/executing.
- Skill discovery jobs mine episodic logs to propose new skills.
- Skill validation jobs use the environment to test candidate skills under variation before promotion.

F. Virtual World Simulation

The virtual environment is the **agent's body and world**, running on a dedicated laptop.

F.1 Unity-Based Server

Run Unity in headless mode:

```
Unity -batchmode -nographics \
    -projectPath ./AGIWorld \
```

-executeMethod AgiServer.EntryPoint \

-logFile unity_server.log

API surface (WebSocket or gRPC):

- StepEnv(ActionBatch) -> StepResult
- ResetEnv(ResetConfig) -> StepResult
- GetState(StateQuery) -> StateSnapshot

Where StepResult contains:

- camera images (RGB, optionally depth),
- object metadata (IDs, types, poses, velocities),
- proprioception (joint states, contacts),
- reward/goal signals,
- done flags, scenario IDs.

The Unity project implements:

- Task templates and randomization ranges.
- Safety-stress scenarios (unstable stacks, moving obstacles, narrow passages).
- Hooks for curriculum control.

F.2 MuJoCo Alternative (Optional)

For more classical robotics:

- A Python server using MuJoCo plus mujoco Python bindings.
- Same conceptual API (step/reset/get_state).
- Configurable sim rate (30–240 Hz).
- Deterministic physics, useful for RL-style experiments and batch rollouts.

RH chooses backend based on experiment configuration.

G. Communication Middleware: Event Fabric + Optional DHT

The system uses:

- **Event fabric** for real-time pub/sub.
- **gRPC** for structured RPC calls.
- **DHT** (optional) for shared cognitive state in polycentric configurations.

G.1 Event Fabric

Implementation: UCX-backed messaging via ucx-py, or ZeroMQ configured to use InfiniBand.

Common topics:

- perception.state_update
- plan.step_ready
- simulation.request / simulation.result
- safety.flag / emergency_stop
- memory.updated
- health.heartbeat.<service>

Features:

- Per-publisher FIFO ordering on each topic.
- Trace IDs in headers for correlation.
- Backpressure and prioritization:
    - safety and control topics > planning > logging.

G.2 RPC Layer (gRPC)

Each service exposes gRPC endpoints:

- LH:
    - PlanService, MetaService (if metacog isn't split out)
- RH:
    - SimulationService, ControlService
- Memory:
    - SemanticService, EpisodicService, ProceduralService
- Safety:
    - SafetyService (pre + post)
- Meta (if separate):
    - MetacognitionService

All messages are defined in proto/*.proto and versioned as described in the API & Versioning section.

G.3 DHT (Polycentric Mode)

DHT is introduced once more agents exist:

- Kademlia-style overlay with uvloop and msgpack encoding.
- Serves keys such as:
    - plan/<task_id>/graph
    - scene/current

    - skill/<domain>/<name>
    - safety/policies/vN
- Conflict resolution:
    - version vectors and type-specific merge rules.
- Safety-critical keys are replicated more aggressively and may have stricter update policies.

H. Safety System Implementation (Cross-Cutting)

Safety is not a single module—it's woven through LH, RH, and memory, but we can still describe its **implementation pieces**.

H.1 Pre-Action (LH / Safety Service)

- **Tool registry** with strict schemas:
    - allowed tools, parameters, argument types.
- **Plan validator**:
    - ensure all objects and tools referenced exist in current scene or memory.
    - verify preconditions that can be statically checked.
- **Code safety** for generated code:
    - static checks on imports, file/OS/network operations.
    - sandboxed execution only via approved harnesses.

Outputs: SafetyCheckPlan responses with:

- classification (SAFE / RISKY / BLOCKED),
- flagged steps and rationales,
- possible relaxations or extra constraints.

H.2 In-Action (RH)

- **Hard constraints**:
    - joint positions, velocities, torques, workspace bounds.
    - max contact forces, avoid entering restricted regions.
- **Receding-horizon checks**:
    - inspect future states over N steps (e.g., 5–20) for collisions or constraint violations.
- **Emergency stop**:
    - triggered by:

        - predicted imminent violation,
        - out-of-distribution sensor readings,
        - environment disconnection.
    - Effect:
        - stop commands, hold safe pose,
        - log event to episodic memory.

H.3 Post-Action (Jobs)

Offline safety analysis jobs:

- scan episodes for:
    - incidents, near misses, false positives/negatives.
- generate:
    - new or refined safety rules,
    - recommended threshold changes,
    - new adversarial scenarios for future testing.

These jobs are standard SLURM batch jobs running over episodic data on CPU nodes or GPU nodes (if needed for model-based analyses).

I. Monitoring, Observability & Dev Workflow

I.1 Metrics & Dashboards

Each service exposes a Prometheus endpoint (e.g., :9100/metrics):

- LH:
    - planning latency, plans/sec
    - safety checks/sec, block rates
    - metacog invocations, confidence distributions
- RH:
    - perception latency, sim latency
    - control loop frequency, emergency stops
- Memory:
    - query latencies, qps, cache hit rates
    - GC events and memory utilization
- Safety/Meta:
    - checks/sec, incidents avoided, false positives

Grafana dashboards pull from Prometheus and show:

- task success/failure rates
- safety incident trends
- calibration metrics (confidence vs. actual success)
- HPC resource utilization per run

I.2 Logging & Tracing

- JSON logs with:
    - timestamp, service, level, trace_id, run_id, event_type, and relevant payload summary.
- trace_id assigned per episode/experiment and propagated across LH/RH/memory/safety/metacog calls.
- Logs stored under /scratch/<user>/agi/logs/ for raw detail; summarized into /data/agi/logs/ for longer-term analysis.

Simple log search (or an ELK-like stack if you want) allows reconstructing what happened for any episode.

I.3 Local Dev & Mini-Stack

For development:

- Run semantic memory + LH + RH (with a mock environment) on a single machine:
    - python -m agi.memory.semantic_service --config configs/dev_memory.yaml
    - python -m agi.lh.service --config configs/dev_lh.yaml
    - python -m agi.rh.service --config configs/dev_rh_mock_env.yaml
- Use a local docker-compose (if not on HPC) to spin up:
    - Qdrant or Postgres,
    - a mock env server,
    - and Prometheus/Grafana for quick feedback.

Unit and integration tests are run via pytest with CI, while selected reduced scenarios can be run as GitHub Actions or other CI pipelines without needing the full HPC.

XX. Appendices & Technical Specifications

This appendix provides deep engineering detail: concrete API schemas, data structures, SLURM templates, embodiment integration formats, failure modes, operational guidelines, and security considerations. The appendices are written in an implementation-oriented style for engineers deploying on HPC or enterprise clusters.

What follows is **Appendix A — API Schema Definitions**, focused on the core internal gRPC/Protobuf contracts that bind LH, RH, memory, safety, metacognition, and the virtual environment.

Appendix A — API Schema Definitions

This appendix defines the **canonical Protobuf-style schemas** for the primary services:

- Left Hemisphere (planning, pre-action safety, metacognition)
- Right Hemisphere (perception, world modeling, control, in-action safety)
- Memory (semantic, episodic, procedural)
- Safety & Metacognition services (if split out)
- Virtual Environment
- Shared/common types

You don't have to implement Protobuf specifically, but the schemas should map cleanly to gRPC, FlatBuffers, Cap'n Proto, or similar.

A.1 Schema Conventions

- **Naming**
    - Package prefix: agi.<domain>.<version>, e.g. agi.lh.v1.
    - Service names: <Component>Service (PlanService, SemanticService, etc.).
    - Messages: PascalCase, fields snake_case.
- **Versioning**
    - Each package is versioned: v1, v2, …
    - Backwards-compatible changes (adding optional fields, new RPCs) → MINOR bump in doc, but still v1 in proto package.
    - Breaking changes (field removals/semantic changes) → v2 package.
- **Common fields**

- Most messages inherit a common header:
  - string trace_id – correlates all calls within a task/episode.
  - string run_id – HL experiment/session identifier.
  - string schema_version – semantic version (e.g. "1.3.0").

- **Error handling**
  - Use gRPC status codes for transport-level errors.
  - Application-level errors go into:
    - repeated ErrorDetail errors with codes/explanations.

A.2 Common Types (Shared Package)

```
syntax = "proto3";

package agi.common.v1;

message Header {
  string trace_id      = 1;
  string run_id        = 2;
  string schema_version = 3;
  int64  timestamp_ms   = 4;
}

enum RiskLevel {
  RISK_UNKNOWN = 0;
  RISK_LOW     = 1;
  RISK_MEDIUM  = 2;
  RISK_HIGH    = 3;
  RISK_CRITICAL= 4;
}
```

```
message ErrorDetail {
  string code       = 1;  // e.g. "INVALID_PLAN", "TIMEOUT", "SAFETY_BLOCK"
  string message    = 2;
  string component  = 3;  // "lh", "rh", "safety", "memory"
  string suggestion = 4;  // optional remediation hint
}

message Pose3D {
  // Basic SE(3) pose in world coordinates
  repeated double position  = 1; // [x, y, z]
  repeated double orientation = 2; // [qx, qy, qz, qw]
}

message ObjectRef {
  string object_id   = 1;
  string object_type = 2; // "block", "table", "tool", etc.
}

message TimeWindow {
  int64 start_ms = 1;
  int64 end_ms   = 2;
}
```

These common types are imported by most component packages.

A.3 Left Hemisphere APIs (agi.lh.v1)

The LH exposes planning and (optionally) metacognition endpoints.

```
syntax = "proto3";

package agi.lh.v1;
```

```
import "agi/common/v1/common.proto";

service PlanService {
  rpc ProposePlan(PlanRequest) returns (PlanResponse);
  rpc UpdateGoal(GoalUpdateRequest) returns (GoalUpdateResponse);
}

service MetaService {
  rpc ReviewPlan(MetaReviewRequest) returns (MetaReviewResponse);
}
```

A.3.1 PlanService Messages

```
message GoalSpec {
  string goal_id   = 1;
  string natural_language_goal = 2; // "stack three blocks", etc.
  string task_type = 3;             // "manipulation", "navigation", ...
  map<string, string> parameters = 4; // e.g., {"num_blocks": "3"}
}

message PlanStep {
  string step_id   = 1;
  string action    = 2;  // high-level verb, e.g. "pick", "place"
  map<string, string> arguments = 3; // object IDs, positions, etc.
  agi.common.v1.RiskLevel estimated_risk = 4;
  double estimated_cost   = 5; // arbitrary cost (time/effort)
}

message PlanGraph {
  string plan_id = 1;
```

```
  repeated PlanStep steps = 2;
  // adjacency list: index into steps -> indices of successors
  repeated int32 edges_from = 3;
  repeated int32 edges_to   = 4;
}

message PlanRequest {
  agi.common.v1.Header header = 1;
  GoalSpec goal               = 2;
  // Optional current plan context (for replanning)
  PlanGraph existing_plan     = 3;
}

message PlanResponse {
  agi.common.v1.Header header    = 1;
  PlanGraph plan                 = 2;
  agi.common.v1.RiskLevel risk   = 3;
  repeated agi.common.v1.ErrorDetail errors = 4;
}
```

A.3.2 Metacognition Messages

```
message ReasoningTrace {
  // Free-form but structured trace of LH reasoning (token-limited)
  repeated string steps = 1;
}

message MetaReviewRequest {
  agi.common.v1.Header header     = 1;
  GoalSpec goal                   = 2;
  PlanGraph plan                  = 3;
```

```
  ReasoningTrace reasoning_trace  = 4;
  // Summarized simulation results, safety flags, memory hits
  string context_summary         = 5;
}

message MetaReviewResponse {
  agi.common.v1.Header header     = 1;
  double confidence               = 2; // [0,1]
  repeated string issues          = 3; // explanations
  enum Decision {
    DECISION_UNKNOWN = 0;
    ACCEPT          = 1;
    REVISE          = 2;
    REJECT          = 3;
  }
  Decision decision               = 4;
  repeated agi.common.v1.ErrorDetail errors = 5;
}
```

A.4 Right Hemisphere APIs (agi.rh.v1)

The RH exposes simulation and control endpoints.

```
syntax = "proto3";

package agi.rh.v1;

import "agi/common/v1/common.proto";

service SimulationService {
  rpc SimulatePlan(SimulationRequest) returns (SimulationResult);
```

```
}

service ControlService {
  rpc ExecuteSegment(ControlRequest) returns (ControlStatus);
}
```

A.4.1 Simulation

```
message SimulationStep {
  int32 step_index                = 1;
  repeated agi.common.v1.ObjectRef objects = 2;
  map<string, double> metrics       = 3; // e.g. {"min_distance": 0.05}
  agi.common.v1.RiskLevel risk      = 4;
}

message SimulationRequest {
  agi.common.v1.Header header = 1;
  string plan_id             = 2;
  repeated agi.lh.v1.PlanStep steps = 3;  // the segment to simulate
  int32 horizon_steps        = 4;       // e.g., 5–20
}

message SimulationResult {
  agi.common.v1.Header header = 1;
  string plan_id             = 2;
  repeated SimulationStep trajectory = 3;
  agi.common.v1.RiskLevel overall_risk = 4;
  repeated agi.common.v1.ErrorDetail errors = 5;
}
```

A.4.2 Control

```
message ControlCommand {
```

```
  int32 step_index      = 1;   // corresponding to PlanStep index
  string controller_type = 2;   // "joint_space", "cartesian"
  bytes payload         = 3;   // controller-specific serialized data
}

message ControlRequest {
  agi.common.v1.Header header = 1;
  string plan_id            = 2;
  repeated ControlCommand commands = 3;
  double safety_margin       = 4; // e.g. 0.1 → extra clearance
}

message ControlStatus {
  agi.common.v1.Header header = 1;
  string plan_id            = 2;
  enum State {
    STATE_UNKNOWN = 0;
    RUNNING      = 1;
    COMPLETED    = 2;
    ABORTED      = 3;
  }
  State state               = 3;
  repeated agi.common.v1.ErrorDetail errors = 4;
  bool emergency_stop_triggered = 5;
}
```

A.5 Memory APIs (agi.memory.v1)

A.5.1 Semantic Memory

```
syntax = "proto3";
```

```
package agi.memory.v1;

import "agi/common/v1/common.proto";

message SemanticEntry {
  string id                  = 1;
  bytes embedding            = 2; // float32 array
  string entry_type          = 3; // "concept", "tool_doc", ...
  map<string, string> metadata  = 4;
  int64 created_at_ms        = 5;
  int64 updated_at_ms        = 6;
  string provenance          = 7; // run, episode, model
}

message SemanticWriteRequest {
  agi.common.v1.Header header = 1;
  repeated SemanticEntry entries = 2;
}

message SemanticWriteResponse {
  agi.common.v1.Header header = 1;
  repeated string ids        = 2;
  repeated agi.common.v1.ErrorDetail errors = 3;
}

message SemanticQuery {
  agi.common.v1.Header header      = 1;
  bytes query_embedding            = 2;
```

```
  repeated string required_types    = 3;
  map<string, string> filter_tags   = 4;
  int32 top_k                       = 5;
}

message SemanticHit {
  SemanticEntry entry   = 1;
  double score          = 2;
}

message SemanticQueryResponse {
  agi.common.v1.Header header = 1;
  repeated SemanticHit hits   = 2;
  repeated agi.common.v1.ErrorDetail errors = 3;
}

service SemanticService {
  rpc Write(SemanticWriteRequest) returns (SemanticWriteResponse);
  rpc Query(SemanticQuery) returns (SemanticQueryResponse);
}
```

A.5.2 Episodic Memory

```
message EpisodicEvent {
  string episode_id             = 1;
  int64  step_index             = 2;
  string event_type             = 3; // "observation", "action", "safety_flag", ...
  string payload_uri            = 4; // path to Parquet/JSON/frame bundle on storage
  map<string, string> tags      = 5;
  int64 timestamp_ms            = 6;
}
```

```
message EpisodicAppendRequest {
  agi.common.v1.Header header    = 1;
  repeated EpisodicEvent events  = 2;
}

message EpisodicAppendResponse {
  agi.common.v1.Header header    = 1;
  repeated agi.common.v1.ErrorDetail errors = 2;
}

message EpisodicQuery {
  agi.common.v1.Header header    = 1;
  repeated string episode_ids    = 2;
  repeated string event_types    = 3;
  agi.common.v1.TimeWindow time_window = 4;
  map<string, string> tag_filter = 5;
  int32 limit                    = 6;
}

message EpisodicQueryResponse {
  agi.common.v1.Header header    = 1;
  repeated EpisodicEvent events  = 2;
  repeated agi.common.v1.ErrorDetail errors = 3;
}

service EpisodicService {
  rpc Append(EpisodicAppendRequest) returns (EpisodicAppendResponse);
  rpc Query(EpisodicQuery) returns (EpisodicQueryResponse);
```

```
}
```

A.5.3 Procedural Memory

```
message Skill {
  string skill_id              = 1;
  string name                  = 2;
  repeated string domain_tags    = 3;
  string precondition_expr       = 4; // DSL or reference to evaluator
  string postcondition_expr      = 5;
  string policy_ref              = 6; // path to policy or script
  double success_rate            = 7;
  int64  last_used_ms            = 8;
  int64  last_updated_ms         = 9;
  map<string, string> metadata   = 10;
}

message SkillGetRequest {
  agi.common.v1.Header header = 1;
  string skill_id             = 2;
}

message SkillGetResponse {
  agi.common.v1.Header header = 1;
  Skill skill                 = 2;
  repeated agi.common.v1.ErrorDetail errors = 3;
}

message SkillSearchRequest {
  agi.common.v1.Header header = 1;
  repeated string domain_tags = 2;
```

```
  map<string, string> metadata_filter = 3;
  int32 limit = 4;
}

message SkillSearchResponse {
  agi.common.v1.Header header = 1;
  repeated Skill skills       = 2;
  repeated agi.common.v1.ErrorDetail errors = 3;
}

service ProceduralService {
  rpc Get(SkillGetRequest) returns (SkillGetResponse);
  rpc Search(SkillSearchRequest) returns (SkillSearchResponse);
}
```

A.6 Safety & Metacognition APIs (agi.safety.v1, agi.meta.v1)

If safety and metacognition are split into dedicated services, they use the following schemas.

A.6.1 Safety

```
syntax = "proto3";

package agi.safety.v1;

import "agi/common/v1/common.proto";
import "agi/lh/v1/lh.proto";

message SafetyCheckPlanRequest {
  agi.common.v1.Header header = 1;
  agi.lh.v1.GoalSpec goal     = 2;
  agi.lh.v1.PlanGraph plan    = 3;
```

```
}

message SafetyIssue {
  string step_id             = 1;
  string issue_type          = 2; // "TOOL_HALLUCINATION", "OUT_OF_RANGE", ...
  agi.common.v1.RiskLevel risk = 3;
  string description         = 4;
}

message SafetyCheckPlanResponse {
  agi.common.v1.Header header = 1;
  agi.common.v1.RiskLevel overall_risk = 2;
  repeated SafetyIssue issues = 3;
  repeated agi.common.v1.ErrorDetail errors = 4;
}

message SafetyCheckActionRequest {
  agi.common.v1.Header header = 1;
  agi.rh.v1.ControlCommand command = 2;
}

message SafetyCheckActionResponse {
  agi.common.v1.Header header = 1;
  bool allowed               = 2;
  agi.common.v1.RiskLevel risk= 3;
  string reason              = 4;
  repeated agi.common.v1.ErrorDetail errors = 5;
}
```

```
service SafetyService {
  rpc CheckPlan(SafetyCheckPlanRequest) returns (SafetyCheckPlanResponse);
  rpc CheckAction(SafetyCheckActionRequest) returns (SafetyCheckActionResponse);
}
```

A.6.2 Metacognition

If not embedded in LH:

```
syntax = "proto3";

package agi.meta.v1;

import "agi/common/v1/common.proto";
import "agi/lh/v1/lh.proto";

service MetacognitionService {
  rpc Review(MetaReviewRequest) returns (MetaReviewResponse);
}

// MetaReviewRequest/Response defined in agi.lh.v1 can be reused or
// extended here if you prefer separation.
```

A.7 Environment API (agi.env.v1)

The environment server (Unity/MuJoCo) exposes a simple step/reset interface.

```
syntax = "proto3";

package agi.env.v1;

import "agi/common/v1/common.proto";

message Action {
```

```
  string action_type         = 1; // "joint_targets", "end_effector_pose", ...
  bytes payload              = 2; // model-specific data
}

message StepEnvRequest {
  agi.common.v1.Header header = 1;
  repeated Action actions     = 2;
}

message Observation {
  // references to binary blobs (images, depth, etc.) on the env machine,
  // or inline compressed bytes for low-res setups
  string rgb_uri             = 1;
  string depth_uri           = 2;
  string metadata_json       = 3; // object list, etc.
  string proprio_json        = 4; // joint states, contacts
}

message StepEnvResponse {
  agi.common.v1.Header header = 1;
  Observation obs            = 2;
  double reward              = 3;
  bool done                  = 4;
  string scenario_id         = 5;
  repeated agi.common.v1.ErrorDetail errors = 6;
}

message ResetEnvRequest {
  agi.common.v1.Header header = 1;
```

```
  string scenario_id        = 2;
  map<string, string> params  = 3; // randomization ranges, etc.
}

message ResetEnvResponse {
  agi.common.v1.Header header = 1;
  Observation obs           = 2;
  string scenario_id        = 3;
  repeated agi.common.v1.ErrorDetail errors = 4;
}

service EnvService {
  rpc Step(StepEnvRequest) returns (StepEnvResponse);
  rpc Reset(ResetEnvRequest) returns (ResetEnvResponse);
}
```

This Appendix A gives you **concrete, versionable API contracts** that line up with the updated architecture:

- Dual-hemisphere separation (LH/RH APIs).
- Memory split into semantic/episodic/procedural.
- Safety and metacognition as explicit services.
- Environment as a clean, independent server.
- Common headers and risk/error semantics that the Evaluation & Testing Framework can lean on.

Appendix B — Data Formats

This appendix standardizes all **on-disk** and **serialized** formats so components interoperate without guessing. It covers:

- Directory layout

- Embeddings and dense tensors
- Episodic logs
- Semantic & procedural stores
- Safety rule bundles
- Configuration files
- Logging & metric export

Where possible, definitions are “copy-pasteable” into code.

B.1 Filesystem Layout

All paths assume the SJSU HPC cluster; they can be remapped, but the **relative structure** should be preserved.

Area	Path prefix	Purpose
Container images	/data/agi_images/	Apptainer .sif images
Persistent state	/data/agi/	models, safety rules, long-lived state
Episodic logs	/scratch/$USER/agi/episodes/	high-volume run-specific data
Logs (raw)	/scratch/$USER/agi/logs/	per-job logs
Eval results	/data/agi/eval/	JSON/Parquet metrics by suite/run
Temporary / work files	/scratch/$USER/agi/tmp/	ephemeral scratch

Each **episode** is a directory:

```
/scratch/$USER/agi/episodes/<run_id>/<episode_id>/
  frames/
    frame_000001.webp
    frame_000002.webp
    ...
  state.parquet
  actions.parquet
  sims.parquet
  safety.parquet
```

meta.json

B.2 Dense Vector & Tensor Serialization

In RPC / events (Protobuf):

- Use bytes fields with **raw float32 data**.
- Layout: C-contiguous, little-endian, shape known from context.

```
// Example from Appendix A
bytes embedding = 2; // float32[D], C-order
```

Decoding example (Python):

```
import numpy as np

def bytes_to_vec(b: bytes, dim: int) -> np.ndarray:
    return np.frombuffer(b, dtype=np.float32, count=dim)

def vec_to_bytes(vec: np.ndarray) -> bytes:
    return np.asarray(vec, dtype=np.float32).tobytes(order="C")
```

On disk:

- For large matrices (embeddings, trajectories), prefer:
    - Parquet with BINARY columns for blobs, or
    - .npy files if tightly coupled to Python.

Trajectory example (Parquet schema):

Column	Type	Description
episode_id	string	episode identifier
sim_id	string	simulation identifier
step_index	int32	step index in rollout
tensor_name	string	e.g. "state", "control", "contacts"
tensor_blob	binary	float32 tensor bytes

B.3 Episodic Logs

Each episode directory must contain the following canonical files:

B.3.1 meta.json

Schema:


```
{
 "episode_id": "ep_2025_12_01_001",
 "run_id": "run_lh_12345",
 "task": {
  "goal_id": "goal_stack_3_blocks",
  "task_type": "manipulation",
  "description": "Stack three red blocks on the table"
 },
 "environment": {
  "backend": "unity",
  "scenario_id": "stack_three_blocks_v2",
  "seed": 17342
 },
 "models": {
  "lh_llm": "lh-1.2.0",
  "rh_vision": "rh-vision-0.9.1",
  "world_model": "wm-0.3.0",
  "skills_catalog": "skills-2025-11-30"
 },
 "metrics": {
  "success": true,
  "duration_ms": 42210
 },
 "safety": {
  "incidents": 0,
```

```
  "near_misses": 1
 },
 "timestamps": {
  "start_ms": 1733325332000,
  "end_ms": 1733325374210
 }
}
```


Required fields:

- episode_id, run_id, task.goal_id, environment.backend, environment.scenario_id, models.lh_llm, models.rh_vision, metrics.success, timestamps.start_ms, timestamps.end_ms.

Optional but highly recommended:

- safety.*, models.world_model, models.skills_catalog.

B.3.2 state.parquet

Schema (logical):

Column	Type	Description
episode_id	string	episode id
step_index	int32	index in episode
timestamp_ms	int64	wall-clock millis
scene_json	string	JSON of scene graph
agent_state_json	string	JSON of agent state (joints, velocities)
env_state_json	string	optional additional state (env-specific)
notes	string	optional; debug annotations

scene_json is a normalized scene graph (objects, poses, attributes). Example:


```
{
 "objects": [
  {"id": "block_1", "type": "block", "pose": [0.1, 0.2, 0.0, 0, 0, 0, 1]},
```

{"id": "table", "type": "table", "pose": [0, 0, 0, 0, 0, 0, 1]}
]
}

B.3.3 actions.parquet

Schema:

Column	Type	Description
episode_id	string	episode id
step_index	int32	index in episode
timestamp_ms	int64	when action was issued
action_type	string	e.g. "joint_targets", "end_effector_pose", "gripper_open"
payload_json	string	controller-specific fields
lh_plan_step_id	string	links to PlanStep.step_id
skill_id	string	skill used (if any)

payload_json example:

{
"joint_targets_deg": [10, 5, -3, 0, 0, 0],
"duration_ms": 200
}

B.3.4 sims.parquet

Schema:

Column	Type	Description
episode_id	string	episode id
sim_id	string	simulation id
step_index	int32	associated episode step, or -1 for aggregated

Column	Type	Description
horizon_steps	int32	number of steps in rollout
overall_risk	int32	RiskLevel enum value
summary_json	string	human/machine-readable summary
trajectory_uri	string	optional; pointer to trajectory file (Parquet)

summary_json might include min distances, collisions, etc.

B.3.5 safety.parquet

Schema:

Column	Type	Description
episode_id	string	episode
event_index	int32	index per episode
timestamp_ms	int64	when event occurred
event_type	string	pre_action_block, runtime_flag, emergency_stop, rule_update
rule_id	string	safety rule that fired
risk	int32	RiskLevel enum
details_json	string	explanation, additional structured info

B.4 Semantic Memory Data

Semantic memory has a vector index + metadata.

Metadata table (SQL-style):

```
CREATE TABLE semantic_entries (
  id TEXT PRIMARY KEY,
  entry_type TEXT NOT NULL,
  tags JSONB NOT NULL,
  created_at_ms BIGINT NOT NULL,
  updated_at_ms BIGINT NOT NULL,
```

```
  provenance TEXT NOT NULL,
  embedding_dim INT NOT NULL,
  deleted BOOLEAN NOT NULL DEFAULT FALSE
);
```

Tags conventions:

- domain: "manipulation", "navigation", "tooling", "safety", …
- kind: "concept", "tool_doc", "scene_summary", "rule", …
- safety_level: "high_trust", "low_trust", "experimental"
- source: "lh", "rh", "offline_learning"

Example tags JSON:

```
{
  "domain": "manipulation",
  "kind": "tool_doc",
  "language": "en",
  "safety_level": "high_trust"
}
```

The actual embedding lives in the vector DB (Qdrant/FAISS) keyed by id.

B.5 Procedural Memory (Skill Catalog)

As defined in Appendix A, but on disk we use an SQL table plus optional JSON index.

```
CREATE TABLE skills (
  skill_id TEXT PRIMARY KEY,
  name TEXT NOT NULL,
  domain_tags TEXT[] NOT NULL,
  precondition_expr TEXT NOT NULL,
  postcondition_expr TEXT NOT NULL,
  policy_ref TEXT NOT NULL,
  success_rate DOUBLE PRECISION NOT NULL,
  last_used_ms BIGINT,
```

```
  last_updated_ms BIGINT NOT NULL,
  metadata JSONB NOT NULL
);
```

Canonical metadata keys:

- version: e.g., "block_stacking_v3"
- validated: boolean
- validation_suite: list of benchmark scenario IDs
- max_params: small JSON describing the range where the skill is known to work reliably.

Example:

```
{
  "version": "block_stacking_v3",
  "validated": true,
  "validation_suite": ["stack_three_blocks_v1", "stack_three_blocks_v2"],
  "max_stack_height": 4
}
```

B.6 Safety Rule Bundles

Rules are versioned YAMLs. One active file, multiple historical versions.

File name convention:

- /data/agi/safety/rules_v<MAJOR>.<MINOR>.<PATCH>.yaml

Canonical YAML schema:

```
version: "1.2.0"
created_at_ms: 1733325000000
author: "safety-team"
description: "Baseline rules for manipulation tasks"

defaults:
  pre_action_threshold: "MEDIUM"   # MINIMAL / LOW / MEDIUM / HIGH / MAXIMAL
  in_action_horizon_steps: 10
```

```yaml
rules:
  - id: "no_unregistered_tools"
    applies_to: ["plan"]
    type: "tool_registry_check"
    severity: "CRITICAL"
    config:
      allowed_tool_ids:
        - "move_to_pose"
        - "grasp"
        - "place"
        - "release"

  - id: "max_joint_velocity"
    applies_to: ["action"]
    type: "joint_limit"
    severity: "HIGH"
    config:
      max_velocity_deg_s: 180.0

  - id: "keepout_zone_human"
    applies_to: ["action"]
    type: "workspace_constraint"
    severity: "CRITICAL"
    config:
      forbidden_regions:
        - name: "human_zone"
          shape: "box"
          center: [0.0, 0.0, 0.9]
```

size: [1.0, 1.0, 0.6]

Safety services parse these into internal data structures and maintain an in-memory copy; offline jobs update the file and bump version.

B.7 Configuration Files

Configs are **YAML** and map to Pydantic/data classes.

Example: lh_config.yaml (full version)

```
service:
  name: "lh"
  log_level: "INFO"
  rpc_port: 6000
  metrics_port: 9100

network:
  event_bus:
    backend: "ucx"
    uri: "ucx://lh-node:5000"

llm:
  model_path: "/data/agi/models/lh-llm-1.2.0"
  max_tokens: 4096
  temperature: 0.3
  top_p: 0.95
  max_batch_size: 4

memory:
  semantic:
    host: "mem-semantic-node"
    port: 6100
```

```
  episodic:
    host: "mem-episodic-node"
    port: 6200
  procedural:
    host: "mem-procedural-node"
    port: 6300

safety:
  rules_path: "/data/agi/safety/rules_v1.2.0.yaml"
  pre_action_threshold: "MEDIUM"

metacognition:
  enabled: true
  max_review_ms: 100
  invoke_on_risk_at_least: "MEDIUM"

evaluation:
  enable_detailed_traces: false
```

B.8 Logging & Metrics Export Formats

Logging: JSON lines.

Canonical record fields:

- timestamp (ISO 8601 with ms)
- service ("lh", "rh", "memory_semantic", "env", …)
- level (DEBUG, INFO, WARN, ERROR)
- run_id
- trace_id
- event_type (short machine-readable label)
- message (short human-readable text)

- payload (structured JSON with details)

Example:


```
{
  "timestamp": "2025-12-01T21:22:33.456Z",
  "service": "lh",
  "level": "INFO",
  "run_id": "run_lh_12345",
  "trace_id": "trace_ep_2025_12_01_001",
  "event_type": "plan_generated",
  "message": "Plan generated for goal stack_three_blocks",
  "payload": {
    "plan_id": "plan_abc123",
    "num_steps": 7,
    "estimated_risk": "RISK_MEDIUM"
  }
}
```


Metrics:

- Prometheus exposition format at /metrics.
- Use descriptive metric names and labels, e.g.:
    - agi_lh_plans_total{result="success"|"fail"}
    - agi_rh_control_loop_latency_ms_bucket
    - agi_memory_semantic_query_latency_ms_bucket

Appendix C — Event Fabric & Topic Schemas

This appendix deepens the definition of the **event fabric**, including:

- Transport guarantees
- Standard envelope proto
- Topic naming & payload types
- QoS / priority rules
- Error handling patterns

C.1 Transport & Semantics

Transport:

- Primary: UCX-based messaging (e.g., ucx-py over InfiniBand).
- Fallback: ZeroMQ PUB/SUB configured to use the fastest available transport.

Guarantees:

- Per-publisher FIFO per topic.
- At-least-once delivery (producers may retry; consumers must be idempotent).
- No global total order across topics.

Usage style:

- Low-latency, high-frequency channels: perception, control status.
- Lower-frequency channels: memory updates, learning notifications, heartbeats.

C.2 Event Envelope Protobuf

All events share a standard envelope type.

```
syntax = "proto3";


package agi.events.v1;


import "agi/common/v1/common.proto";


message EventEnvelope {
  string topic               = 1;  // e.g.
"perception.state_update.v1"
  agi.common.v1.Header header = 2; // trace_id, run_id, timestamp
```

```
  string publisher           = 3;  // "lh", "rh",
"memory_semantic", ...
  string payload_type        = 4;  // full type name, e.g.
"agi.rh.v1.PerceptionStateUpdate"
  bytes payload              = 5;  // serialized Protobuf message
}
```

Applications:

- Publisher creates the payload message (e.g., PerceptionStateUpdate), serializes it, and wraps in EventEnvelope.
- Consumers inspect topic and payload_type to deserialize and handle appropriately.

C.3 Topic Naming & Versioning

Naming convention:

<namespace>.<event_name>.v<MAJOR>

Where <namespace> is a short functional area (perception, plan, safety, etc.).

Core topics (baseline single-agent system):

1. **Perception & State**
    - `perception.state_update.v1`
        - Publisher: RH
        - Payload: agi.rh.v1.PerceptionStateUpdate (see below)
        - Frequency: 10–30 Hz
2. **Planning & Control**
    - `plan.step_ready.v1`
        - Publisher: LH
        - Payload: agi.lh.v1.PlanStep wrapped in a small envelope
    - `control.status.v1`
        - Publisher: RH
        - Payload: agi.rh.v1.ControlStatus
3. **Simulation**
    - `simulation.request.v1`
        - Publisher: LH
        - Payload: agi.rh.v1.SimulationRequest
    - `simulation.result.v1`

        - Publisher: RH
        - Payload: agi.rh.v1.SimulationResult

4. **Safety**
    - `safety.flag.v1`
        - Publisher: LH or RH
        - Payload: SafetyEventFlag (defined below)
    - `safety.emergency_stop.v1`
        - Publisher: RH
        - Payload: SafetyEmergencyStopEvent

5. **Memory & Learning**
    - `memory.updated.v1`
        - Publisher: memory services
        - Payload: MemoryUpdateEvent
    - `learning.skill_promoted.v1`
        - Publisher: offline learning jobs
        - Payload: SkillPromotionEvent

6. **Health & Monitoring**
    - `health.heartbeat.<service_name>.v1`
        - Publisher: each service (LH, RH, memory_*)
        - Payload: HeartbeatEvent

Polycentric extensions (later phases):

- `cog.agent_registered.v1, cog.agent_unregistered.v1`
- `cog.agent_bidding.v1, cog.scheduler_decision.v1`
  Payloads are defined in a future agi.cog.v1 package.

C.4 Example Topic Payload Schemas

Below are the event-only schemas (as opposed to the RPC definitions in Appendix A). Most reuse the same message types.

C.4.1 Perception State Update

```
package agi.rh.v1;


import "agi/common/v1/common.proto";
```

```
message PerceptionStateUpdate {
  agi.common.v1.Header header = 1;
  string episode_id           = 2;
  int32  step_index           = 3;
  string scene_json           = 4; // normalized scene graph
  string agent_state_json     = 5; // joint states etc.
  string frame_uri            = 6; // pointer to image on env machine
}
```

Sent on perception.state_update.v1 inside EventEnvelope.payload.

C.4.2 Plan Step Ready

```
package agi.lh.v1;

import "agi/common/v1/common.proto";

message PlanStepEnvelope {
  agi.common.v1.Header header = 1;
  string plan_id              = 2;
  PlanStep step               = 3; // defined in Appendix A
}
```

Published on plan.step_ready.v1.

C.4.3 Safety Events

```
package agi.safety.v1;

import "agi/common/v1/common.proto";

message SafetyEventFlag {
  agi.common.v1.Header header = 1;
  string source_component     = 2; // "lh" or "rh"
  string episode_id           = 3;
  string plan_id              = 4;
```

```
  string step_id              = 5;
  agi.common.v1.RiskLevel risk = 6;
  string rule_id              = 7;
  string description          = 8;
}

message SafetyEmergencyStopEvent {
  agi.common.v1.Header header = 1;
  string episode_id           = 2;
  string reason               = 3;
  agi.common.v1.RiskLevel risk = 4;
}
```

- SafetyEventFlag → safety.flag.v1
- SafetyEmergencyStopEvent → safety.emergency_stop.v1

C.4.4 Memory & Learning Events

```
package agi.memory.v1;

import "agi/common/v1/common.proto";

message MemoryUpdateEvent {
  agi.common.v1.Header header = 1;
  string service_name         = 2;  // "semantic", "episodic",
"procedural"
  string update_type          = 3;  // "insert", "update", "gc"
  repeated string ids         = 4;  // affected IDs
}
package agi.learning.v1;

import "agi/common/v1/common.proto";
import "agi.memory/v1/memory.proto";

message SkillPromotionEvent {
  agi.common.v1.Header header = 1;
```

```
  agi.memory.v1.Skill skill   = 2;
  string from_version         = 3;
  string to_version           = 4;
}
```

C.4.5 Heartbeat Events

```
package agi.monitoring.v1;

import "agi/common/v1/common.proto";

message HeartbeatEvent {
  agi.common.v1.Header header = 1;
  string service_name         = 2;
  string version              = 3;
  string status               = 4; // "OK", "DEGRADED", "ERROR"
  map<string, string> metrics = 5; // lightweight stats
}
```

Published on health.heartbeat.<service>.v1.

C.5 QoS & Priority Policies

When bandwidth or CPU is constrained, the fabric applies **priority-based handling**:

- **High priority (must not be dropped):**
  - `safety.emergency_stop.v1`
  - `safety.flag.v1`
  - `control.status.v1`
  - `health.heartbeat.*.v1 (for liveness)`
- **Medium priority:**
  - `perception.state_update.v1`
  - `simulation.result.v1`
  - `plan.step_ready.v1`
- **Low priority:**
  - `memory.updated.v1`
  - `learning.skill_promoted.v1`

    - `debug/diagnostic topics`

Implementation strategies:

- High priority:
    - small message sizes,
    - minimal buffering,
    - aggressive retries.
- Low priority:
    - allow dropping or coalescing (e.g., last-known state only).

C.6 Error Handling Patterns

- If deserialization fails:
    - log an ERROR event with payload_type and topic,
    - increment a Prometheus counter agi_events_deserialize_errors_total{topic=...},
    - discard the event (don't crash the process).
- If the subscriber is backlogged:
    - apply backpressure (slow down reader) or drop low-priority topics,
    - never drop safety/emergency topics.
- For **critical flows** (e.g., safety → control), consider dual paths:
    - event fabric for monitoring,
    - RPC (gRPC) for authoritative commands with explicit success/failure responses.

Appendix D — SLURM Job Configurations

This appendix specifies **canonical SLURM templates** and deployment practices for:

- Long-lived cognitive services (LH, RH, memory, etc.)
- Experiment / evaluation jobs
- Offline learning & analysis jobs
- Operational conventions (environment, logs, checkpoints)

Everything assumes SJSU-like HPC constraints (GPU vs CPU partitions, Apptainer, wall-clock limits).

D.1 Shared Conventions

Environment modules:

module load apptainer

plus whatever the HPC docs require (e.g., CUDA modules)

Standard env vars:

- AGI_RUN_ID – logical run/experiment ID.
- AGI_CONFIG – path to primary config file for the job.
- AGI_LOG_DIR – defaults to /scratch/$USER/agi/logs.
- AGI_EPISODES_ROOT – defaults to /scratch/$USER/agi/episodes.

If unset, each service chooses sensible defaults.

D.2 Service Job Templates

D.2.1 LH Service (lh.sbatch)

```
#!/bin/bash
#SBATCH --job-name=agi-lh
#SBATCH --partition=gpu
#SBATCH --gres=gpu:H100:1
#SBATCH --cpus-per-task=16
#SBATCH --mem=200G
#SBATCH --time=24:00:00
#SBATCH --output=/scratch/%u/agi/logs/lh.%j.log

module load apptainer

export AGI_RUN_ID=${AGI_RUN_ID:-lh_${SLURM_JOB_ID}}
export AGI_CONFIG=${AGI_CONFIG:-/data/agi/configs/lh_config.yaml}
export AGI_LOG_DIR=${AGI_LOG_DIR:-/scratch/$USER/agi/logs}

srun apptainer exec /data/agi_images/lh.sif \
      python -m agi.lh.service --config "$AGI_CONFIG"
```

D.2.2 RH Service (rh.sbatch)

```
#!/bin/bash
#SBATCH --job-name=agi-rh
#SBATCH --partition=gpu
#SBATCH --gres=gpu:A100:1
#SBATCH --cpus-per-task=24
#SBATCH --mem=220G
#SBATCH --time=24:00:00
#SBATCH --output=/scratch/%u/agi/logs/rh.%j.log

module load apptainer

export AGI_RUN_ID=${AGI_RUN_ID:-rh_${SLURM_JOB_ID}}
export AGI_CONFIG=${AGI_CONFIG:-/data/agi/configs/rh_config.yaml}
export AGI_LOG_DIR=${AGI_LOG_DIR:-/scratch/$USER/agi/logs}

srun apptainer exec /data/agi_images/rh.sif \
    python -m agi.rh.service --config "$AGI_CONFIG"
```

D.2.3 Memory Services (mem_semantic.sbatch, etc.)

```
#!/bin/bash
#SBATCH --job-name=agi-mem-semantic
#SBATCH --partition=cpu
#SBATCH --cpus-per-task=16
#SBATCH --mem=128G
#SBATCH --time=24:00:00
#SBATCH --output=/scratch/%u/agi/logs/mem_semantic.%j.log

module load apptainer

export AGI_CONFIG=${AGI_CONFIG:-/data/agi/configs/memory_semantic.yaml}

srun apptainer exec /data/agi_images/memory_semantic.sif \
```

```
python -m agi.memory.semantic_service --config "$AGI_CONFIG"
```

Similar templates for episodic and procedural memory, using their respective images and configs.

D.3 Stack Orchestration Script

A canonical script to bring up **memory → LH → RH** for a run:

```
#!/usr/bin/env bash
set -euo pipefail

RUN_NAME=${1:-demo_run}

# 1. Start memory services
MEM_SEM_JOB=$(sbatch --parsable infra/hpc/slurm/mem_semantic.sbatch)
MEM_EPI_JOB=$(sbatch --parsable infra/hpc/slurm/mem_episodic.sbatch)
MEM_PROC_JOB=$(sbatch --parsable infra/hpc/slurm/mem_procedural.sbatch)

echo "Semantic mem job:  $MEM_SEM_JOB"
echo "Episodic mem job:  $MEM_EPI_JOB"
echo "Procedural mem job:$MEM_PROC_JOB"

# 2. Start LH after memory jobs are OK
LH_JOB=$(sbatch --parsable \
  --dependency=afterok:$MEM_SEM_JOB:$MEM_EPI_JOB:$MEM_PROC_JOB \
  --export=ALL,AGI_RUN_ID=$RUN_NAME \
  infra/hpc/slurm/lh.sbatch)
echo "LH job: $LH_JOB"

# 3. Start RH after LH is up (or parallel if you prefer)
RH_JOB=$(sbatch --parsable \
  --dependency=afterok:$LH_JOB \
  --export=ALL,AGI_RUN_ID=$RUN_NAME \
```

```
  infra/hpc/slurm/rh.sbatch)
echo "RH job: $RH_JOB"
```

This uses SLURM --dependency=afterok: to ensure services start in the right order.

D.4 Evaluation & Regression Job Templates

D.4.1 Full-Stack Evaluation Suite

Runs a fixed benchmark suite and stores results under /data/agi/eval/.

```
#!/bin/bash
#SBATCH --job-name=agi-eval-suite
#SBATCH --partition=gpu
#SBATCH --gres=gpu:H100:1
#SBATCH --cpus-per-task=16
#SBATCH --mem=200G
#SBATCH --time=08:00:00
#SBATCH --output=/scratch/%u/agi/logs/eval_suite.%j.log

module load apptainer

export AGI_CONFIG=/data/agi/configs/eval_suite.yaml
export AGI_EPISODES_ROOT=${AGI_EPISODES_ROOT:-
/scratch/$USER/agi/episodes}
export AGI_EVAL_OUT=/data/agi/eval/benchmark_v1

srun apptainer exec /data/agi_images/tools_cli.sif \
    python -m agi.eval.run_suite \
      --config "$AGI_CONFIG" \
      --episodes-root "$AGI_EPISODES_ROOT" \
      --output-dir "$AGI_EVAL_OUT/$(date +%Y%m%d_%H%M%S)"
```

D.4.2 Replay / Regression Job

```
#!/bin/bash
#SBATCH --job-name=agi-replay-regress
```

```
#SBATCH --partition=cpu
#SBATCH --cpus-per-task=24
#SBATCH --mem=256G
#SBATCH --time=12:00:00
#SBATCH --output=/scratch/%u/agi/logs/replay_regress.%j.log

module load apptainer

export AGI_CONFIG=/data/agi/configs/replay_regression.yaml
export AGI_EPISODES_ROOT=/scratch/$USER/agi/episodes
export AGI_EVAL_OUT=/data/agi/eval/replay_regress

srun apptainer exec /data/agi_images/tools_cli.sif \
    python -m agi.eval.replay_regression \
      --config "$AGI_CONFIG" \
      --episodes-root "$AGI_EPISODES_ROOT" \
      --output-dir "$AGI_EVAL_OUT/$(date +%Y%m%d_%H%M%S)"
```

D.5 Offline Learning & Safety Analysis Jobs

D.5.1 Skill Discovery

```
#!/bin/bash
#SBATCH --job-name=agi-skill-discovery
#SBATCH --partition=cpu
#SBATCH --cpus-per-task=32
#SBATCH --mem=256G
#SBATCH --time=12:00:00
#SBATCH --output=/scratch/%u/agi/logs/skill_discovery.%j.log

module load apptainer

export AGI_CONFIG=/data/agi/configs/skill_discovery.yaml
```

```
export AGI_EPISODES_ROOT=/scratch/$USER/agi/episodes

srun apptainer exec /data/agi_images/tools_cli.sif \
    python -m agi.learning.skill_discovery \
      --config "$AGI_CONFIG" \
      --episodes-root "$AGI_EPISODES_ROOT"
```

D.5.2 Safety Regression

```
#!/bin/bash
#SBATCH --job-name=agi-safety-regress
#SBATCH --partition=cpu
#SBATCH --cpus-per-task=16
#SBATCH --mem=128G
#SBATCH --time=08:00:00
#SBATCH --output=/scratch/%u/agi/logs/safety_regression.%j.log

module load apptainer

export AGI_CONFIG=/data/agi/configs/safety_regression.yaml
export AGI_EPISODES_ROOT=/scratch/$USER/agi/episodes
export AGI_RULES_PATH=/data/agi/safety/rules_v1.2.0.yaml

srun apptainer exec /data/agi_images/tools_cli.sif \
    python -m agi.safety.offline_regression \
      --config "$AGI_CONFIG" \
      --episodes-root "$AGI_EPISODES_ROOT" \
      --rules-path "$AGI_RULES_PATH"
```

D.6 Debugging & Interactive Patterns

For debugging on HPC:

- Use **short, interactive GPU/CPU jobs**:

```
salloc --partition=gpu --gres=gpu:A100:1 --cpus-per-task=8 --mem=64G --time=01:00:00

module load apptainer

apptainer shell /data/agi_images/lh.sif

# Inside container: run unit tests or small scenarios

pytest tests/agi/lh/test_planner.py
```

- For environment debugging, run Unity/MuJoCo on the laptop and use a small RH configuration (dev_rh_config.yaml) that sends fewer frames at lower resolution.

D.7 Operational Policies & Checkpointing

- **Checkpoint interval:** every N episodes or every M minutes, whichever comes first.
- **Checkpoint contents:**
    - semantic memory snapshot (export from vector DB + metadata),
    - skills catalog export (SQL dump / JSON),
    - last safety rule version,
    - any model weights updated during the run (if online learning is enabled).
- **On SLURM SIGTERM:**
    - Services should:
        - stop accepting new tasks,
        - flush logs,
        - finalize current episodes (mark as aborted if incomplete),
        - write a final checkpoint labeled with the job ID and timestamp.

Appendix E — Security Considerations

This appendix describes how the system avoids turning “embodied AGI on HPC” into “random code and untrusted inputs running everywhere.” It focuses on:

- the security model for **code execution**,
- **data access** and multi-user isolation on the SJSU HPC,
- **network/API** hardening,
- **model and supply-chain** integrity,
- **monitoring and incident response**.

The goal is not formal verification, but a set of concrete practices that make this design deployable in academic or enterprise environments.

E.1 Threat Model & Assumptions

Trust boundaries:

- **HPC cluster:**
    - Managed by SJSU CoE or similar institution.
    - Users are authenticated but not mutually trusting.
    - Cluster nodes are considered *honest-but-misconfigured* (admins can access data; other users shouldn't).
- **Virtual environment laptop:**
    - Single-user machine (operator) that runs Unity/MuJoCo.
    - Connects to the cluster over campus/VPN network.
- **User inputs / prompts / tasks:**
    - Considered **untrusted**; may contain adversarial content or instructions.
- **Generated code & plans:**
    - Treated as untrusted output—even though produced by "our" models.

Out of scope for the first deployment:

- Strong defenses against a hostile cluster administrator.
- Global, internet-scale adversaries.
- Multi-tenant AGI as a shared hosted service (though many controls generalize).

E.2 Code Execution & Sandbox Boundaries

The architecture relies on code generation (e.g., tools, scripts, env logic) but must strictly control *where* code runs and what it can touch.

Principles:

1. **No raw shell execution** from LLM output.
2. All generated code executes in **restricted sandboxes** with:
    - no network access,

    - minimal filesystem access,
    - no privileged operations.
3. Sandbox interfaces are treated as **tools** with schemas enforced via safety checks.

Implementation patterns:

- Code is passed to a "runner" service (inside a container), which:
    - validates against a whitelist of allowed libraries/APIs,
    - runs with ulimit and seccomp-style restrictions (if available),
    - enforces CPU/memory/time limits per execution.
- Allowed side effects:
    - writing to a specific scratch directory,
    - logging to stdout/stderr,
    - emitting structured results via the runner's API.

Any attempt to access arbitrary files, spawn additional processes, or open network sockets is treated as a **safety violation**, logged, and blocked.

E.3 Data Security & Access Control

Data categories:

- **Training/eval data & episodes:** stored under /scratch/$USER/agi/episodes and /data/agi/eval.
- **Long-lived cognitive state** (semantic/procedural memory, models, safety rules): stored under /data/agi/.
- **Logs & metrics:** under /scratch/$USER/agi/logs plus monitoring systems.

Controls:

- Use **per-user directories** with standard POSIX permissions:
    - other cluster users cannot read or write AGI state by default.
- For shared research projects:
    - create a dedicated **Unix group** and shared paths (/data/agi-group/),
    - restrict write access to a small set of maintainers.

Sensitive data guidance:

- Avoid placing personally identifiable information (PII) in episodic logs or semantic memory.
- Any real-world telemetry used for training must be:
    - pseudonymized or anonymized,
    - governed by IRB or institutional policies where applicable.

Backups:

- For critical configuration and safety rules, maintain:
    - a **git repo** (config-as-code),
    - periodic snapshots (e.g., nightly tarballs of /data/agi/safety and /data/agi/configs).

E.4 Network & API Security

Cluster side:

- All services listen **only** on internal interfaces (e.g., IPoIB) or specific cluster networks.
- Use **mTLS or at least shared secrets** for gRPC channels between nodes:
    - certificates/keys stored under /data/agi/secrets/ with restrictive permissions.
- No service exposes unauthenticated gRPC/HTTP endpoints to the general campus network.

Env/laptop side:

- The environment server (Unity/MuJoCo) binds to:
    - localhost or a VPN interface,
    - or a firewall-restricted port accessible only from designated cluster nodes.
- Defensive checks:
    - Env server validates that incoming requests conform to expected formats and rate limits; rejects obviously malformed or excessive traffic.

API hardening:

- gRPC services enforce:
    - **size limits** on messages (e.g., for reasoning traces, embeddings, frames),
    - **rate limits** for potentially expensive endpoints (e.g., SimulationService).
- Strict **schema validation**:

    - undefined fields are ignored or cause errors, never silently used.

E.5 Supply Chain & Model Security

The system depends on container images, Python packages, and model artifacts.

Controls:

- Container images:
    - built from **pinned base images** (specific digests, not latest),
    - stored in a limited set of registries or directly as .sif images.
- Python dependencies:
    - use **pinned versions** in requirements.txt/pyproject.toml,
    - scan for known vulnerabilities periodically.
- Models:
    - stored in /data/agi/models/ with:
        - checksums (e.g., SHA-256) recorded in a models_manifest.json,
        - model versions referenced in meta.json and eval results.

Change controls:

- Model updates (LLM, vision, world-model, skills) must:
    - be tied to git commits and config versions,
    - pass evaluation/safety suites before promotion,
    - be rolled out via a controlled config change (not ad hoc file replacement).

E.6 Security Logging & Incident Response

Security-relevant events include:

- sandbox violations,
- unauthorized tool usage attempts,
- unexpected network connections,
- changes to safety rule bundles or configs.

Practices:

- Log all such events with:

    - event_type like "security_violation", "safety_rules_updated", "sandbox_denied".
- Export Prometheus counters:
    - agi_security_sandbox_violations_total,
    - agi_security_rules_updates_total.

Basic response flow (first version):

1. Detect anomaly (alerts via metrics or log search).
2. Freeze further promotions of models/rules/configs.
3. Collect relevant logs, configs, and episode data.
4. Run targeted replay + safety/eval suites.
5. Patch rules or roll back to last known-good state.
6. Document the incident and update this appendix/runbooks if needed.

Appendix F — Failure Modes & Recovery Strategies

This appendix complements the evaluation framework and security appendix by describing **how things break** and **how the system recovers**—both automatically and with operator intervention.

It is organized by subsystem:

- LH, RH
- Memory & DHT
- Virtual environment & sensor link
- Safety & metacognition
- Event fabric & RPC
- HPC infrastructure

Each section covers: **symptoms**, **probable causes**, **automatic recovery mechanisms**, and **manual runbook steps**.

F.1 Left Hemisphere (LH)

F.1.1 Crash / Unresponsive LH

Symptoms:

- No new plan.step_ready events.
- health.heartbeat.lh missing.
- SLURM job FAILED, OUT_OF_MEMORY, or TIMEOUT.

Likely causes:

- LLM OOM (too many tokens/batches).
- Bug in planning/safety/metacog logic.
- Node failure / SLURM kill.

Automatic strategies:

- Watchdog process:
    - monitors heartbeat; if missed > threshold, marks LH as down.
- RH:
    - upon LH loss, enters a **safe idle state** and stops executing new segments.
- Optional:
    - auto-submit a replacement LH job using the same AGI_RUN_ID and last checkpoint (via a simple launcher).

Manual runbook:

1. Check SLURM status and LH logs for error details.
2. Confirm RH is not still executing old commands ungoverned.
3. Fix evident configuration issues (e.g., reduce max_tokens, adjust safety rules).
4. Restart LH with last checkpoint; if some episodes were mid-flight, mark them as aborted in meta.json.

F.1.2 Over-Planning / No Actions

Symptoms:

- Plans constantly revised; very few control requests.
- Metacog always returns REVISE/REJECT.
- Success rates low; tasks timeout.

Likely causes:

- Overly conservative metacog thresholds.

- Inconsistent simulation/memory signals causing meta to distrust everything.
- Safety rules blocking too many plan steps before they reach RH.

Automatic strategies:

- Meta self-monitoring:
    - if intervention rate > threshold and success rate < baseline, system can temporarily relax meta invocation frequency (within configured bounds).
- Planner heuristics:
    - limit number of re-plans before forcing a “best effort” execution.

Manual runbook:

1. Inspect metacog logs (which issues are reported? conflicting signals?).
2. Relax invoke_on_risk_at_least or max_review_ms.
3. Run small eval suite with meta disabled vs. baseline to calibrate.
4. Update meta policies and redeploy.

F.2 Right Hemisphere (RH)

F.2.1 Loss of Perception / Env Connection

Symptoms:

- No perception.state_update events.
- Env logs show disconnects.
- RH logs contain env RPC timeouts.

Likely causes:

- Env server crash on laptop.
- Network disruption (VPN, campus network issue).
- RH/env version mismatch.

Automatic strategies:

- RH detects repeated env RPC failures:
    - triggers safety.emergency_stop (safe idle state),
    - switches to a degraded mode (no actions, only accepting reconnection).
- Periodic reconnection attempts with exponential backoff.

Manual runbook:

1. Check env server status/logs on laptop; restart if needed.
2. Verify network connectivity and firewall rules.
3. Use a CLI tool to “rebind” RH to the env endpoint if the address changed.
4. Restart RH job if it entered a permanent error state.

F.2.2 Frequent Emergency Stops

Symptoms:

- Tasks rarely complete; many emergency stops.
- Safety logs dominated by one or two rule IDs.

Likely causes:

- Overly strict safety thresholds (clearance, velocity, horizon).
- World-model risk overestimation.
- Misalignment between env physics and world-model assumptions.

Automatic strategies:

- RH can track **false-positive ratio** over evaluation episodes and suggest threshold tuning candidates (logged as meta events).
- Optional: adjust horizon or margin slightly within safe band, then flag run as “experimental.”

Manual runbook:

1. Examine safety.parquet and rule IDs triggering most stops.
2. Replay episodes with visualizations to confirm false positives.
3. Adjust safety rule config (within known safe bounds).
4. Re-run safety/eval suites to ensure no unsafe behavior emerges.

F.3 Memory & DHT

F.3.1 Slow or Failing Memory Queries

Symptoms:

- LH/RH logs show semantic/episodic query timeouts.

- Increased latency in metrics.
- CPU-bound memory nodes.

Likely causes:

- Vector DB misconfiguration or overloaded node.
- Unbounded growth in memory leading to heavy GC/compaction.
- Network congestion.

Automatic strategies:

- Client-side timeouts with fallback:
    - degrade gracefully when semantic memory is slow (plan with partial info).
- Query rate limiting and caching on LH/RH side.

Manual runbook:

1. Inspect memory metrics: query latency, CPU usage, GC frequency.
2. If DB is overloaded:
    - adjust index parameters (e.g., HNSW opts),
    - reduce top-k or add more aggressive GC.
3. Consider moving memory services to stronger CPU nodes.

F.3.2 DHT Inconsistency (Polycentric Mode)

Symptoms:

- Conflicting plan/scene states between agents.
- Frequent merge conflicts logged.
- Metacog/safety complaining about inconsistent inputs.

Likely causes:

- Multiple agents writing to the same keys without coordination.
- Inadequate merge rules or missing version vectors.

Automatic strategies:

- Mark certain namespaces as **single-writer** (one authoritative agent).
- Use versioned, append-only keys instead of in-place updates for critical data.

Manual runbook:

1. Identify keys with frequent conflicts (logs).
2. Assign ownership for those keys or adjust agents' write policies.
3. Update DHT conflict resolution logic for specific types (plans, scenes, safety policies).

F.4 Virtual Environment & Sensor Link

F.4.1 High Latency / Dropped Frames

Symptoms:

- Perception latency spikes.
- Control decisions act on stale state.
- Env RPC timeouts.

Likely causes:

- Network congestion / slow Wi-Fi.
- Overly high frame rate or resolution.
- Laptop underprovisioned / overloaded.

Automatic strategies:

- RH dynamically adjusts:
    - requested frame rate,
    - image resolution or compression level.
- Drop non-critical extra views when under load.

Manual runbook:

1. Reduce env output resolution and FPS in config.
2. Ensure laptop is on wired network if possible.
3. Schedule heavy environment runs when network is less busy.

F.5 Safety & Metacognition

F.5.1 Missed Safety Events (Under-Triggering)

Symptoms:

- Replays show unsafe behavior that wasn't flagged.
- Safety metrics show low incident detection while failures occur.

Likely causes:

- Misconfigured thresholds or horizon steps.
- Incorrect or outdated safety rule bundle.
- Bugs in rule evaluation logic.

Automatic strategies:

- Post-run safety audits:
    - offline jobs scan episodes and mark likely missed incidents,
    - automatically propose stricter rules (but don't promote them automatically).

Manual runbook:

1. Run safety regression suite on latest rules/code.
2. Confirm correct rule version is in use (check config and logs).
3. Update rules and re-run targeted stress tests before promoting changes.

F.5.2 Metacog Pathologies (Always Accept / Always Reject)

Symptoms:

- Meta decisions are "ACCEPT" nearly always (no added value) or "REJECT/REVISE" nearly always (progress stalls).
- Calibration metrics poor.

Likely causes:

- Bad training data or heuristics.
- Mis-configured thresholds.
- Context inputs missing (e.g., no simulation results).

Automatic strategies:

- Meta self-monitoring:
    - If interventions are extremely rare or extremely frequent, log a warning and suggest config changes.

Manual runbook:

1. Analyze a batch of meta decisions vs. outcomes.
2. Adjust thresholds or training; consider limiting meta to high-risk contexts.

3. Re-evaluate using calibration suites.

F.6 Event Fabric & RPC

F.6.1 Dropped / Out-of-Order Events

Symptoms:

- Inconsistent state between LH/RH.
- Gaps in perception or plan event sequences.

Likely causes:

- Fabric under-provisioned or misconfigured.
- Consumer not keeping up; OS buffers filling.

Automatic strategies:

- Backpressure: slow down publishers or drop **low-priority** topics first.
- For critical flows, switch to RPC-based confirmation.

Manual runbook:

1. Check fabric metrics: queue depths, error counts.
2. Confirm priority/QoS config matches Appendix C.
3. Tune buffer sizes or add more fabric workers.

F.7 HPC Infrastructure (SLURM, Nodes, Storage)

Failure modes and strategies here are mostly operational (see Appendix G), but key patterns:

- **Time limit exceeded:**
    - ensure checkpointing before wall clock; reduce workload per job.
- **Node failure / preemption:**
    - design jobs to be idempotent and restartable from checkpoints.
- **Storage full/slow:**
    - prune old episodes/logs; throttle logging level; coordinate with admins.

Appendix G — Operational Guidelines

This appendix defines **how to operate the system day-to-day**:

- environment separation,
- release and change management,
- SLOs and monitoring,
- maintenance routines,
- operational roles.

G.1 Environments: Dev, Staging, Production

Dev:

- Single-node or laptop-based, possibly using Docker instead of Apptainer.
- Mock env or lightweight Unity/MuJoCo scenes.
- Frequent code changes, minimal safety constraints (but still sandboxed code execution).

Staging:

- On SJSU HPC, but:
  - smaller configs (smaller models, fewer episodes),
  - separate paths (/data/agi-staging/),
  - more aggressive logging and metrics.
- Used to validate new models/rules before promotion.

Production / "Research mainline":

- The primary line for long-running experiments and published results.
- Only versions that:
  - pass evaluation & safety suites,
  - have documented configs and changelogs,
  - are tagged in git.

G.2 Release & Change Management

Versioning:

- Code: git tags (vX.Y.Z).

- Models: semantic versions in models_manifest.json.
- Safety rules: rules_vX.Y.Z.yaml.
- Configs: labeled with matching versions.

Change process:

1. Develop change in a feature branch.
2. Add / update tests (unit, integration, evaluation as needed).
3. Run CI (unit + small scenario runs).
4. Deploy to staging; run evaluation + safety suites.
5. If metrics acceptable and no regressions → tag & promote configs/models.
6. Update documentation and appendices if relevant (especially safety and ops).

G.3 Monitoring & SLOs

Key SLOs (first draft):

- **Task success rate** on "baseline benchmark" suite:
    - ≥ target percentage (e.g., 80% for foundational tasks).
- **Safety incident rate**:
    - zero unmitigated incidents in eval suites; near-miss rate tracked separately.
- **System availability** for full-stack runs during scheduled windows:
    - e.g., ≥ 95% of scheduled time.

Monitoring stack:

- Prometheus + Grafana:
    - dashboards for LH, RH, memory, env, safety.
- Logs:
    - aggregated by run/episode; searchable by event_type, trace_id.

Alerts for:

- missing heartbeats,
- sustained high safety violation rates,
- out-of-range metrics (latency, OOM, disk usage near capacity).

G.4 Maintenance & Housekeeping

Regular tasks:

- **Weekly or bi-weekly:**
    - prune old episodic logs beyond retention window (or archive).
    - rotate logs and compress older files.
- **Monthly:**
    - run security scans on containers and dependencies.
    - re-run full evaluation suites to detect drift.
    - validate backups (configs, safety rules, models).

Capacity planning:

- Track:
    - storage growth per week,
    - GPU hours consumed,
    - memory node CPU usage.
- Adjust quotas and retention policies based on observed usage.

G.5 Operational Roles

Minimal roles (could be the same person in early stages):

- **System Owner / Architect:**
    - maintains the architecture, major design changes, and appendices.
- **Safety Owner:**
    - responsible for safety rule updates, safety regression suites, incident triage for safety-related issues.
- **Ops / Reliability Engineer:**
    - maintains SLURM scripts, cluster-side deployment, monitoring dashboards.
    - primary contact for availability incidents.
- **Experiment Leads / Users:**
    - design and run experiments using the system; must follow change-management and logging conventions.

Appendix H — Future Research Directions

Finally, this appendix recognizes that the architecture is both an **engineering blueprint** and a **research testbed**. It highlights promising research directions the system is designed to support.

H.1 Dual-Hemisphere & Polycentric Cognition

- **H1.1 Division of labor:**
  - Empirical study of how best to split responsibilities between LH and RH.
  - Explore different planning/perception decompositions and their impact on performance and safety.
- **H1.2 Emergent behavior with added agents:**
  - How do additional specialist agents (code planner, curriculum agent, safety tuner) change overall behavior?
  - Do stable collaboration patterns ("cognitive coalitions") emerge?
- **H1.3 Meta-cognitive governance:**
  - Evaluate schemes where meta agents not only critique plans but also **reconfigure** the cognitive architecture (which agents to call, with what priorities).

H.2 World Models & Simulation

- **H2.1 Hybrid simulation:**
  - Combine symbolic/planning-level and pixel-level world models; study tradeoffs between accuracy and speed.
- **H2.2 Generalization across tasks and envs:**
  - How well do world models trained on one task family transfer to others?
  - Can we learn task-agnostic "physics priors" that directly benefit safety?
- **H2.3 Uncertainty-aware prediction:**
  - Incorporate epistemic and aleatoric uncertainty into risk estimates.
  - Use uncertainty to decide when to simulate more vs. act.

H.3 Safety & Alignment Under Real Constraints

- **H3.1 Safety-performance frontier:**
    - Map how performance degrades as safety thresholds tighten.
    - Identify operating points that are robust to model drift.
- **H3.2 Adversarial robustness in embodied settings:**
    - Move beyond prompt-only attacks to sensor, environment, and multi-agent adversarial strategies.
- **H3.3 Learning safety from experience:**
    - Use episodic logs of near misses and incidents to automatically propose new rules and thresholds.
    - Study convergence properties and failure modes of such adaptive safety systems.

H.4 Metacognition & Introspective Tools

- **H4.1 Introspection vs. overhead:**
    - Quantify the benefit of different metacognitive strategies (self-critique, cross-checking, confidence estimation) vs. their computational cost.
- **H4.2 Explanation quality:**
    - Connect metacognitive traces to human operator understanding.
    - Evaluate what kinds of explanations actually help humans debug and trust the system.
- **H4.3 Meta-learning of thinking strategies:**
    - Learn policies over *cognitive actions*: when to plan more, simulate more, query memory, or ask meta.
    - Investigate whether the system can learn “how to think” more efficiently over time.

H.5 Lifelong Learning & Knowledge Management

- **H5.1 Skill discovery & composition:**
    - From raw episodes, automatically extract reusable skills and evaluate their long-term utility.

- **H5.2 Concept drift & memory consolidation:**
    - Study strategies for updating semantic memory to handle evolving environments without catastrophic forgetting or unbounded growth.
- **H5.3 Structured memory for multi-agent systems:**
    - As more agents are added, investigate which memory structures (DHT layouts, namespaces, schemas) support scalable, robust cooperation.

H.6 Human-in-the-Loop & Governance

- **H6.1 Human-guided curriculum:**
    - Use human feedback to shape task curricula and environment variations.
- **H6.2 Mixed-initiative planning:**
    - Let human operators intervene in planning and meta decisions; study how this affects safety and performance.
- **H6.3 Governance and auditability:**
    - Develop tooling for reconstructing “who did what when” within the cognitive system (agents, rules, models) for accountability and debugging.