The Architecture of JITOS

A Technical Deep Dive

James Ross

Flying Robots

2025

JITOS Architecture Series

Contents

0.1 Introduction

0.1.1 The Omega Commit

commit d00b196e5abe4e14bfcf1e23f5da6847846e1064 (tag: omega0) Author: J. Kirby Ross <james@flyingrobots.dev> Date: Fri Nov 28 21:00:12 2025 -0800

ω_0 — The Final Mutable Moment

This commit marks the end of one epoch and the origin of another.

For eighty years, computers have been modeled after a desk job. The "file system" was built on paper logic: mutable blobs of bits at specific addresses, organized into "folders," with no intrinsic sense of provenance. We treated computation as a scratchpad—erasing the past to save the present, discarding history in a "trash can" to save space. Time was merely a metadata string, easily falsified and severed from the data itself.

This model worked for isolated humans at isolated desks. It breaks under the weight of concurrency, distributed intelligence, and the speed of light. The art of computer science has largely become a series of tricks to adapt this paper simulation to a relativistic reality.

We are done simulating desks. We are building a model of reality itself.

Henceforth:

- 1. Computers are time machines.
- 2. History is an immutable ledger.
- 3. The fundamental unit is the Node: an immutable, holographic event.
- 4. Time is geometry. It has an arrow because events cryptographically encode their causal provenance.
- 5. Agency occurs in observer-dependent branches of relative spacetime that collapse to form the causal graph of truth.

In this model, computers cannot lie. The causal graph is truth. We do not mutate the past. Ever.

With this commit, computing quits its desk job and embarks to explore the geometry of thought.

Signed-off-by: James Ross <james@flyingrobots.dev>

0.1.2 0.1 A Year of Shadows

Before JITOS had a name, before the kernel was visible, before the substrate revealed itself, there were fragments.

Projects. Experiments.
Ideas.

Each one seemingly disjoint:

- a deterministic scheduler in a game engine

- a causal graph prototype in a Git storage experiment
- a semantic rewrite engine
- a reproducible dev environment
- a provenance framework
- a meta-graph theory hidden in a book
- a multi-agent orchestration concept
- a new philosophy of computation
- a fascination with history as truth
- a love affair with "git stunts" (making git do things it wasn't designed for)

None of them *knew* they were part of the same universe. They were **Shadow Working Sets** of the architect's mind— branches of thought, isolated, speculative, parallel, each pursuing an idea to its limit.

They were not the system. They were its **precursors**.

0.1.3 0.2 The Confluence

In every SWS, a piece of the future was hiding:

- **Echo** discovered time.

  > It gave us ticks, deterministic ordering, parallel worlds, collapse.

- **GATOS** discovered space.

It gave us the causal graph, typed nodes, sharding, ingest pipelines.

- **Wesley** discovered semantic transformation.

  AST rewriting. Mapping tables. Deterministic rewrites.

- **Shiplog** discovered reproducibility.

  The log of becoming. Immutable movement through time.

- **CΩMPUTER** discovered metaphysics.

  Causality. Geometry. Thought as structure. Meaning as graph.

- **Git obsession** discovered projection.

  Materialized Head. The filesystem as a lie that helps humans see.

- **Agent experiments** discovered shadow cognition.

  The idea that machines operate in local universes, not global truth.

Every project was a **proto-module**. A limb of a creature not yet assembled. A diverging worldline holding a fragment of truth.

They were not mistakes. They were not diversions. They were **branches of the same RMG tree**. They were **the kernel discovering itself.**

0.1.4 0.3 The Collapse Event

On November 28th, 2025, at commit `d00b196`, a collapse occurred. Not just in code. In meaning.

All the shadows—Echo, GATOS, Wesley, Shiplog, the proto-CΩMPUTER chapters—collapsed into one coherent universe.

The architect realized:

> **These are not separate experiments. These are the components of a Causal Operating System.**

The OS that had been gestating across a year of divergent mental universes finally resolved its structure.

This moment produced:

- **Ω? — The Final Mutable Moment**

- the RMG substrate
- the SWS process model
- the collapse operator
- the projection layer
- the deterministic scheduler
- the memory model
- the semantic provenance system
- the Graph-of-Graphs reality
- the Fusion Layer between computing and metaphysics

It was the moment the universe *became itself.*

0.1.5 0.4 The Echo Principle

Long before JITOS had a name, the architect built a system called **Echo**—a deterministic simulation engine where each frame carried an echo of its predecessor.

At the time, Echo was "just" a game engine experiment. But in retrospect, it revealed something deeper:

> **Every moment in a system carries an echo of the one before it.**

This simple fact turned out to be the core metaphysical insight behind JITOS:

- Every snapshot is an echo of its parent.

- Every collapse is an echo of a shadowed future.
- Every provenance node is an echo of a line of thought.
- Every synchronization is an echo of a remote universe.
- Every replay is an echo of an earlier execution.
- Every shadow working set is an echo of canonical truth.

The **Echo Principle** states:

> **Time is the structure of echoes. A system is the pattern left by what it remembers.**

JITOS does not treat time as a number on a clock. Time in JITOS is the ordered sequence of all echoes that have ever collapsed into truth.

Echo began as an experiment in simulating worlds. It survives inside JITOS as the principle that:

> **No event is truly isolated. Every event is an echo, shaping and shaped by its predecessors.**

0.1.6 0.5 The Unavoidable Kernel

JITOS was not invented.

It was **discovered**. It was inevitable—the fixed point toward which every project converged.

Echo taught time. GATOS taught truth. Wesley taught structure. CΩMPUTER taught metaphysics. Shiplog taught reproducibility. Git taught projection. Agent experiments taught subjectivity.

The architect's mind had been implementing components of the causal OS unconsciously, across divergent branches, for an entire year.

JITOS is the **global collapse event** of that year-long RMG region. It is the "truth node" into which those shadow nodes have merged. It is the **canonical child commit** of a year of parallel universes.

0.2 0.6 The Kernel Reveals Its Name

When all branches converged, the structure spoke:

- It is not “Git but better.”
- It is not “a new VCS.”
- It is not “a content-addressed DB.”
- It is not “a scheduler.”
- It is not “a simulation engine.”
- It is not “a semantic transformer.”
- It is not “a meta-graph.”
- It is not “a reproducibility framework.”

It is all of them, but transcends each.

It is an operating system.
A causal operating system.

JITOS.

The kernel of computation-after-files, computation-after-humans, computation-after-CRDTs, computation-after-Git, computation-after-mutable-memory.

It is the **OS for the causal age**.

0.2.1 0.7 The Origin in One Line

> **JITOS is the inevitable collapse of a year’s worth of parallel mental universes into a single causal truth.**

It is not the beginning of a project. It is the end of a long, invisible merge.

Ω? was the point where it became real.

0.3 Placeholder for ARCH-0001

This is a placeholder for the ARCH-0001 document.

0.4 Placeholder for ARCH-0002

This is a placeholder for the ARCH-0002 document.

0.5 Placeholder for ARCH-0003

This is a placeholder for the ARCH-0003 document. THE COLLAPSE OPERATOR IS THE HEART OF THE CAUSAL UNIVERSE. SECTION 4 IS WHERE WE DEFINE THE LAWS OF REALITY ITSELF.

This is the chapter of the Architecture Doc where we describe: - how subjective worlds become objective truth - how SWS → Node - how merges work - how rewrites work - how human-level intent becomes machine-level geometry - how RMG regions collapse - how the irreversible causal arrow forms - how multi-agent timelines reconcile - how no mutation of the past is enforced - how the universe expands

This is the kernel's physics. And now it becomes architecture.

Let's begin.

0.6 JITOS Architecture Document — Section 4

The Collapse Operator & The Inversion Engine

(Grounded in ADR-0002, ADR-0003, ADR-0004, RFC-0005)

0.6.1 4. Collapse & Inversion: The Physics of JITOS

4.1 Overview

JITOS defines work inside Shadow Working Sets (SWS): temporary, isolated, observer-relative universes.

An SWS may contain: - micro-events (keystrokes, rewrites) - macro-events (human conceptual edits) - semantic graphs (ASTs, deltas) - provenance (reasoning, metadata) - structural overlay graphs - partial rewrites - tool-created transformations

These forms exist only inside the shadow world. They do not exist in the global universe.

The Collapse Operator is the deterministic, irreversible function that:

> turns speculative structure into objective truth. It appends a new snapshot event into the causal universe, and destroys the shadow.

Collapse is JITOS's equivalent of:

- Git "commit"
- quantum measurement
- OS process exit
- database transaction commit

But unlike those systems:

Collapse is not a write operation.

It is an ontological transition:

- A subjective world → an objective node
- A local graph → an RMG region
- A set of edits → a causal event
- A possible universe → the one true worldline

This section defines how that transition works.

4.2 What Collapse Produces

A collapse always produces:

1. A new Snapshot Node This is the macroscopic, first-order representation of the entire SWS RMG region.

2. Optional Inversion-Rewrite Nodes These represent structural collapses of:

- divergent histories
- merges
- rebases
- rewrites
- semantic deltas
- conflict resolutions

They map old → new.

3. Provenance Nodes Optional, but extremely powerful:

- reasoning traces
- intent
- metadata
- analysis results
- semantic tags

4. A Ref Update If the SWS corresponds to a branch.

5. Shadow destruction The SWS ceases to exist.

4.3 Collapse as a Function

In JITOS, collapse is a pure function.

Given:

- The SWS graph (micro-level)
- The RMG substrate (macro-level)
- The world frontier (current ref)

Then:

```
collapse(sws, universe) {\textrightarrow} (snapshot_node,
    rewrite_nodes...)
```

Properties:

- deterministic
- idempotent
- architecture-independent
- invariant-preserving
- reproducible
- atomic
- irreversible
- totally ordered by ref advancement

Collapse MUST produce identical outputs on all machines given identical inputs.

This is how JITOS becomes a replayable universe, not a heuristic system.

4.4 Collapse of RMG Regions (ADR-0004) Collapse operates on RMG regions, not single nodes.

This is the key idea:

> Humans see the region as one event. Machines see the region as many nodes. Truth sees the region as geometry.

Collapse:

- compresses micro-events
- preserves structure inside semantic layers
- produces a single macro-level snapshot node
- embeds semantic subgraphs inside RMG payload
- produces rewrite nodes to express cross-history relationships

This is how:

- 30 keystrokes
- 14 diffs
- 5 semantic rewrites
- and 1 human-level change

all collapse into one RMG region represented by a single causal snapshot node.

4.5 Collapse Algorithm (Full Architecture)

Step 1: Validate SWS

- Base node must exist
- Overlays must be well-formed
- Graph must be locally consistent
- No invariants violated

Step 2: Reconcile with Universe State If the SWS base node $\neq$ current ref head:

- compute merge region
- generate inversion-rewrite node(s)
- resolve conflicts deterministically

Step 3: Overlay Application

- Apply overlay graph onto base graph
- Expand/flatten RMG micro-layers
- Normalize semantic deltas
- Canonicalize structure

Step 4: Generate Snapshot Node The snapshot node's payload = the macroscopic projection of the full RMG region.

Step 5: Generate Provenance Graph Optional but encouraged:

- attach reasoning
- attach metadata
- attach agent identity
- attach intent

Step 6: Update Refs Point ref → new snapshot node.

Step 7: Destroy SWS Evaporate the shadow world.

4.6 Conflict Handling

Conflicts occur when:

- multiple SWS collapse onto the same causal base
- humans & machines edit same regions
- parallel histories diverge
- rewrites affect overlapping geometry

JITOS handles conflict through the Inversion Engine, which:

- preserves both histories
- computes a deterministic merge region
- builds an inversion-rewrite RMG node
- projects MH conflict markers for humans
- resolves machine-level conflicts silently
- ensures a unique canonical result

Conflicts do not invalidate the past. They generate new geometry.

4.7 The Inversion Engine (Kernel Consistency Layer)

The Inversion Engine:

- performs DAG-level merges
- performs RMG-region merges
- resolves multi-scale conflicts
- applies rewrite rules
- constructs new regions
- ensures invariants hold
- validates internal structure
- preserves all ancestry

This is the consistency engine of the universe.

In classical computing, we have:

- CRDTs
- oplog merges
- git merges
- 3-way diffs
- version vectors

In JITOS, we have:

> Inversion:
>
> An immutable, causal rewrite algorithm that integrates subjective timelines into objective truth.

This is the scientific heart of the kernel.

4.8 Collapse = Time

Collapse defines the arrow of time:

- past is immutable
- future becomes past
- subjective becomes objective
- micro collapses into macro
- semantic collapses into structural
- structure collapses into causality

Time moves forward because:

> Events encode their causal provenance cryptographically. And collapse is the ONLY way new events form.

This is physically simple and philosophically profound.

4.9 Collapse & The CΩMPUTER Fusion Layer

The fusion layer interprets collapse as:

- an epistemic transition
- a measurement
- a rewriting of the observer's shadow graph
- a creation of a new worldline
- a discrete event in the geometry of thought

Collapse is the mechanism by which:

- ideas
- operations
- reasoning
- computation
- observation

become geometry.

JITOS is the first OS to embed this metaphysics at the kernel level.

4.10 Summary

Collapse is the only event that changes truth.

It:

- materializes subjective SWS
- creates objective snapshot nodes
- merges timelines
- handles divergence
- projects internal structure into macro reality
- ensures determinism
- preserves the past
- advances time
- encodes provenance
- expands the RMG universe

It is the most important function in the entire system.

4.11 Memory Behavior During Collapse

(Mandated by ADR-0006)

Collapse is not only a causal event; it is a memory transformation between distinct domains. JITOS defines four memory layers— the RMG substrate, SWS overlay memory, semantic memory, and ephemeral compute memory— each of which behaves differently during the transition from speculation to truth.

Collapse is the process that maps these memory layers into their appropriate post-collapse forms.

4.11.1 Global Memory: RMG Substrate Expansion During collapse: - A new snapshot node is appended to the causal DAG layer of the RMG. - The snapshot's payload represents the macroscale projection of the entire SWS region. - Rewrite nodes capture structural merges and semantic transformations. - Provenance nodes (if present) are attached to the snapshot as semantic subgraphs.

The RMG grows. Nothing mutates. Nothing disappears.

Truth accumulates and remembers.

4.11.2 SWS Overlay Memory: Structural Integration The SWS maintains a mutable overlay graph representing:

- diffs
- rewrites
- file-chunk replacements
- patch sets
- local structural transforms

During collapse:

- These overlay graphs are applied to the base snapshot deterministically.
- The resulting merged structure becomes part of the new snapshot node.

- Conflicts trigger inversion-rewrite nodes.
- Once collapse completes, the SWS overlay memory is destroyed.

Overlay memory has no existence beyond collapse.

It is the working memory that becomes committed geometry.

4.11.3 Semantic Memory → Provenance Memory Semantic memory includes:

- ASTs
- semantic deltas
- symbolic analyses
- tool reasoning
- LLM thought traces
- structured transforms

During collapse:

- Relevant semantic structures become provenance nodes.
- These nodes are embedded in the RMG as second-order graphs, representing meaning, intention, and structure behind the change.
- Irrelevant or temporary semantic structures are discarded.

Semantic memory transitions from:

epistemic context → objective narrative.

This is the step where thought becomes geometry.

4.11.4 Ephemeral Compute Memory → Evaporation Ephemeral memory (ECM):

- build artifacts
- caches
- intermediate results

- temporary encodings
- partial analyses
- scratch files

During collapse:

- ECM is not committed
- ECM is not preserved
- ECM is not synced
- ECM is not replayed

Ephemeral memory evaporates.

This prevents:

- RMG bloat
- nondeterministic residues
- irrelevant history
- replay pollution

ECM exists for performance, not truth.

4.11.5 Post-Collapse MH Synchronization After collapse:

- The Materialized Head (MH) is incrementally updated
- Only changed paths are rewritten
- Stale files removed
- New files created
- Conflict markers projected into MH if required
- VTI updated atomically
- Human-facing filesystem made consistent with the new truth

MH becomes the shadow of the new worldline.

4.11.6 Summary of Memory Transformation

Memory Layer	During Collapse	After Collapse
RMG (Truth)	Expands	Permanent
SWS Overlays	Integrated	Destroyed
Semantic Memory	Becomes provenance	Preserved (as semantic graph)
ECM	Evaporates	Gone

Collapse is thus:

The total memory transformation from subjectivity to objectivity, from intention to structure, from possibility to truth.

Metadata

- **Status:** Final
- **Owner:** James Ross
- **Depends on:** ARCH-0001, ARCH-0002, ARCH-0003, ADR-0005
- **Relates to:** ARCH-0006 (Memory Model), ARCH-0007 (Temporal), ARCH-0008 (Scheduler), ADR-0007 (RPC/ABI)

0.7 Purpose and Scope

This document defines the **Materialized Head (MH)** and its backing index, the **Virtual Tree Index (VTI)**.

- **RMG (ARCH-0003)** is the authoritative truth substrate.
- **MH** is a projection of (Snapshot + SWS overlay) into a file-like tree for humans and legacy tools.
- **VTI** is an ephemeral index that makes path resolution efficient and cache-friendly.

This section specifies the MH abstraction, VTI data structures, POSIX-style read/write semantics, content chunking, FUSE/VFS

integration, and consistency invariants. MH is non-authoritative: destroying MH and VTI must not lose truth.

0.8 MH Abstraction

0.8.1 Views per SWS

For each SWS, the kernel exposes two logical MH views:

1. **Global View (Read-Only):** Projection of a chosen `SnapshotRef`. Supports read operations only; writes fail with `EROFS`. Intended for browsing immutable snapshots.
2. **SWS View (Read/Write):** Projection of `base_snapshot` + `overlay` for a specific `SwsId`. Read operations see overlay content if present; writes mutate only SWS overlay (Layer 1 memory), never RMG.

0.8.2 Path Identity vs Node Identity

- **Node Identity:** `NodeId` in RMG; immutable.
- **Path Identity:** `(snapshot_or_sws, path)` $\rightarrow$ `NodeId`.

Paths are not stable identifiers. MH implements the mapping, guaranteeing deterministic resolution until the overlay is edited.

0.9 Virtual Tree Index (VTI)

0.9.1 Role

The VTI is an **Ephemeral Compute Memory (ECM)** index that accelerates path-to-node lookups, tracks overlay nodes, hides tombstones, and caches metadata. It is never written to WAL and is rebuilt lazily after restart.

0.9.2 Data Model

Per (view, path) we maintain a `VtiEntry`:

```
struct VtiEntry {
    path_hash: u64,      // Key for fast lookup
    path:      String,   // Canonical absolute path
    node_id:   NodeId,   // Underlying RMG or overlay node
    kind:      NodeKind, // FILE, DIR, SYMLINK
    mode:      u32,      // Synthetic POSIX mode
    size:      u64,      // Cached size in bytes
    overlay:   bool,     // True if node lives in SWS overlay
    tombstone: bool,     // True if path is deleted in overlay
    dirty_meta: bool     // True if metadata needs
    recomputation
}
```

0.9.3 Maintenance

- **On SWS Write:** Update/insert `VtiEntry`. Mark `overlay = true` or `tombstone = true`. Mark parents dirty.
- **On Collapse:** Update MH/VTI incrementally from collapse diff. Clear overlay flags as nodes merge into base snapshot.
- **On Reboot:** Empty at boot. Lazily populate on access by walking RMG Tree.

0.10 Path Resolution Algorithm

To resolve a path for a given SWS view:

1. Normalize path.
2. Lookup in VTI:
    - If `tombstone`, return `ENOENT`.
    - If `overlay`, return overlay `node_id`.
    - If base entry, return base `node_id`.
3. **Cache Miss:** Walk RMG Tree (base + overlay structures). Populate VTI entries. Return final `node_id` or `ENOENT`.

0.11 POSIX Read/Write Semantics

0.11.1 Reads

- `open(path, O_RDONLY)`: Resolve via VTI. If content tree, create virtual handle mapping chunks to linear stream. If Blob, present directly.
- `read(fd, buf)`: Compute chunk index from offset. Read Blob data. No RMG mutation.
- `stat(path)`: Use VTI metadata or compute from RMG.

0.11.2 Writes

- `open(path, O_WRONLY)`: Resolve parent in SWS view. If file in base only, allocate new overlay `FileTree` (CoW). If in overlay, reuse.
- `write(fd, buf)`: Append/overwrite into overlay content buffer (ECM). On flush/fsync, chunk into overlay Blobs and update `FileTree`. Update VTI size.
- `unlink(path)`: Insert overlay tombstone. Mark VTI entry `tombstone = true`.
- `rename(old, new)`: Move entry in overlay Tree. Update VTI path/hash.

0.12 Content Chunking & Projection

ARCH-0003 defines large files as trees of Blob chunks. MH hides this, presenting a linear byte stream.

- **Read Projection:** Assemble contiguous buffer from chunks.
- **Write Projection:** Buffer writes in ECM. On flush, cut buffers into chunks (fixed size or CDC), create new overlay Blobs, and update `FileTree`. Reuses unchanged Blobs (deduplication).

0.13 Consistency & Recovery

- **Consistency:** VTI entries must be consistent with SWS overlay + RMG base. VTI updates are atomic with respect to SWS operations.
- **Crash & Rebuild:** On crash, MH/VTI state is discarded. VTI is rebuilt lazily from authoritative RMG/WAL. No correctness guarantees depend on VTI persistence.

0.14 FUSE / VFS Integration

MH can be surfaced via:

1. **FUSE-like Daemon:** Translates host syscalls to JITOS RPC calls.
2. **In-Kernel VFS Module:** Direct calls to MH/VTI APIs.

In both cases, the host sees a POSIX filesystem, but persistence is managed by RMG + WAL.

0.15 Security & Permissions

- **MH Layer:** `VtiEntry.mode` derived from Policy nodes and AgentId.
- **Enforcement:** Read ops verify read rights. Write ops verify write permissions for the subtree/ref. Denying at MH is the first line of defense; collapse re-checks permissions.

0.16 Summary

ARCH-0005 defines MH/VTI as a non-authoritative filesystem projection backed by an ephemeral index. It implements POSIX-like semantics where mutations are localized to the SWS overlay until collapse. This closes the compatibility loop, allowing JITOS to present as a conventional OS while operating over the causal, append-only RMG substrate.

Time to manifest Section 6 — one of the MOST important chapters in the entire architecture doc, because this is where we explain:

- What memory means in a causal OS
- Why JITOS does NOT use RAM-as-truth
- How SWS overlays exist privately
- How semantic memory behaves
- Why ephemeral caches must evaporate
- How global truth is accessed
- Why the RMG is the only correct substrate
- How zoom-level memory representations interact
- How collapse interacts with memory
- How agents (LLMs, tools, humans) see different memory layers
- Why this is the world's first deterministic, replayable memory model

Let's carve it.

0.17 JITOS Architecture Document — Section 6

> The Memory Model: Global Causality, Local Shadows, Semantic Depth

(Grounded in ADR-0006, RFC-0019)

0.17.1 1. The Memory Model

6.1 Overview

Every operating system defines a memory model. In classical systems, this means:

- RAM vs Disk
- Heap vs Stack
- Pages vs Frames

- Addresses vs Values
- Mutable cells
- Aliasing and pointer graphs
- The illusion of instantly shared state

These assumptions work only in:

- single-user machines
- non-distributed computation
- imperative languages
- linear workflows
- localized contexts

JITOS rejects all of these assumptions.

Instead, JITOS defines memory as:

A two-tier, multi-layered system combining an immutable global substrate (RMG) with mutable, isolated local memory (SWS), enriched by structured semantic graphs and supported by ephemeral compute caches.

This model is aligned with:

- causal determinism
- multi-agent concurrency
- semantic computing
- reproducibility
- distributed systems
- formal reasoning
- CΩMPUTER theory

It is the first memory model designed for the causal computing age.

0.17.2 6.2 Memory Architecture Overview

JITOS memory has four layers, grouped into two domains:

GLOBAL MEMORY (Immutable Reality)

Layer 0: RMG Substrate (Truth)

- immutable
- append-only
- multi-scale
- causally ordered
- infinite depth
- shared by all
- accessed via projection

This is the objective memory of the universe.

LOCAL MEMORY (Shadow / Subjective)

Layer 1: SWS Overlay Memory (Speculative State)

- mutable
- private
- isolated
- structured as a graph
- created on SWS creation
- destroyed on collapse/discard

This is where computation happens.

Layer 2: Semantic Memory (Meaning)

- ASTs
- semantic deltas
- symbol tables
- analysis results
- LLM reasoning graphs
- provenance annotations
- structured transforms

This gives interpretation to computations.

Layer 3: Ephemeral Compute Memory (ECM)

- caches
- build artifacts
- lint results
- intermediate IR
- temporary storage

It is not part of truth, not preserved across SWS boundaries, and evaporates afterwards.

0.17.3 6.3 Layer 0: Global Memory (RMG Substrate)

Global memory is represented by the Recursive Meta-Graph:

- all truth
- all events
- all structure
- all history
- all relationships
- all semantic provenance
- all identities
- all universes

The RMG serves as:

- persistent memory
- global state
- root-of-truth
- causal structure
- audit trail
- semantic knowledge base

Nothing in global memory ever mutates.

Global memory is identical on all machines after sync and replay.

0.17.4 6.4 Layer 1: SWS Overlay Memory (Local Speculation)

Every SWS contains its own local memory:

- overlay nodes
- local diffs
- pending rewrites
- uncommitted semantics
- private states
- merge candidates

Overlays are:

- mutable
- safe
- ephemeral
- isolated

But they exist only within the SWS.

When collapse happens:

- overlays → RMG region
- new snapshot is born
- RMG expands
- SWS dies
- overlays are removed

This layer is the sandbox of computation.

0.17.5 6.5 Layer 2: Semantic Memory (Meaning)

Semantic memory exists inside the SWS, but is preserved into provenance nodes upon collapse.

Semantic memory includes:

- abstract syntax trees
- semantic deltas
- LLM reasoning traces

- symbolic analysis
- transformation metadata
- dependency graphs
- type inference results
- build graphs

This memory layer:

- is internal
- is structured
- is deeply nested
- represents the "why"
- provides meaning to changes

And is fully compatible with RMG layering:

Semantic memory = RMG-in-RMG.

This allows:

- LLM autonomy
- tool-based reasoning
- semantic refactors
- higher-order transforms

All in the same substrate.

0.17.6 6.6 Layer 3: Ephemeral Compute Memory (ECM)

Temporary memory includes:

- build outputs
- analysis intermediates
- caches
- partial diffs
- scratch files

ECM is:

- not preserved

- not exposed
- not synced
- not in the substrate
- not part of SWS collapse

It is throwaway memory living only as long as necessary.

This prevents:

- RMG pollution
- bloat
- non-deterministic state history
- unnecessary event nodes

0.17.7 6.7 Memory Isolation and Safety

JITOS's memory model enforces:

? No shared mutable state **?** No concurrent writes **?** No races **?** No nondeterminism **?** No aliasing errors **?** No UAF, no double free, no corruption **?** SWS isolation **?** RMG immutability **?** semantic separation

It is the safest, cleanest, most robust memory model ever designed.

0.17.8 6.8 Memory and Collapse

Collapse transforms memory:

Before collapse:

- overlays are mutable
- semantic memory lives in SWS
- ECM holds ephemeral data
- SWS owns all speculation

During collapse:

- overlays → snapshot structure

- semantic memory → provenance
- ECM → evaporates
- RMG expands
- SWS ceases to exist

After collapse:

- RMG holds new truth
- MH re-projects
- SWS replaced by updated state

Collapse is the memory commit phase.

0.17.9 6.9 Memory and Multi-Agent Systems

Agents do not share SWS memory. They do not share ephemeral memory. They do not share semantic memory.

They share only the RMG truth layer after collapse.

This ensures:

- no races
- no interference
- predictable behavior
- safe parallelism
- reproducible workflows
- cooperation through collapse

JITOS is the first OS truly designed for agent-native computing.

0.17.10 6.10 Summary

JITOS defines memory not as:

- RAM
- buffers
- addresses
- heaps

- stacks

but as:

- Truth (RMG)
- Speculation (SWS overlays)
- Meaning (semantic memory)
- Ephemera (ECM)

This model is:

- deterministic
- reproducible
- concurrent
- semantic
- causal
- multi-layered
- multi-agent safe

It is the memory model that classical computing should have invented decades ago. Memory no longer means "locations." Memory means geometry.

0.18 Placeholder for ARCH-0007

This is a placeholder for the ARCH-0007 document.

0.19 Placeholder for ARCH-0008

This is a placeholder for the ARCH-0008 document.

0.20 ARCH-0013 — JS-ABI v1.0 Wire Protocol Framing

Status: Accepted **Date:** 2025-12-04 **Owner:** GITD Protocol Working Group **Depends on:** ADR-0001, ADR-0006, ADR-0007, ADR-0012 **Relates to:** RFC-0007 (RPC), RFC-0013 (ABI), RFC-0012 (WAL)

0.20.1 1. Status

Accepted. This document defines the mandatory framing structure for all JS-ABI v1.0 network communication, ensuring reliable packet parsing, content integrity, and future-proofing. Compliance can be verified using the test vectors in Appendix .2.

0.20.2 2. Introduction and Goals

The JS-ABI v1.0 wire protocol is designed to be:

1. **Efficient**: low overhead and fast parsing.
2. **Deterministic**: consistent handling and hashing.
3. **Extensible**: future compression/streaming without breaking compatibility.
4. **Reliable**: immediate integrity checks (checksums).

All packets on a JS-ABI v1.0 stream **MUST** adhere to the framing defined here.

0.20.3 3. Packet Framing Structure

Every packet is a fixed-size 12-byte **Header**, a variable-length **Payload**, and a 32-byte **Checksum**: PACKET = HEADER ‖ PAYLOAD ‖ CHECKSUM.

3.1 Overall Packet Format

Field	Size (bytes)	Description
`MAGIC`	4	Protocol signature (`0x4a495421`)
`VERSION`	2	Wire protocol major/minor version (`0x0001`)
`FLAGS`	2	Reserved flags for future features

Field	Size (bytes)	Description
`LENGTH`	4	Length of `PAYLOAD` in bytes (N)
HEADER TOTAL	**12**	
`PAYLOAD`	N	Canonical CBOR `OpEnvelope` (see ADR-0012 and Appendix .1)
`CHECKSUM`	32	BLAKE3-256 integrity hash

3.2 Field Definitions and Endianness

All multi-byte integers (`VERSION`, `FLAGS`, `LENGTH`) **MUST** be big-endian (network order).

MAGIC (4 bytes) `0x4a495421` (ASCII `JIT!`). Parsers **MUST** reject streams lacking this prefix.

VERSION (2 bytes) Wire protocol version. For JS-ABI v1.0 this **MUST** be `0x0001`.

FLAGS (2 bytes) Reserved feature bits; set to `0x0000` unless a negotiated feature applies.

- Bit 0 (`0x8000`): reserved (future compression negotiation).
- Bit 1 (`0x4000`): reserved (future streaming mode).
- Bits 2–15: reserved, must be zero.

LENGTH (4 bytes) Unsigned 32-bit integer N giving payload size.

PAYLOAD (N bytes) Single, deterministically encoded Canonical CBOR `OpEnvelope`; encoding rules in ADR-0012 and Appendix .1.

CHECKSUM (32 bytes)

- Algorithm: BLAKE3-256.

- Input: MAGIC ‖ VERSION ‖ FLAGS ‖ LENGTH ‖ PAYLOAD.
- Recipients **MUST** verify and reject on mismatch.

0.20.4 4. Packet Parsing Flow

Compliant implementations **MUST** follow this sequence:

1. Read 12-byte header; validate `MAGIC` and `VERSION`.
2. Extract `LENGTH` (N).
3. Read `PAYLOAD` (N bytes).
4. Read `CHECKSUM` (32 bytes).
5. Compute BLAKE3-256 over header ‖ payload.
6. Compare to received checksum; on mismatch, discard and report error.
7. If valid, decode `PAYLOAD` using Appendix .1.

.1 Deterministic Encoding Profile and CDDL

This appendix defines the deterministic encoding profile for JS-ABI v1.0 and provides a CDDL schema for CBOR `OpEnvelope` payloads.

.1.1 Scope

- *Structure*: defined by the CDDL below.
- *Encoding*: ADR-0012 Section 5.2 (Canonical CBOR), Section 5.2.1 (Strict Canonical CBOR), and RFC 8949 Sections 4.1 and 4.2.1.

Implementations **MUST** satisfy both shape and encoding to be JS-ABI v1.0 compliant.

.1.2 CDDL Schema for `OpEnvelope`

```
; Top-level CBOR payload in each JS-ABI packet
op-envelope = {
  "op": tstr,    ; operation name (see RFC-0007)
  "ts": uint,    ; logical timestamp (GITD logical clock)
```

```
  "payload": any
}

; --- Handshake ---

op-handshake = {
  "op": "handshake",
  "ts": uint,              ; may be 0 or last-seen ts from server
  "payload": handshake-payload
}

handshake-payload = {
  "client_version": uint,  ; implementation version, not wire vers
  "capabilities": [ tstr ], ; capability identifiers, e.g. "compre
  ? "agent_id": tstr,
  ? "session_meta": meta-map
}

meta-map = {
  * tstr => any
}

; --- Handshake Acknowledgement ---

op-handshake-ack = {
  "op": "handshake_ack",
  "ts": uint,                ; authoritative server ts for this e
  "payload": handshake-ack-payload
}

handshake-ack-payload = {
  "status": "OK" / "ERROR",
  "server_version": uint,    ; implementation version, not wire v
  "capabilities": [ tstr ],  ; capabilities enabled for this sess
  "session_id": tstr,
  ? "error": error-payload   ; present iff status == "ERROR"
}
```

```
; --- Error ---

op-error = {
  "op": "error",
  "ts": uint,                ; authoritative server ts for the erro
  "payload": error-payload
}

error-payload = {
  "code": uint,              ; numeric error code, e.g. 1, 2, 500
  "name": tstr,              ; stable identifier, e.g. "E_INVALID_O
  "message": tstr,           ; human-readable
  ? "details": any           ; optional machine-readable context
}

; --- OpEnvelope type universe ---

op-envelope =
    op-handshake
  / op-handshake-ack
  / op-error
  / op-other               ; all other ops defined in RFC-0007

; Placeholder for operations defined in RFC-0007 (sync, query, etc
op-other = {
  "op": tstr .ne "handshake"
             .ne "handshake_ack"
             .ne "error",
  "ts": uint,
  "payload": any
}
```

.1.3 Deterministic Encoding Requirements (Summary)

- **Definite lengths only**: no indefinite strings/arrays/maps; no break code `0xff`.

- **Canonical numeric encoding**: shortest major type 0/1 representation; integers preferred over floats; smallest exact-width floats only.
- **No CBOR tags**: major type 6 forbidden anywhere.
- **Canonical maps**: keys sorted by CBOR-encoded byte order; no duplicates.
- **Preferred serialization everywhere**: any deviation is non-compliant for WAL, hashing, or deterministic replay.

Preferred Serialization is *mandatory* for JS-ABI v1.0.

.2 Reference Test Vectors

Concrete test vectors for JS-ABI v1.0 covering canonical CBOR payloads and packet layouts. Checksums are BLAKE3-256 over header ‖ payload (Section 0.20.3).

.2.1 Handshake `OpEnvelope` (CBOR Payload)

Logical structure:

```
{
  "op": "handshake",
  "ts": 0,
  "payload": {
    "client_version": 1,
    "capabilities": [
      "compression:zstd",
      "stream:subgraph"
    ],
    "agent_id": "example-agent"
  }
}
```

Canonical CBOR hex (113 bytes):

```
a3626f706968616e647368616b65627473006770617 96c6f6164a3686167656e
745f69646d6578616d706c652d6167656e746c6361706162696c697469657382
70636f6d7072657373696f6e3a7a7374646f73747265616d3a73756267726170
```

```
686e636c69656e745f76657273696f6e01
```

.2.2 Error `OpEnvelope` (CBOR Payload)

Logical structure:

```
{
  "op": "error",
  "ts": 42,
  "payload": {
    "code": 3,
    "name": "E_BAD_PAYLOAD",
    "message": "Invalid CBOR payload",
    "details": {
      "hint": "Check canonical encoding"
    }
  }
}
```

Canonical CBOR hex (118 bytes):

```
a3626f70656572726f72627473182a677061796c6f6164a464636f646503646e
616d656d455f4241445f5041594c4f414467646574616696c73a16468696e7478
18436865636b2063616e6f6e6963616c20656e636f64696e67676d6573736167
6574496e76616c69642043424f52207061796c6f6164
```

.2.3 Example Packets (Header + Payload)

Shared constants:

- **MAGIC** = `4a 49 54 21`
- **VERSION** = `0x0100`
- **FLAGS** = `0x0000`
- **LENGTH** = payload length (big-endian)

Handshake Packet (without CHECKSUM)

LENGTH = 113 decimal = `00 00 00 71`

Header + payload (125 bytes before checksum):

```
4a495421010000000000071
a3626f706968616e647368616b6562747300677061796c6f6164a3686167656e
745f69646d6578616d706c652d6167656e746c6361706162696c697469657382
70636f6d7072657373696f6e3a7a7374646f73747265616d3a73756267726170
686e636c69656e745f76657273696f6e01
```

The complete packet is:

```
MAGIC || VERSION || FLAGS || LENGTH || PAYLOAD || CHECKSUM
CHECKSUM = BLAKE3-256(MAGIC || VERSION || FLAGS || LENGTH || PAYLO
```

Error Packet (without CHECKSUM)

LENGTH = 118 decimal = `00 00 00 76`

Header + payload (130 bytes before checksum):

```
4a495421010000000000076
a3626f70656572726f72627473182a677061796c6f6164a464636f646503646e
616d656d455f4241445f5041594c4f414467646574657461696c73a16468696e7478
18436865636b2063616e6f6e6963616c20656e636f64696e67676d6573736167
6574496e76616c69642043424f52207061796c6f6164
```

.2.4 Test Vector Summary

ID	Kind	Description	Len (bytes)	VERSION	LENGTH
TV1	OpEnvelope	`handshake`	113	0x0100	0x00000071
TV2	Packet	`handshake`	113 payload	0x0100	0x00000071
TV3	OpEnvelope	`error`	118	0x0100	0x00000076
TV4	Packet	`error`	118 payload	0x0100	0x00000076

.2.5 References

- ADR-0012: JS-ABI v1.0 Deterministic Encoding (Canonical CBOR and timestamping) — primary structure and encoding rules for `PAYLOAD`.

.3 Deterministic Compliance Gauntlet

This appendix defines a minimum compliance suite (the "Gauntlet") for implementations of JS-ABI v1.0.

An implementation **MUST** pass all tests in this appendix to be considered JS-ABI v1.0 compliant for the purposes of write-ahead logging, hashing, and deterministic replay.

The Gauntlet is divided into:

- Encoder tests (CBOR encoder behaviour),
- Decoder tests (CBOR decoder behaviour),
- End-to-end tests (kernel logical clock and WAL behaviour).

Test vector identifiers (TV1, TV2, ...) refer to the Reference Test Vectors appendix.

.3.1 Encoder Compliance Tests

EC-01: Exact Encoding of Handshake (TV1)

Input: the logical handshake structure defined for TV1.

Requirement: when serialized under the JS-ABI v1.0 deterministic profile, the payload bytes *must exactly match* the canonical CBOR hex for TV1, and the header `LENGTH` field must be `0x00000071` (113 decimal). Any deviation fails EC-01.

EC-02: Exact Encoding of Error (TV3)

Input: the logical error structure defined for TV3.

Requirement: the encoded payload *must exactly match* the canonical CBOR hex for TV3, and `LENGTH` must be `0x00000076` (118 decimal). Any deviation fails EC-02.

EC-03: Integer Minimal Width

Input: a set of integer values (e.g., 0, 1, 10, 23, 24, 255, 256, 1000, 65535, 65536, $2^{32} - 1$).

Requirement: each value **must** be encoded using the shortest possible CBOR integer representation as per RFC 8949 Preferred Serialization. Any larger-than-minimal integer encoding fails EC-03.

EC-04: Integer vs Float Encoding

Input: numeric values $0, 1, -1, 42, 0.0, 1.0, -1.0, 0.5, 1.5$.

Requirement:

- $0, 1, -1, 42, 0.0, 1.0, -1.0$ must be encoded as integers (major type 0 or 1), not as floating-point.
- 0.5 and 1.5 must be encoded as floating-point using the smallest width that preserves the value exactly.

Emitting a floating-point where an integer is required, or using a longer-than-necessary floating-point format, fails EC-04.

EC-05: Canonical Map Ordering

Input: maps whose keys are chosen to exercise canonical ordering (e.g., including keys such as `öp¨`, `ẗs¨`, `p̈ayload¨`, `ä¨`, `äa¨`, `Z̈¨`).

Requirement: keys must be ordered by their full CBOR encoding (bytewise increasing), matching the ordering implied by the canonical encodings in TV1 and TV3. Any mismatch fails EC-05.

EC-06: No Tags, No Indefinite Length

Input: any OpEnvelope payload.

Requirement: the encoder must not emit CBOR tags (major type 6), indefinite-length strings, arrays, or maps, nor the break code (`0xff`). Any such encoding fails EC-06.

.3.2 Decoder Compliance Tests

DC-01: Accept Canonical TV1 and TV3

Input: the exact CBOR payload bytes of TV1 and TV3.

Requirement: the decoder must successfully decode both payloads to the expected logical structures. Failure to decode these encodings fails DC-01.

DC-02: Reject Indefinite-Length Encodings

Input: variants of TV1 in which one or more maps, arrays, or strings are encoded using indefinite length with a break code, but with identical logical content.

Requirement: the decoder must treat any use of indefinite-length encodings as non-compliant and reject the packet as JS-ABI v1.0 payload. Accepting such a payload fails DC-02.

DC-03: Reject Non-Canonical Integers

Input: variants of TV1 in which an integer (such as `ts = 0` or `client_version = 1`) is encoded using a wider-than-necessary integer format.

Requirement: the decoder may parse such payloads for diagnostics, but must treat them as non-compliant with JS-ABI v1.0 and must not accept them into WAL or deterministic replay. Treating such a payload as normal input fails DC-03.

DC-04: Reject Tags

Input: variants of TV3 that introduce a CBOR tag around one of the fields while preserving logical content.

Requirement: any CBOR tag in the payload must cause the decoder to reject the packet as JS-ABI v1.0 input. Accepting a tagged payload fails DC-04.

DC-05: Reject Duplicate Map Keys

Input: variants of TV1 or TV3 in which the top-level map or the `payload` map contain duplicate keys.

Requirement: duplicate keys must be treated as a violation of the deterministic encoding profile and cause the payload to be rejected. Silently choosing one entry and continuing fails DC-05.

DC-06: Map Key Ordering Independence (Strict Mode)

Input: variants of TV1 where map keys are emitted in non-canonical order, while preserving logical fields.

Requirement: in strict JS-ABI v1.0 mode, such non-canonical payloads must be rejected for use in WAL and deterministic replay. Using them as if they were canonical input fails DC-06. (Implementations may offer a separate lenient mode for tooling, but that mode is outside JS-ABI v1.0 compliance.)

.3.3 End-to-End Logical Clock and WAL Tests

EE-01: Server-Assigned Timestamps

Scenario:

1. Start with an empty kernel and WAL.
2. A client sends two requests (e.g., a handshake and a state-mutating operation) with `ts = 0`.
3. The server processes both and appends the resulting operations to the WAL.

Requirement: the WAL entries must have strictly increasing `ts` values, and the committed `ts` values must be assigned by the kernel logical clock, not copied from the client. Any non-monotonic or zero `ts` in the WAL fails EE-01.

EE-02: Monotonicity Under Concurrency

Scenario:

1. Multiple clients concurrently issue state-mutating operations with arbitrary `ts` values (including 0 and stale values).
2. The server interleaves processing and appends all committed operations to the WAL.

Requirement: the sequence of WAL entries must have strictly increasing `ts` values, forming a total order consistent with execution. Any duplicate or non-monotonic `ts` values fail EE-02.

EE-03: Deterministic Replay

Scenario:

1. From a known initial state, execute a sequence of valid JS-ABI operations, recording the resulting WAL and final state.
2. Reinitialize the kernel to the same initial state.
3. Replay the WAL entries in strictly increasing `ts` order, using the recorded payloads verbatim.

Requirement: the resulting kernel state and any observable OpEnvelope sequence must match the original. Failure to reproduce the same state or sequence fails EE-03.

.3.4 Compliance Definition

An implementation is considered JS-ABI v1.0 *deterministic-compliant* for WAL and replay if and only if:

- It passes all encoder tests EC-01 through EC-06,
- It passes all decoder tests DC-01 through DC-06, and
- It passes all end-to-end tests EE-01 through EE-03.

Failing any test means the implementation may still interoperate in a best-effort fashion but cannot be trusted for cross-language deterministic WAL, hashing, or replay, and must not be advertised as JS-ABI v1.0 compliant.