Request for Comments

Proposals and Specifications

James Ross

Flying Robots

2025

JITOS Documentation Series

Contents

Chapter 1

RFC Log

1.1 RFC-0022

JIT RFC-0024

CΩMPUTER Fusion Layer (CFL v1.0)

Unifying the OS Kernel and the Metaphysics of Computation

Status: Draft
Author: James Ross
Contributors: JIT Community
Requires: - RFC-0001 through RFC-0021 - CΩMPUTER Volume I: The Geometry of Thought
Start Date: 2025-11-30
Target Spec: JITOS v1.0**
License: TBD

---

1. Summary

This RFC defines the Fusion Layer between:

JIT

The operating system kernel based on: - causal DAGs - shadow worldlines - inversion semantics - collapse physics - immutable truth - deterministic execution

CΩMPUTER

The metaphysical foundation of computation based on: - geometry - causality - observer-relative shadows - invariant forms - worldline computing - semantic structure - universal rewrite theory

This RFC unifies them into a single conceptual + technical model.

---

2. Motivation

JIT is a mechanical implementation of laws.
CΩMPUTER is the philosophical model of those laws.

One without the other is incomplete: - CΩMPUTER needs machinery. - JIT needs meaning.

The Fusion Layer organizes how the high-level CΩMPUTER concepts map down to: - node structures - shadow semantics - collapse rules - provenance nodes - identity and agency - causality invariants - distributed sync - universe federation

This RFC formalizes that mapping.

---

3. The Fusion Diagram

CΩMPUTER (Metaphysics)
?
??? Ontology of Forms → Node Types
??? Epistemic Shadows → SWS Semantics
??? Collapse of Potential → Commit Operator
??? Geometry of Thought → Provenance Nodes
??? Causal Spacetime → DAG Invariants
??? Relative Observation → MH Projection
??? Many-Worlds / Frames → Multi-Shadow Parallelism

JIT (Physics & Kernel)

Every CΩMPUTER concept becomes a JIT subsystem.

---

4. Mapping the Core Concepts

4.1 Forms → Nodes

CΩMPUTER says:

> All information exists as Forms, immutable patterns.

JIT implements:

```
Node {
id,
type,
parents,
payload,
metadata
}
```

Nodes are Forms.

4.2 Shadows → SWS

CΩMPUTER: > Reality appears to observers through shadows.

JIT:
Shadows = SWS, isolated epistemic worldlines.

Agents live inside Shadows.

4.3 Collapse → Commit

CΩMPUTER: > Potential becomes actual when observed.

JIT:
Commit collapses shadow states into real snapshot nodes.

This is identical structurally.

4.4 Geometry → Provenance

CΩMPUTER: > Meaning is encoded as geometry.

JIT:
Semantic provenance nodes attach meaning to events.

4.5 Causal Universe → DAG

CΩMPUTER: > Reality is a causal graph.

JIT:
Reality IS the DAG.
Nodes are events.
Edges are causality.

4.6 Observer Frames → MH & SWS Views

CΩMPUTER: > Different observers see different slices.

JIT:
Materialized Head for humans.
SWS for agents.
Projection = perception.

4.7 Multi-Reality → Multi-Shadows

CΩMPUTER: > Every agent observes its own reality.

JIT:
Multiple SWS coexist, all valid.

4.8 Worldline Consistency → Inversion Engine

CΩMPUTER: > Conflicts reconcile to preserve global structure.

JIT:
Inversion Engine merges worldlines deterministically.

1.1.1 4.9 Federation → Inter-Universe Causality

CΩMPUTER: > Multiple universes may contact each other.

JIT:
RFC-0021 federation protocol.

1.2 5. The Unified Model

We now define the formal fusion:

Definition: CΩMPUTER is the semantic meta-graph.

Definition: JIT is the physical implementation of that meta-graph.

Definition: The Fusion Layer binds them.

This allows:

- generic agents
- symbolic reasoning
- ontological debugging
- universal provenance
- simulation inside simulation
- high-dimensional introspection
- next-generation OS integrations

This RFC is basically the Riemannian geometry of JITOS.

1.3 6. Use Cases Enabled

1.3.1 6.1 ML/AI Thought + Action Alignment

Thought (provenance) and action (nodes) unified.

1.3.2 6.2 Symbolic + Substrate Fusion

LLMs reason in CΩMPUTER semantics.
Kernel enforces physical law.

1.3.3 6.3 Scientific Cosmology Models

Simulations map directly to the DAG fabric.

1.3.4 6.4 Code as Narrative

Every change is a story with meaning.

1.3.5 6.5 Next-Gen Languages

Languages built on causal semantics.

1.4 7. Why This Matters

Because CΩMPUTER is:

- the conceptual model
- the theory of truth, causality, and shadow
- the mathematical framework

And JIT is:

- the OS kernel
- the execution engine
- the practical machinery

The Fusion Layer is where:

THEORY ↔ PRACTICE
PHILOSOPHY ↔ ARCHITECTURE
GEOMETRY ↔ CODE
COSMOS ↔ KERNEL

meet.

This turns JITOS into:

- a universal operating system
- a cognitive substrate
- a computational physics engine
- a semantic reasoning environment
- a multi-agent collaboration universe
- a new field of computer science

And CΩMPUTER becomes:

- the theory behind it
- the mathematical language
- the epistemic model
- the metaphysical justification
- the narrative lens
- the cognitive interface

Together?

This is not an OS.
Not a VCS.
Not a DB.
Not a framework.

This is a New Paradigm.

1.5 CΩMPUTER • JITOS

1.6 JIT RFC-0001

1.6.1 Node Identity & Canonical Encoding (NICe v1.0)

1.7 1. Summary

This RFC defines:

- **how nodes are identified**
- **how nodes are serialized**
- **how node integrity is guaranteed**
- **how node equality is evaluated**
- **how canonical encoding is achieved**

This establishes the foundational invariant of JIT:

> **Node identity is a pure function of its content and parent list.**

This RFC is the ground truth that allows:

- determinism
- idempotence
- reproducibility
- causal ordering
- infinite auditability
- distributed correctness

Without this, JIT collapses.

With it, JIT becomes physically sound.

1.8 2. Motivation

JIT relies on an immutable, append-only DAG.

This requires:

- stable identity
- canonical encoding
- deterministic hashing
- cross-machine consistency
- architecture independence

If ANY machine in the network produces different node IDs for the same logical content, **the universe fractures.** So this RFC formalizes the physics of node identity.

1.9 3. Requirements

Node identity must be:

1.9.1 Deterministic

Same input → same output, across all architectures.

1.9.2 Order-sensitive

Parents must be hashed in deterministic order.

1.9.3 Type-sensitive

Different node types produce different encodings.

1.9.4 Encoding-stable

Versioning cannot alter identity retroactively.

1.9.5 Hash-secure

Strong cryptographic guarantees.

1.9.6 Provenance-faithful

Parent list is required and must not be empty (except for the root node).

1.10 4. Canonical Node Structure

All nodes have the structure:

```
Node {
    type: NodeType
    parents: list<NodeID>
    payload: bytes
    metadata: Metadata
}
```

Metadata fields:

- timestamp (logical + wall clock)
- author ID
- signature (optional)
- compression flags
- schema version

Metadata must be strictly ordered.

1.11 5. Canonical Serialization Format

JIT **MUST** use **canonical CBOR** (RFC 8949 §4.2):

- sorted map keys
- minimal integer encoding
- UTF-8 canonical strings
- deterministic floats
- no ambiguous structures

This ensures:

- architecture-independent serialization
- byte-stable hashing
- reproducibility across time

1.12 6. Hashing Algorithm

JIT **MUST** use **BLAKE3-256** as the node identity hash.

Justification:

- extremely fast
- cryptographically strong
- parallelizable
- tree-hash capable
- flexible
- future-proof

```
NodeID = blake3(canonical_encoding(node)).
```

1.13 7. Parent Ordering Rule

Parents **MUST** be sorted lexicographically by NodeID prior to serialization.

This ensures:

- determinism
- order independence of logical causal descent
- reproducible merges
- identical rewrites under identical conditions

1.14 8. Signature Layer (Optional)

Signatures are not part of NodeID.

They are stored in metadata.

This ensures:

- same node ≠ same identity as signed node
- signatures don’t alter causal truth

Node signature **MUST** be:

```
sig = sign(private_key, NodeID)
```

1.15 9. Root Node

The root node (the initial event of the universe):

```
type = "root"
parents = []
payload = empty
metadata = minimal
```

Hashing rules apply; this produces a deterministic root ID.

1.16 10. Backwards Compatibility

If future schemas add fields:

- new fields **MUST** serialize to canonical defaults (0/null/empty)
- missing fields **MUST** be interpreted as canonical default
- **identity cannot change retroactively**

We do not mutate the past.

Ever.

1.17 11. Security Considerations

- BLAKE3-256 ensures strong collision resistance.
- CBOR canonicalization ensures no cross-machine divergence.

- Parent sorting ensures rewrite determinism.
- Metadata versioning ensures forward compatibility.
- Signatures separate identity from attestation.

This RFC provides **cryptographic grounding** for the entire JIT universe.

1.18 12. Status & Next Steps

Once accepted, RFC-0001 becomes:

- mandatory for JITD prototypes
- referenced by RFC-0002 (DAG Invariants)
- foundational for JITOS v0.1
- the backbone of all future JIT systems

1.19 CΩMPUTER • JITOS

1.20 JIT RFC-0002

1.20.1 Causal DAG Invariants (CDI v1.0)

1.21 1. Summary

This RFC defines the **global structural invariants** that govern the JIT causal substrate (GATOS-Core DAG). These invariants ensure:

- deterministic behavior
- irreversible history
- causality preservation

- safe concurrency
- shadow isolation semantics
- conflict-free merges
- correct replay
- global consistency

These rules form the “physics” of the JIT universe.

Everything: nodes, shadows, inversion, commits, sync. Everything depends on these invariants being preserved at all times.

1.22 2. Motivation

A causal DAG is only as correct as the invariants that define it.

Break an invariant →

you break causality →

you break determinism →

you break reproducibility →

you break the entire universe.

Therefore, JIT requires mathematical guarantees analogous to physical laws:

- No cycles
- No deletion
- No mutation of past
- Monotonic growth
- Deterministic edges
- Append-only temporal extension

These invariants ensure:

- Shadow Working Sets are safe
- commits collapse unambiguously
- inversion rewrites do not corrupt the universe
- distributed sync is correct

- conflict resolution is deterministic
- the DAG remains canonical across machines

Without this RFC, JIT is just “a cool Git idea.” With it, JIT becomes a **causal operating system**.

1.23 3. Invariant Definitions

Below are the **non-negotiable** invariants of the GATOS-Core causal DAG.

Each **MUST** hold at all times.

1.24 Invariant 1 — Acyclicity (No Cycles)

> **The causal graph *MUST* be a Directed Acyclic Graph.**

No node may reference itself as an ancestor, directly or indirectly.

Why:

Cycles would imply a state is both cause and effect of itself → paradox.

Implications:

- deterministic replay
- conflict-free topological ordering
- causal consistency
- no temporal violations

1.25 Invariant 2 — Append-Only (No Deletion)

> **Once a node exists, it *MUST NEVER* be altered or removed.**

Past is immutable.

Only new nodes can be appended.

Why:

Rewrite = “new history”, not “edited history.”

Implications:

- perfect reproducibility
- infinite audit trail
- deterministic cross-machine sync
- time is monotonic

1.26 Invariant 3 — No Mutation (Immutability)

> **Nodes *MUST* be immutable after creation.**

Payload? Immutable.

Metadata? Immutable.

Parents? Immutable.

Why:

Mutation breaks identity.

Identity breaks hashing.

Hashing breaks causality.

Causality breaks the universe.

1.27 Invariant 4 — Causal Completeness (Parents First)

> **A node *MUST NOT* exist unless all its parents already exist.**

This ensures that every event has its dependencies fully realized.

Why:

Guarantees proper topological ordering. Allows deterministic reconstruction.

1.28 Invariant 5 — Monotonic Event Ordering

> **Adding a node *MUST* increase the causal "size" of the universe.**

I.e.

the DAG must grow, not shrink or mutate.

Why:

Supports WAL replay, invariance under distribution, and time semantics.

1.29 Invariant 6 — Parent List Canonicalization

> **Parents of a node *MUST* be lexicographically sorted by NodeID.**

Required to enforce deterministic encoding and hashing.

Why:

Two machines must produce identical results for identical inputs.

1.30 Invariant 7 — Single Universe (Global Graph)

> **There *MUST* be one and only one global DAG per repository.**

Shadows (SWS) represent *subjective frames*,

but commit events collapse into **one global worldline**.

Why:

- avoids universe bifurcation
- ensures consistent truth
- supports deterministic ref updates

1.31 Invariant 8 — Shadows Cannot Mutate the Universe

> **SWS MAY NOT directly modify the DAG. Only GITD through commit may append nodes to the DAG.**

Why:

Ensures shadow isolation and avoids race conditions / paradoxes.

Shadows = potentiality.

DAG = actuality.

1.32 Invariant 9 — Deterministic Collapse (Commit)

> **Given identical SWS and identical DAG frontier, commit *MUST* always produce identical nodes.**

This invariant turns commit into a pure, deterministic operator:

```
collapse(sws, graph_state) $\rightarrow$ node
```

Why:

- no nondeterministic history
- no divergence across machines
- reproducible builds
- deterministic rewriting

1.33 Invariant 10 — Rewrites Create, Never Mutate

> **Inversion rewrites *MUST* produce new nodes rather than alter existing ones.**

Rebases, merges, amends, cherry-picks:

- do not edit the past
- generate inversion-rewrite nodes
- map old→new via rewrite tables

Why:

This preserves immutability and allows algebraic reasoning.

1.34 Invariant 11 — Temporal Consistency (Lamport Ordering)

> **Every node *MUST* include a logical timestamp that reflects causal ordering.**

Not wall clock—logical clock.

This ensures all nodes satisfy:

```
parent.ts < child.ts
```

Why:

- handles distributed sync
- avoids concurrency paradoxes
- supports replay

1.35 Invariant 12 — Deterministic Reconstruction

> **Given the DAG, the system *MUST* be able to reconstruct:

- filesystem projection
- Git view
- subgraph slices
- conflict states
- shadow overlays

> with perfect determinism.**

If reconstruction differs across machines, JIT collapses.

1.36 4. Enforcement

GITD **MUST** enforce all invariants:

- before accepting a commit
- before accepting remote nodes
- during WAL replay
- during local operation
- during SWS operations

Violation **MUST** cause:

- reject the operation
- quarantine the node
- sync refusal
- or emergency halt

JIT never allows invalid states.

1.37 5. Security

These invariants prevent:

- conflict-related nondeterminism
- malicious forks
- state corruption
- backdated mutations
- improper rewrites

A JIT universe is immune to:

- forced history rewrites
- concurrency bugs
- desync
- unauthorized state mutation

1.38 6. Status & Next Steps

This RFC:

- becomes mandatory for all future JIT implementations
- underpins RFC-0003 (Shadow Semantics)
- defines the physics of the JIT universe

1.39 CΩMPUTER • JITOS

1.40 JIT RFC-0003 — Shadow Working Set Semantics (SWS v1.0)

1.40.1 The Observer Model, Epistemic Isolation, and the Collapse Operator

---

1. Summary

This RFC defines the semantics of Shadow Working Sets (SWS) — the foundational abstraction for:

- agent-level processes
- parallel worldlines
- speculative computation
- observer-relative state
- distributed editing
- LLM/autonomous agent collaboration
- safe local mutation
- deterministic collapse into the global DAG

Shadow Working Sets represent local, subjective views of the universe — projections of the causal DAG — which remain isolated until resolved through a commit collapse event.

SWS is to JIT what “process” was to Unix — except properly grounded in physics, causality, and distributed truth.

1.40.2 2. Motivation

Modern computation is multi-agent, distributed, asynchronous, and subject to the limits of:

- latency
- CAP theorem
- causality
- local knowledge
- observer frames

Traditional operating systems expose a global mutable filesystem, creating illusions of simultaneity and absolute state that are neither true nor safe.

JIT acknowledges physical and epistemic reality:

No agent ever sees the whole universe. All views are subjective. Reality is append-only.

Shadow Working Sets formalize this truth as a computational primitive.

---

3. Definition

A Shadow Working Set (SWS) is a temporary, isolated, agent-relative projection of the causal DAG.

It includes:

```
ShadowSet {
    id: uuid
    base_ref: string
    base_node: NodeID
    overlay_nodes: list<NodeID>
    virtual_tree_index: map<Path, NodeID>
    metadata: map<string,string>
}
```

- `base_node` = the snapshot node the Shadow branches from
- `overlay_nodes` = ephemeral nodes representing edits
- `virtual_tree_index` = projected filesystem view for the agent

All fields MUST be maintained in LMDB or equivalent.

1.40.3 4. Formal Semantics

Below are the rules governing SWS behavior.

4.1 SWS Isolated Worldline

An SWS is an epistemically isolated worldline which MUST NOT affect the global DAG until committed.

- overlays exist only within the SWS
- overlays cannot mutate the past
- overlays cannot commit without collapsing
- SWS cannot introduce new parents into the DAG

4.2 SWS Causal Anchoring

Each SWS MUST anchor to a single base node.

This ensures:

- deterministic diffs
- deterministic rewrite potential
- clear causal ancestry
- unambiguous commit behavior

The base node MUST be a valid snapshot node.

4.3 Local Consistency

Within an SWS:

- all edits must form a locally consistent graph
- overlay nodes MUST obey RFC-0001 and RFC-0002 invariants
- ordering must be deterministic within the SWS

Overlay graph is an internal micro-DAG, not merged with the global DAG.

4.4 No Cross-SWS Interference

Two SWS MUST NOT observe each other's edits unless explicitly merged.

This enforces true agent isolation.

Agents operate in separate worldlines until collapse.

4.5 Subjective Filesystem Projection

SWS must maintain a virtual tree index for filesystem-like interactions.

This projection MUST be deterministic:

```
virtual_tree_index[path] = most recent overlay node OR
    inherited from base snapshot
```

No real files need exist; this is an in-memory projection.

4.6 Allowed Operations

Operations permitted inside an SWS:

- patch application
- diff computation
- linting

- refactoring
- build computations
- semantic agents (LLMs) modifying content
- structural rewrites
- preview merges
- pattern-matches for rewrites

These operations MUST NOT modify the global DAG.

1.40.4 5. Collapse Operator (Commit)

The commit is the most important concept:

Commit collapses a subjectively computed SWS into an objectively real DAG event.

The collapse operator:

```
collapse(sws, global_dag) $\rightarrow$ new_snapshot_node
```

Collapse MUST obey:

- determinism
- reproducibility
- inversion semantics (via Inversion Engine)
- causal ordering
- full validation of overlays
- merge conflict resolution
- topological soundness

Collapse also MUST:

- generate a new snapshot node
- discard the SWS afterward
- update the ref target
- log the event to WAL
- advance the universe

SWS death is required:

Once committed, a Shadow can no longer exist. Potential becomes reality. Superposition collapses into truth.

1.40.5 6. Merge & Conflict Semantics

If SWS collapse encounters divergence from `base_node`:

- the Inversion Engine MUST integrate changes
- conflicts MUST be represented as multi-stage entries
- merge MUST be deterministic
- if unresolvable, commit MUST fail

Conflicts do not mutate the global DAG. They exist only during collapse evaluation.

1.40.6 7. Parallel Shadows

Multiple SWS may exist atop the same base_node or different base nodes.

Rules:

- parallel SWS are independent
- commit order provides serialization
- later SWS commit must resolve against updated reality
- deterministic merge rules must apply

This captures:

- concurrency
- distributed editing
- eventual consistency through collapse

1.40.7 8. Deletion & Failure Semantics

SWS MAY be:

- discarded by agent
- timed out
- destroyed by system
- invalidated by upstream commits

Invalidation MUST occur if:

- `base_node` no longer matches HEAD (or ref target)
- collapse produces inconsistency
- SWS violates invariants

In all cases:

Discarding an SWS must not affect the global DAG.

1.40.8 9. Lifecycle

I. Creation

```
shadow.create(ref)
```

Creates new SWS anchored to ref's snapshot.

II. Mutation

Edits accumulate in overlay nodes.

III. Computation

Agents synthesize new states.

IV. Collapse

```
shadow.commit(id)
```

Converts overlays → new snapshot.

V. Death

SWS is destroyed. Truth is updated.

1.40.9 10. Security & Isolation

- SWS cannot tamper with DAG
- cannot impersonate global nodes
- cannot bypass collapse
- collapse validates signature/authorship
- each SWS owner is tracked in metadata

1.40.10 11. Why SWS Is Foundational

SWS solves:

- distributed agent concurrency
- local-only mutation
- parallel universes
- speculative worlds
- conflict-free workflows
- deterministic collapse
- reproducible builds
- multi-agent coordination
- physics-consistent computation

Without SWS, JIT would revert to Git-like semantics and lose the physics.

This RFC defines the heart of the inversion computation model.

1.40.11 12. Status & Next Steps

This RFC precedes:

- RFC-0004 (Materialized Head)

- RFC-0005 (Inversion Engine Semantics)
- RFC-0006 (WAL Format & Replay)
- RFC-0007 (JIT RPC API)
- RFC-0008 (Message Plane Integration)

1.41 CΩMPUTER • JITOS

1.42 JIT RFC-0004

1.42.1 Materialized Head: The Human Projection Layer (MHP v1.0)

1.42.2 1. Summary

This RFC defines Materialized Head (MH) — JIT's representation of the global graph as a human-friendly filesystem. It is the illusion layer that provides:

- familiar file I/O
- editor/IDE integration
- compatibility with existing tools
- Git CLI support
- deterministic file reconstruction

Materialized Head is not the true state of the universe — it is a shadow projection, analogous to Plato's Cave.

The DAG is reality. Materialized Head is perception.

1.42.3 2. Motivation

Humans require:

- files
- directories
- text editors
- terminal-based workflows
- Git operations

But JIT uses:

- nodes
- causal graphs
- rewrites
- SWS overlays
- collapse events

There must be a bridge — a projection system that maps a high-dimensional causal geometry into a low-dimensional tree of files.

Materialized Head provides this layer without sacrificing:

- determinism
- immutability
- causal truth
- JIT invariants

This is how humans work inside a post-file universe.

1.42.4 3. Definition

Materialized Head is a filesystem view backed by:

1. Tree Index (LMDB)
2. Working Directory Mirror
3. Metadata Cache
4. Filesystem Watcher
5. Projection Engine

It is tied to a ref (typically HEAD), representing one particular slice of objective reality.

1.42.5 4. Core Principles

4.1 MH is Epistemic

Materialized Head is not truth. It is a shadow of truth.

Only the DAG carries objective meaning. MH is a convenient fiction.

4.2 MH Must Be Deterministic

Given:

- a snapshot node
- the node store
- RFC-0001 encoding

Materialized Head **MUST** reconstruct the same file tree everywhere.

4.3 MH is Incremental

Reconstruction **MUST** be incremental, not full:

- It **MUST** only materialize files that change.
- Unchanged paths **MUST** remain untouched.
- Directory structure **MUST** be updated lazily.

4.4 MH Is Not Write-Through

Writes to the working directory NEVER mutate reality.

They:

- update MH state
- trigger overlay creation
- enter Shadow Working Sets (SWS)

MH is the input to SWS, not the substrate.

4.5 MH Must Be Fast

Key performance requirements:

- git status < 20ms
- file edits detected < 10ms
- diff computation < 50ms

Achieved by:

- tree-index caching
- file metadata hashing
- fs watchers
- minimal reconstruction

1.42.6 5. Tree Index (The Real Data Structure)

The Tree Index is the authoritative representation of the human-facing file system.

It maps:

```
path $\rightarrow$ FileProjection
```

Where:

```
FileProjection {
node: NodeID
size: int
hash: blake3-256
mtime: timestamp
flags: bitset (conflict, executable, symlink, etc.)
}
```

Stored in LMDB for durability.

1.42.7 6. Working Directory Mirror

MH maintains a mirror of the file tree on disk under actual OS files.

Rules:

1. Only modified files are rewritten.
2. Unchanged files are untouched between checkouts.
3. Conflicted files include conflict markers.
4. Deleted files are removed deterministically.
5. Symlinks are restored exactly.

This mirror is the cave wall.

1.42.8 7. Conflict Semantics

During merges/rebases/inversions:

Materialized Head **MUST** support multi-stage entries:

- `BASE`
- `OURS`
- `THEIRS`
- `RESOLVED`

The index tracks all three, but the filesystem presents textual conflict markers for human resolution.

Example markers (must match Git exactly):

«««< OURS ...content... ======= ...content... »»»> THEIRS

After git add, a new file-chunk node is created and the conflict entry becomes `RESOLVED`.

1.42.9 8. Relationship to SWS

> [!WARNING] *This is critical:*

8.1 MH serves humans; SWS serves agents.

MH performs:

- file reads
- file writes
- Git CLI operations
- editor interaction

SWS performs:

- ephemeral computation
- speculative operations
- agent-based edits
- automated transformations
- previews

8.2 MH must remain consistent with DAG + SWS

When:

- commit occurs
- ref updates
- remote sync
- SWS collapse

MH must merge its view with the new universe state.

This may cause:

- local changes being rebased
- conflicts
- forced refresh
- incremental reconstruction

Consistency **MUST** be maintained at all times.

1.42.10 9. Filesystem Watcher

MH **MUST** include a watcher that detects:

- file edits
- renames
- deletes

- directory creation

Watcher events become SWS overlays.

MH does NOT “write” to the DAG directly — only SWS can trigger commit.

---

10. Checkout Semantics

During checkout:

1. Retrieve snapshot node
2. Reconstruct tree-index
3. Apply incremental updates to working directory
4. Discard any out-of-date SWS associated with the ref
5. Update MH metadata
6. Update active shadow context

Performance requirement:

- checkout < 150ms on typical repos

1.42.11 11. Revert & Reset Behavior

Revert:

- create inversion node
- update MH via tree-index
- rewrite filesystem for changed paths

Reset:

- detach MH from previous overlays
- reconstruct tree-index
- remove untracked files if requested

MH must remain deterministic.

1.42.12 12. Sync Semantics (Remote)

When pulling from remote:

- fetch new snapshot nodes
- merge with MH tree-index
- handle conflicts
- present conflict markers
- ensure deterministic rehydration (RFC-0003)

MH must remain consistent with the updated universe.

1.42.13 13. Failure & Crash Semantics

Upon crash or abrupt termination:

- MH reconstruction **MUST** be possible from:
- the DAG
- the WAL
- the tree-index

MH is fully recoverable as long as the substrate exists.

Filesystem inconsistencies are ALWAYS repaired during initialization.

1.42.14 14. Security

MH must obey:

- no SWS can bypass commit
- no direct DAG mutation
- no arbitrary node insertion
- permissions match underlying OS
- signature validations enforced at commit

Filesystem writes are sandboxed to working directory.

1.42.15 15. Why MH Is Foundational

Materialized Head provides:

- human compatibility
- backwards Git compatibility
- stable user workflows
- deterministic projections
- performance
- reproducibility
- separation of observer frames
- the illusion needed for humans to operate in a post-file world

MH is the human lens through which the geometric universe of JIT becomes visible.

It is the shadow on the wall.

1.43 CΩMPUTER • JITOS

1.44 JIT RFC-0005

1.44.1 The Inversion Engine Semantics (IES v1.0)

1.44.2 1. Summary

This RFC defines the Inversion Engine, the subsystem responsible for:

- collapsing a Shadow Working Set (SWS) into a deterministic snapshot node
- integrating subjective edits into the objective causal DAG
- resolving conflicts
- performing merges and rebases

- generating inversion-rewrite nodes
- preserving immutability and causal order
- producing the canonical next event in the universe

This subsystem is the computational analog of:

- quantum collapse
- observer synchronization
- relativistic frame merging
- provenance validation
- structural rewriting

Without the Inversion Engine, JIT cannot maintain truth.

1.44.3 2. Motivation

Shadow Working Sets represent subjective, isolated, locally consistent worldlines that agents operate within.

But the universe (the DAG) is singular. Only one version of history exists. Only one next event becomes real.

The Inversion Engine provides the laws and machinery that:

- merge subjective changes into objective truth
- reconcile divergent views
- enforce determinism
- preserve immutability
- maintain causal invariants

It is the OS kernel's consistency layer, the physics engine's update loop, the version-control system's merge engine, and the distributed system's truth arbitrator.

1.44.4 3. Definitions

Inversion

A transformation where: - the universe remains unchanged - but a new node representing a rewrite, merge, or collapse is appended - creating a new "view" of the past without modifying it

This is how rebases, merges, cherry-picks, and amends occur without violating immutability.

Collapse

Application of the collapse operator:

```
collapse(sws, graph) $\rightarrow$ new_snapshot_node
```

Rewrite Node

A node of type inversion-rewrite that expresses:

- `old-state -> new-state` mapping
- causal relationships
- conflict resolutions
- structural transformations

1.44.5 4. Engine Responsibilities

The Inversion Engine MUST:

1. Anchor the SWS to its base node
2. Compare overlay graph against substrate
3. Detect conflicts
4. Derive minimal rewrite set
5. Validate all overlay nodes
6. Resolve divergences deterministically
7. Generate a new snapshot node
8. Emit an inversion-rewrite node if needed
9. Guarantee causal invariants

10. Finalize the collapse

Failure of any step MUST abort commit.

1.44.6 5. Collapse Algorithm (Formal)

Given:

- `base_node`
- `sws.overlay_nodes`
- `global_dag`

The engine MUST:

function collapse(sws, dag):

```
1. verify $$base_node \ensuremath{\in} dag$$
2. compute $$latest reality = dag.head(ref)$$
3. if ‘base_node != latest reality‘:
        ‘perform merge(base_node, latest reality)‘
4. apply overlays
5. detect conflicts
6. resolve conflicts deterministically
7. compute final tree projection
8. generate file-chunk nodes
9. create new snapshot
```

10. create inversion-rewrite node if merge occurred
11. update DAG: append events
12. destroy SWS `return new_snapshot_node`

All steps MUST be deterministic.

---

6. Merge Semantics

If the SWS’s base_node does NOT match the current ref head:

```
merge(base, head) $\rightarrow$ merged_structure
```

Merge MUST:

- preserve both lineages
- produce deterministic conflict sets
- apply Git-compatible merge strategy
- encode conflict data into rewrite node

After merging:

- MH updates for human users
- SWS virtual index updates for agents
- collapse continues

No merge ever mutates past events. Merge produces new structures, not edits.

1.44.7 7. Conflict Semantics

Three kinds:

1. Structural conflicts

- file added in one world, deleted in another

2. Edit conflicts

- overlapping modifications

3. Semantic conflicts

- unsafe state transitions (runtime-defined)

Conflicts MUST be:

- detected deterministically
- resolved using canonical resolution order
- represented as multi-stage entries in MH
- returned to SWS for retry OR resolved via overlay

No conflict may EVER produce nondeterministic output.

---

8. Rewrite Nodes

If collapse involves a history divergence (merge/rebase/etc), the engine MUST produce a node:

```
InversionRewrite {
    parents: [old_nodes... new_nodes...]
    mapping: old $\rightarrow$ new
    merge_type: enum
    conflicts: optional data
    metadata: signatures, timestamps
}
```

This node:

- documents the transformation
- preserves immutability
- enables provenance
- maintains chronological truth

This is the mathematical identity of "rebase without rewriting history."

1.44.8 9. Deterministic Rewrite Rules

The engine MUST guarantee:

- same SWS + same DAG → same output snapshot
- identical merges on different machines produce identical rewrite nodes
- collapse is a pure function
- no hidden state, randomness, or ordering bugs

This is vital for distributed correctness and replay.

1.44.9 10. Collapse Outcome

Collapse MUST produce exactly one:

- snapshot node (actual new state)
- optional inversion-rewrite node (if a merge/rewrite occurred)

Both MUST be appended to the DAG atomically.

Afterward:

- the SWS is destroyed
- MH updates
- refs update
- full invariants guaranteed

1.44.10 11. Legal Rewrites

Allowed:

- `merge`
- `rebase`
- `cherry-pick`
- `commit amend`
- `revert`
- `structural transforms` (future)
- `semantic rewrites` (future)

Forbidden:

- altering existing nodes
- deleting nodes
- modifying previous causal links

> [!DANGER] History never mutates. ***Ever.*** *This is the core JIT philosophy.*

1.44.11 12. Inversion Engine Properties

The Inversion Engine MUST be:

Deterministic

Pure function under fixed inputs.

Idempotent

Repeated attempts produce identical graphs.

Isolated

Engine state cannot bleed between collapses.

Atomic

Collapse either fully commits or fully aborts.

Serializable

Concurrent collapses resolve via causal ordering.

Auditable

Every rewrite is represented in the DAG.

1.44.12 13. Security Considerations

The engine MUST:

- validate shadow provenance
- enforce signature rules
- reject malformed overlays
- reject inconsistent transformations
- reject illegal rewrites
- prevent history mutation attempts

Rewrite nodes MUST be cryptographically signed.

1.44.13 14. Why This Matters

The Inversion Engine is the literal physics of JIT.

It enforces:

- the arrow of time
- observer collapse
- causal consistency
- determinism
- worldline merging
- conflict locality
- immutability
- reproducibility
- truth maintenance

This subsystem is the soul of the Inversion Kernel.

Without it, the universe is chaos. With it, the universe is computable.

1.44.14 15. Status & Next Steps

Next RFC:

RFC-0006 — WAL Format & Replay Semantics (The Temporal Backbone)

1.45 CΩMPUTER • JITOS

1.46 JIT RFC-0006

1.46.1 Write-Ahead Log (WAL) Format & Replay Semantics (WFR v1.0)

Status: Draft **Author: James Ross** Contributors: JIT Community **Requires: • RFC-0001 Node Identity • RFC-0002 DAG Invariants • RFC-0003 Shadow Working Sets • RFC-0004 Materialized Head • RFC-0005 Inversion Engine Start Date:**

2025-11-28 Target Spec: JITOS v0.x, TECHSPEC v0.3+ License: Open Source (TBD)**

?

1. Summary

This RFC defines: • the canonical WAL binary format • the rules for appending events • the rules for replaying events • the rules for crash recovery • the ordering constraints on time • the guarantees necessary for determinism

The WAL is the temporal substrate of JIT, encoding the arrow of time and ensuring the universe can be reconstructed exactly, on any machine, at any point in the future.

This is the system's memory of becoming, as distinct from the DAG's memory of being.

?

2. Motivation

Why WAL? • JIT must survive crashes • JIT must be fully deterministic • JIT must replay nodes and indexes exactly • JIT must restore Materialized Head • JIT must restore SWS states • JIT must recover after partial operations • JIT must preserve ordering

In a causal universe:

WAL is time; DAG is space-time geometry; MH is perception.

WAL is the time-axis of the JIT kernel.

?

3. Requirements

The WAL MUST provide:

Append-Only Safety

Entries are appended atomically.

Durability

fsync required before confirming operations.

Total Ordering

Entries form a single linear timeline.

Determinism

Replay must regenerate identical state.

Crash Tolerance

Corrupted partial entries must be skipped or corrected via checksum.

Atomic Replay

Index state must be rebuilt atomically.

Isolation

No partially applied operations.

Cross-Platform Stability

Endianness, alignment, etc. must be canonical.

?

4. WAL Record Structure

Every WAL entry is binary:

WALRecord { magic: u32 = 0x4A495420 (# "JIT") version: u16 op_type: u16 logical_ts: u64 payload_len: u64 payload: bytes[payload_len] checksum: blake3-256 }

Notes: • magic identifies valid entries • logical_ts MUST be Lamport timestamp • payload MUST be canonical CBOR • checksum MUST be of everything except itself • if checksum fails → skip entry and continue

?

5. WAL Operation Types

5.1 WAL_OP_CREATE_NODE

Payload: canonical node encoding Effect: append node to DAG

5.2 WAL_OP_UPDATE_INDEX

Payload: index delta Effect: update LMDB indexes

5.3 WAL_OP_UPDATE_TREE_INDEX

Payload: per-file tree index updates Effect: update MH tree indexing

5.4 WAL_OP_SWS_CREATE / DESTROY

Payload: SWS metadata Effect: restore working sets

5.5 WAL_OP_SET_REF

Payload: {refname, new_target} Effect: update reference pointers

5.6 WAL_OP_SYNC_EVENT

Payload: sync metadata Effect: track remote sync states

Future expansions: • WAL_OP_REWRITE_METADATA • WAL_OP_COMI

?

6. WAL File Layout

WAL MUST be stored at:

.gitd/wal/log.wal

Rules: • MUST NOT be deleted except by compaction • MUST NOT be truncated without checkpoint • MUST be read sequentially • MUST be sync'd after each entry

?

7. Checkpointing

To avoid infinite WAL growth: • system periodically writes a checkpoint, a snapshot of: • index state • tree-index state • head references • SWS metadata

Stored at:

.gitd/wal/checkpoint.cbor

During startup: 1. Read checkpoint 2. Apply WAL entries newer than checkpoint 3. Reconstruct everything deterministically

?

8. WAL Replay Procedure

Pseudo-code:

function replay_wal(): state = load_checkpoint() for entry in wal: if checksum_invalid(entry): continue apply(entry, state) return state

This MUST result in: • identical DAG • identical indexes • identical MH • identical SWS metadata • identical ref pointers

Replay is truth. If replay changes behavior → invariants broken.

?

9. Atomicity Guarantees

Each WAL entry MUST commit atomically: • write entry • fsync • return success

If crash occurs mid-entry: • replay MUST detect partial data • partial entries MUST be ignored • prior entries MUST be intact

?

10. Interaction with Inversion Engine

During collapse: • Inversion Engine writes multiple WAL entries • MUST ensure atomic consideration • MUST update logical timestamp • MUST reflect rewrite nodes and snapshot nodes • MUST append ref update LAST

WAL ensures that a collapse event is either: • fully applied • not applied at all

No in-between states allowed.

?

11. Interaction with SWS

SWS creation and destruction MUST log: • SWS identity • base node • metadata

Upon crash: • all SWS MUST be restored • or invalid SWS MUST be marked and discarded

SWS replay is essential to restoring agent contexts for in-progress work.

?

12. WAL & Distributed Sync

During sync: • remote nodes appended after validation • remote ref updates logged • WAL ensures sync is replayable • ensures cross-machine consistency

The WAL is the causal ordering mechanism across distributed systems.

?

13. Security Considerations • strict verification of checksums • protect against corrupted WAL entries • enforce replay signatures • SWS replay must validate agent IDs • DST (Distributed Sync Trust) model TBD

?

14. Why WAL Is Foundational

Without WAL: • DAG cannot be reconstructed • MH cannot be recovered • SWS cannot be restored • distributed correctness collapses • crash safety disappears • determinism fails

WAL is literally the time-axis of the JIT universe.

The DAG is the geometry. The WAL is the temporal ordering of becoming.

?

15. Status & Next Steps

Next RFC:

RFC-0007 — JIT RPC API (The Syscall Layer of the Post-File OS)

1.47 CΩMPUTER • JITOS

1.48 JIT RFC-0007

1.48.1 JIT RPC API (JRAPI v1.0)

The Syscall Layer of the Inversion Kernel

1.48.2 1. Summary

This RFC defines the canonical RPC API for interacting with the JIT Kernel (GITD).

This API replaces:

- POSIX syscalls
- Git plumbing commands
- ad hoc filesystem interactions
- shell-based workflows

. . . with a unified, structured, typed, agent-friendly interface.

The RPC API provides:

- Shadow Working Set lifecycle
- DAG operations
- collapse/commit interface
- Materialized Head sync
- Git protocol abstraction
- sync and replication primitives
- introspection and query semantics

This is the ABI of the causal universe.

1.48.3 2. Motivation

The Inversion Kernel is not a file-based OS.

It cannot expose:

- `open()`
- `read()`
- `write()`
- `fork()`
- `exec()`

These make no sense in a causal DAG.

Instead, JIT must expose operations that manipulate:

- nodes
- shadows
- worldlines
- projections
- rewrites
- collapse events
- provenance
- time

This necessitates a new syscall layer: JIT RPC.

1.48.4 3. Transport

RPC MUST support:

- Unix domain sockets (local)
- TCP+TLS (remote)
- QUIC (optional future transport)

Serialization MUST be:

- canonical CBOR (for determinism)

4. RPC Structure

Every RPC request:

```
{
  "op": "string",
  "payload": CBOR-encoded struct,
  "ts": client_logical_clock
}
```

Every RPC response:

```
{
  "status": "OK" | "ERR",
  "result": CBOR-encoded struct,
  "error": optional string
}
```

Logical timestamps MUST propagate into SWS and WAL ordering.

1.48.5 5. RPC Categories

We define RPCs in six categories:

1. Shadow Working Set (SWS) API
2. Collapse / Commit API
3. DAG Access API
4. Projection (MH) API
5. Git Protocol Facade API
6. Sync & Replication API
7. Introspection & Query API

Let's detail each.

1.48.6 6. Shadow Working Set API

These are the "process control" syscalls of the JIT OS.

6.1 shadow.create

op: “shadow.create” payload: { ref: string } result: { id: uuid, base_node: NodeID }

Creates a new Shadow Working Set anchored to ref.

6.2 shadow.apply_patch

op: “shadow.apply_patch” payload: { id: uuid, patch: PatchData } result: { updated: bool }

Applies an edit within the SWS. Diff → chunk → overlay nodes.

6.3 shadow.diff

op: “shadow.diff” payload: { id: uuid, ref: string } result: { diff: DiffData }

Computes difference between SWS and another snapshot or ref.

6.4 shadow.status

op: “shadow.status” payload: { id: uuid } result: { overlays: list, conflicts: list }

Local SWS state reporting.

6.5 shadow.discard

op: “shadow.discard” payload: { id: uuid } result: {}

Destroys a shadow. Zero side effects.

1.48.7 7. Collapse / Commit API

Collapse is how shadows become real.

7.1 collapse.commit

op: “collapse.commit” payload: { id: uuid } result: { snapshot: NodeID, rewrite: optional NodeID, conflicts: list, updated_ref: string }

Triggers collapse. If conflicts cannot be deterministically resolved, commit MUST fail.

7.2 collapse.validate

op: “collapse.validate” payload: { id: uuid } result: { ok: bool, errors: list }

Checks whether a commit would succeed.

1.49 8. DAG Access API

Direct substrate queries (read-only).

8.1 dag.get_node

op: “dag.get_node” payload: { id: NodeID } result: NodeData

8.2 dag.get_parents

op: “dag.get_parents” payload: { id: NodeID } result: { parents: list }

8.3 dag.lineage

op: “dag.lineage” payload: { id: NodeID, depth: int } result: { ancestors: list }

8.4 dag.subgraph

```
op: "dag.subgraph"
payload: { root: NodeID }
result: { nodes: list<Node>, edges: list<Edge> }
```

##9. Projection (Materialized Head) API

MH is the human projection of truth.

9.1 mh.checkout

op: “mh.checkout” payload: { ref: string } result: { root: NodeID, files: list }

Performs incremental reconstruction.

9.2 mh.status

op: “mh.status” payload: {} result: { changes: list }

Simplified status for editors/IDE integrations.

9.3 mh.read_file

op: “mh.read_file” payload: { path: string } result: { content: bytes }

MH → SWS projection.

1.49.1 10. Git Protocol Facade API

For backward compatibility.

10.1 git.upload_pack

op: “git.upload_pack” payload: { wants: list, haves: list } result: GitPackData

---

10.2 git.receive_pack

op: “git.receive_pack” payload: GitObjectSet result: { updated: bool, rewrites: list }

JIT LOSSES NO FIDELITY HERE.

1.49.2 11. Sync & Replication API

Distributed computing, remote sync.

11.1 sync.pull

op: “sync.pull” payload: { remote: string, ref: string } result: { nodes: list, updates: list }

11.2 sync.push

op: “sync.push” payload: { remote: string, ref: string } result: { accepted: bool }

11.3 sync.missing_nodes

op: “sync.missing_nodes” payload: { remote_frontier: list } result: { missing: list }

1.49.3 12. Introspection & Query API

For analytics, tools, LLMs, explorers, etc.

12.1 jit.info

op: “jit.info” payload: {} result: { dag_size: int, node_count: int, head: NodeID, sws_active: int, mh_state: MHInfo }

12.2 jit.search

op: “jit.search” payload: { query: QueryExpr } result: { matches: list }

12.3 jit.graphviz

```
op: "jit.graphviz"
payload: { root: NodeID }
result: { dotfile: string }
```

Enables visualization tooling.

1.49.4 13. Error Model

All RPCs MUST return:

- status: ERR
- standardized error codes
- optional human-readable message

> [!critical] Errors MUST NEVER corrupt the DAG or MH.

1.49.5 14. Security

RPC must enforce:

- per-SWS isolation
- signature validation
- ref protections
- access control (future)
- secure transport

1.49.6 15. Why This API Matters

This is the syscall interface of the new OS.

It:

- abstracts away direct file I/O
- provides a stable ABI for agents
- lets external systems integrate with JIT
- supports introspection & analysis
- enables distributed correctness
- replaces POSIX, Git plumbing, and half of Unix's syscall table
- makes JITOS a real, buildable, extensible kernel

This is where the system becomes usable.

1.49.7 16. Status & Next Steps

Next RFC:

RFC-0008 — Message Plane Integration (Distributed coordination & SWS orchestration)

1.50 CΩMPUTER • JITOS

1.51 JIT RFC-0008

1.51.1 Message Plane Integration (MPI v1.0)

Distributed Coordination for Shadow-Based Causal Computing

1.51.2 1. Summary

This RFC defines how JIT integrates with a Message Plane — a distributed, topic-based communication layer used for:

- Shadow Working Set orchestration
- multi-agent collaboration
- patch flow
- remote execution
- distributed builds
- workflow pipelines
- collaborative editing
- agent swarming
- distributed computation

The Message Plane unifies event sourcing, actor systems, and controller loops, but grounded in causal DAG physics.

JIT does not rely on the MP for correctness or determinism — but uses it for coordination, dispatch, and parallelism.

1.51.3 2. Motivation

Why do we need a Message Plane?

Because the modern universe of computation is:

- multi-agent
- distributed
- asynchronous

- latency-bound
- partially observable
- concurrency-rich

And Shadow Working Sets are isolated ephemeral worlds that must be:

- created
- mutated
- resolved
- collapsed
- synchronized
- coordinated

The Message Plane enables:

- remote agent editing
- shared tasks
- distributed merges
- CI pipelines
- “agent swarms”
- LLM-based code modification
- structured patch flow

It is the coordination fabric, not the truth fabric.

The DAG is truth. The MP is conversation.

---

3. Architectural Role

The Message Plane is: - stateless (mostly) - ephemeral - pub/sub based - non-authoritative - advisory

GITD retains all authority. Message Plane cannot mutate the universe.

Instead, MP delivers: - shadow commands - patch proposals - commit requests - merge directives - execution jobs - remote control messages

---

4. Message Plane Concepts

Topic

A logical channel for message exchange.

Subscriber

Agent or local service listening on a topic.

Publisher

Agent or subsystem sending data into a topic.

Payload

CBOR-encoded messages.

Transport

Message Plane MAY use: - NATS - Kafka - Redis Streams - MQTT - Custom JIT-native layer - etc.

MP is agnostic; RPC + invariants keep truth consistent.

---

5. SWS Topics (Core Model)

Each Shadow Working Set has its own topic namespace:

jit.sws..patch jit.sws..analyze jit.sws..commit jit.sws..discard jit.sws..status

These enable agent swarming: - LLMs editing the same shadow in coordinated fashion - CI tools analyzing patches - formatters, linters, compilers updating overlays - “bots” pushing semantic changes

Invariant:

MP interactions MUST NOT directly write to DAG.

All changes must go through:

shadow.apply_patch shadow.commit

via RPC.

---

6. Global Kernel Topics

GITD itself uses global topics:

jit.kernel.events jit.kernel.sync jit.kernel.ref_updates jit.kernel.sws_lifecycle

These act as kernel notifications:

- SWS created/destroyed
- commit collapse events
- ref updates
- distributed sync start/finish

These are for GUIs, dashboards, logs, orchestrators.

---

7. Message Payloads

MP messages MUST be CBOR-encoded and MUST include:

{ "sender": agent_id, "ts": logical_clock, "payload": {…} }

Why?

- deterministic ordering
- replay
- causal tracing
- provenance

Agents are first-class citizens.

---

8. Ordering Guarantees

The MP provides:

Per-topic FIFO ordering

- In each topic, messages preserve ordering
- No guarantee across different topics

Delivery Semantics

JIT requires at-least-once delivery.

Why?

- collapse/commit is idempotent
- patch operations are idempotent under RFC-0003
- ordering guarantees come from causal invariants, not MP

Idempotence

All MP messages MUST be idempotent at RPC layer.

If the same message arrives twice → same effect.

1.51.4 9. SWS Coordination Patterns

9.1 Multi-Agent Editing

Example flow:

1. Agent A publishes patch
2. Agent B publishes patch
3. SWS receives both
4. RPC layer applies patches in deterministic order
5. Conflicts resolved locally
6. Optional collapse

9.2 Agent Swarms

Topic:

```
jit.sws.<id>.analyze
```

Multiple agents run:

- static analysis
- semantic analysis
- build/test
- refactoring
- documentation generation
- risk scoring

All contribute overlays.

9.3 Human + Agent Co-Editing

Human edits → FS watcher → SWS overlay Agent edits → MP patch → SWS overlay

Equal footing.

1.51.5 10. Distributed Sync

Message Plane carries:

- remote announcements
- frontier messages
- availability signals
- ephemeral state

But all truth is validated and written through RPC → WAL → DAG.

MP coordinates; GITD commits.

1.51.6 11. MP and Conflict Convergence

MP can generate race conditions — good.

GITD must handle:

- multiple patches
- conflicting edits
- concurrent SWS updates
- distributed overlay batching
- delayed collapse

Through deterministic resolution rules in:

- RFC-0003 (SWS)
- RFC-0005 (Inversion Engine)

MP stimulates concurrency. The kernel arbitrates.

This is how emergent behavior appears.

1.51.7 12. Fault Tolerance

MP is advisory, so failure is:

- non-fatal
- non-corrupting
- recoverable

If MP crashes:

- SWS and DAG remain intact
- WAL ensures time/log consistency
- GITD performs eventual convergence

MP MUST be restartable without loss of correctness.

1.51.8 13. Security

MP MUST support:

- agent identity
- signatures (optional)
- topic-level ACLs
- replay detection
- rate limiting
- sandboxing

SWS IDs MUST NOT be guessable.

MP cannot impersonate GITD.

1.51.9 14. Why MP Is Crucial

Because JIT is:

- multi-agent
- distributed
- post-file
- observer-relative
- shadow-native

MP enables:

- orchestration
- automation
- parallel editing
- code swarms
- semantic pipelines
- build observers
- remote agent fleets
- consensus-driven transforms

MP is the nervous system to the DAG's spacetime and the Inversion Engine's physics.

Without the MP:

- no agent cooperation
- no swarming
- no distributed verification
- no real-time collaboration
- no event-driven automation

With it?

JIT becomes a living computational universe.

1.51.10 15. Status & Next Steps

Next RFC:

RFC-0009 — Storage Tiering & Rehydration (The cold/hot/warm substrate dynamics)

1.52 CΩMPUTER • JITOS

1.53 JIT RFC-0009

1.53.1 Storage Tiering & Rehydration (STR v1.0)

Thermodynamics of a Post-File Causal Universe

1.54 1. Summary

This RFC defines the multi-tiered storage model of the JIT substrate and the deterministic rehydration semantics that allow nodes to move between:

- Hot tier (fast local SSD)
- Warm tier (compressed local storage)
- Cold tier (remote object store)

Tiering allows JIT to:

- retain infinite history
- keep the graph immutable
- avoid garbage collection
- support massive datasets and deep provenance
- operate efficiently on commodity hardware
- work equally well on laptops, servers, and clusters

At the same time, rehydration ensures:

- nodes appear “present” when needed
- deterministic reconstruction
- transparent access
- no correctness loss
- no identity mutation
- flawless causal replay

This RFC describes JIT's storage physics.

1.54.1 2. Motivation

JIT is an append-only causal universe. It grows forever. Nodes accumulate indefinitely.

But:

- SSD is finite
- RAM is finite
- compute is bounded

Therefore, JIT must support tiered storage analogous to:

- LSM-leveling
- CPU cache hierarchies
- memory tiers
- cold/warm/hot object lifecycles
- database storage pyramids

Except:

JIT cannot compact.

JIT cannot rewrite.

JIT cannot delete.

JIT cannot mutate.

So we must move, not mutate, nodes.

1.54.2 3. Tier Definitions

3.1 Hot Tier

Location:

```
.gitd/nodes/hot/
```

Characteristics:

- SSD-backed
- uncompressed node files
- minimal latency
- primary working set
- required for:
- active SWS
- current HEAD
- recent history
- local operations

This is “live memory.”

1.54.3 3.2 Warm Tier

Location:

```
.gitd/nodes/warm/
```

Characteristics:

- compressed chunks (CDC or zstd)
- indexed via LMDB
- locally stored but slower than SSD
- used for:
- mid-range history
- older builds
- previous snapshots
- CI data

Intermediate entropy state.

3.3 Cold Tier

Location:

```
remote://<object-store>/jit/<repo>/<node-id>
```

Characteristics:

- object store (S3, GCS, Backblaze, R2, MinIO, ZFS remote)
- content-addressed
- compressed
- deduplicated
- extremely cheap

Cold tier holds:

- deep history
- long-term provenance
- giant payloads
- scientific data
- old builds
- model checkpoints

This is the cryogenic freezer of truth.

1.54.4 4. Tier Promotion & Eviction Rules

Nodes naturally move:

Hot → Warm → Cold

Invariant:

Tier changes MUST NOT alter NodeID or content.

All rehydration MUST reconstruct the node exactly as originally created.

4.1 Eviction Triggers

- storage pressure
- LRU heuristics
- snapshot age
- admin command
- background tiering job
- automatic because of DAG age

Eviction is safe because:

- nodes are immutable
- refs don't change
- DAG structure is preserved

Only location changes.

1.54.5 5. Rehydration Rules

When a node is requested:

- MH
- SWS
- dag.get_node
- collapse
- diff
- projection
- sync

…it MUST be loaded from tiered storage.

Rehydration algorithm:

```
function load_node(id):
    if hot.contains(id): return hot.read(id)
    if warm.contains(id): decompress $\rightarrow$ promote to
     hot $\rightarrow$ return
    if cold.contains(id): fetch $\rightarrow$ decompress $\
   rightarrow$ promote to warm $\rightarrow$ return
    else: error("node not found")
```

Key points:

- cold fetch → warm
- warm decompress → hot
- hot remains until eviction

JIT MUST guarantee perfect reconstruction.

1.54.6 6. Indexing Requirements

The LMDB index MUST store:

{id → tier_location} {id → warm_chunk_offsets} {id → cold_storage_url/hash}

Indexes MUST remain:

- deterministic
- durable
- recoverable via WAL + checkpoint

1.54.7 7. WAL Interaction

Tier movements MUST NOT be recorded in WAL.

Why?

WAL models temporal truth, not storage logistics.

Storage is contingent. Truth is invariant.

WAL MUST NOT care if nodes are in:

- hot
- warm
- cold

Replay MUST produce identical DAG independent of tiers.

1.54.8 8. Distributed Sync

Cold tier is essential for distributed sync:

- nodes pushed to remote cold storage
- references updated
- peers rehydrate on demand
- no requirement to store full history locally

This enables:

- lazy clone
- partial repos
- serverless compute
- thin agents
- low-storage CI workers

1.54.9 9. Edge Cases & Guarantee

9.1 If cold storage is unreachable

Node fetch MUST fail gracefully. System MUST NOT corrupt DAG.

9.2 Partial local copies

Absolutely allowed. System MUST fetch missing nodes deterministically.

9.3 Compression incompatibility

Old cold nodes MUST be treated as opaque; decompressed to canonical encoding.

9.4 Duplicate storage

Allowed but not required. Local caches may remain.

1.54.10 10. Security

Cold tier MUST:

- verify BLAKE3 hashes
- validate canonical encoding
- enforce signature checks
- reject corrupted data

Tier transitions MUST NOT alter content.

1.54.11 11. Why Tiering Matters

Tiering allows:

- infinite history
- safe immutability
- distributed scalability
- scientific reproducibility
- artifact longevity
- low-cost storage
- high-performance compute

Without tiering: JIT becomes too expensive.

With tiering: *JIT becomes practically infinite.*

1.54.12 12. Status & Next Steps

Next RFC:

RFC-0010 — Ref Management & Branch Semantics (The Truth Pointers of the Universe)

1.55 CΩMPUTER • JITOS

1.56 JIT RFC-0010

1.56.1 Ref Management & Branch Semantics (RBS v1.0)

The Truth Pointers of the Inversion Kernel

1.56.2 1. Summary

This RFC defines:

- the ontology of references (“refs”)
- how branches behave
- how tags are represented
- how HEAD operates
- how reference mutation occurs
- how SWS and commits update refs
- how distributed sync applies ref updates
- the invariants refs MUST obey

Refs are the names humans give to points in the causal DAG. Branches are pointers to worldline frontiers. Tags are semantic anchors. HEAD is a lens, not a truth. This RFC defines how pointers to the universe behave.

1.56.3 2. Motivation

Without refs you have:

- an infinite DAG
- no landmarks
- no naming
- no identification
- no “current state”
- no way to express intent

Refs provide the symbolic layer that makes:

- collaboration
- navigation
- tooling
- UX
- agents
- CI
- possible.

But refs MUST NOT mutate truth. Refs must point into truth.

Thus:

Refs are illusions, DAG is reality.

This RFC defines how those illusions function.

1.56.4 3. Definitions

3.1 Reference (Ref)

A ref is:

```
Ref {
    name: string
    target: NodeID
    metadata: map<string,string>
}
```

Refs MUST be stored in LMDB and versioned through WAL.

Refs MUST be human-readable, but truth is the NodeID.

3.2 Branch

A branch is a ref representing a living timeline.

Typical examples:

- `main`
- `dev`
- `feature/x`
- `experiment/y`

Requirements:

- branch ref MUST always point to a snapshot node
- commits update branch target
- branch semantics MUST be deterministic

1.56.5 3.3 Tag

A **tag** is a ref that MUST NOT move.

Used for:

- releases
- checkpoints
- semantic anchors
- temporal anchors
- analysis states

Tags MUST reference:

- snapshot nodes
- rewrite nodes
- provenance nodes

Tags are immutable pointers to events in the universe.

1.56.6 3.4 HEAD

HEAD is:

- a human-facing pointer
- ephemeral
- local-only
- specific to Materialized Head
- MUST NOT appear in the DAG

HEAD indicates current projection, not objective truth.

1.56.7 4. Reference Invariants

These MUST hold at all times:

Invariant 1 — Ref target MUST exist in DAG

No ref may point to non-existent nodes.

Invariant 2 — Branch updates MUST be atomic

Ref update MUST be written LAST during commit.

Invariant 3 — Branch ref changes MUST be WAL-logged

Ensures replay consistency.

Invariant 4 — Tag ref MUST NOT change

Immutable forever.

Invariant 5 — No ref may violate causal ordering

A branch MUST NOT point “backwards” in violation of DAG invariants.

Invariant 6 — HEAD is not truth

HEAD may be detached or point to MH state.

Invariant 7 — Fast-forward rules MUST be deterministic

Conflicting branch updates MUST involve inversion engine.

1.56.8 5. Operations

5.1 ref.get

```
op: "ref.get"
payload: { name }
result: { target: NodeID }
```

5.2 ref.set (Branch only)

```
op: "ref.set"
payload: { name, new_target }
result: { updated: bool }
```

Rules:

- MUST be atomic
- MUST appear after snapshot+rewrite nodes in WAL
- MUST update MH
- MUST invalidate outdated SWS

5.3 ref.create_branch

```
op: "ref.create_branch"
payload: { name, target }
```

Branch MUST begin at an existing node.

5.4 ref.create_tag

```
op: "ref.create_tag"
payload: { name, target }
```

Tag MUST be immutable.

5.5 ref.delete

Allowed only for branches.

Tags MUST NOT be deleted except by explicit override.

1.56.9 6. Branch Update Semantics

Commits MUST update branch target as:

```
branch -> new_snapshot_node
```

Merge semantics MUST be:

- deterministic
- mediated by Inversion Engine
- logged via WAL
- projection applied to MH

1.56.10 7. Rebase Semantics

Rebase MUST NOT mutate branch history.

Instead:

- inversion-rewrite node created
- branch updated to rewritten snapshot
- original lineage preserved

Branch “moves” but truth remains immutable.

1.56.11 8. Distributed Sync

Ref sync MUST obey:

- last-writer-wins
- mediated by inversion
- no destructive pushes
- rewrite required for divergence

Distributed push/pull MUST:

- verify node existence
- validate rewrite rules
- enforce invariant ordering

Remote branch updates MUST be replayed deterministically.

1.56.12 9. HEAD Semantics

HEAD is:

- local
- advisory
- non-authoritative
- ephemeral
- tied to Materialized Head

HEAD MUST NOT appear in DAG or WAL.

HEAD MAY:

- point to SWS
- point to snapshot
- be detached
- be symbolic

HEAD is simply a lens.

1.56.13 10. Security

Reference operations MUST:

- validate user permissions
- require signatures (future)
- reject invalid pointers
- disallow backward ref changes
- log all updates via WAL

1.56.14 11. Why Ref Semantics Matter

Refs are:

- the human interface
- the names of time
- the symbols for worldlines
- the navigational layer
- the external interface for distributed tooling
- the bridge between the cave wall and the causal universe

Refs don't define truth. Refs point to truth.

This RFC formalizes naming in a world without filenames.

1.56.15 12. Status & Next Steps

Next RFC to write:

RFC-0011 — Distributed Sync: Frontier Negotiation & Subgraph Transfer

---

1.57 CΩMPUTER • JITOS

JIT RFC-0011

Distributed Sync: Frontier Negotiation & Subgraph Transfer (DST v1.0)

Truth Reconciliation in a Causal Universe

Status: Draft **Author: James Ross Contributors: JIT Community Requires: • RFC-0001 Node Identity • RFC-0002 DAG Invariants • RFC-0003 SWS • RFC-0005 Inversion Engine • RFC-0006 WAL • RFC-0010 Ref Semantics** Start Date: 2025-11-28 **Target Spec: JITOS v0.x** License: TBD**

?

1. Summary

This RFC defines the Distributed Sync Protocol used by JIT to: • synchronize repositories • replicate causal DAGs • negotiate frontiers • exchange missing nodes • update refs • reconcile divergent histories

Unlike Git, JIT sync is: • causality-aware • branch-consistent • rewrite-safe • deterministic • immutable • subgraph-based

Sync becomes a geometric reconciliation between worldlines.

?

2. Motivation

Git sync is built for: • files • snapshots • patches • textual diffs • mutation

JIT is built for: • nodes • causal chains • rewrites • collapse events • worldlines

The protocols cannot be the same.

Distributed systems demand: • efficient frontier comparison • safe transfer of immutable nodes • conflict-safe ref updates • deterministic reconstruction • efficient rehydration • precise provenance integrity

This RFC defines the rules.

?

3. Core Concepts

3.1 Frontier

A frontier is a set of nodes representing the “tips” of the local universe.

Formally:

frontier = { f | f ∈ DAG and ∄ child such that f is a parent of child }

Comparable to Git’s “heads”, but purely causal, not Git’s commit model.

?

3.2 Delta Set

The delta is the set of nodes one peer has that the other does not.

delta(A → B) = nodes(A) - nodes(B)

This must be computed without sending full DAGs.

?

3.3 Subgraph Transfer

Nodes transfer in closure form:

subgraph(root) = { all reachable ancestors of root }

Transfers MUST send: • the snapshot node • any rewrite nodes • file-chunk nodes • provenance nodes • metadata

Never only partial information.

?

4. Sync Stages

Sync occurs in 3 stages:

?

4.1 Stage 1: Frontier Exchange

Peers exchange:

local_frontier = list refs = list capabilities = list

Comparison determines: • common ancestors • divergence points • delta sets • missing subgraphs

Frontier negotiation MUST be efficient (logarithmic).

?

4.2 Stage 2: Delta Computation

The delta MUST satisfy:

delta = minimal set of nodes that B needs to become causally consistent with A

The delta is a set of root nodes; B will request full subgraphs.

Peers MUST support: • prefix indexing • hash queries • chunked node lists • topological ordering queries

?

4.3 Stage 3: Subgraph Transfer

Transfer MUST be: • chunked • deduplicated • content-addressed • BLAKE3-verified • resumable

Each subgraph MUST be: • validated • appended to DAG • indexed • WAL-logged

No mutation allowed — all new truth accumulates.

?

5. Ref Synchronization

After DAG sync, references MUST be synchronized:

Rules:

5.1 Fast-Forward Ref Update

If remote ref target is a DAG descendant of local target:

local_ref = remote_ref

5.2 Divergent Ref Update

If both refs diverged: • perform inversion rewrite • produce rewrite node • update ref to merged snapshot

5.3 Tag Handling

Tags MUST NOT move. If remote tag conflicts: • local tag remains • remote tag stored under distinct namespace

?

6. DAG Validation

Every received node MUST be validated: 1. BLAKE3 hash 2. canonical encoding 3. parent existence 4. type validation 5. rewrite rules (if rewrite node) 6. provenance correctness

Invalid nodes MUST be rejected.

?

7. Cold Storage Integration

Nodes fetched from remote storage MUST: • be decompressed • be rehydrated • be promoted to warm or hot tier • be indexed • be checked against WAL

This ensures local replicas remain consistent.

?

8. Performance Requirements • Frontier exchange < 20ms • DAG delta computation < O(n) in frontier size • Subgraph transfer parallelized • Cold storage fetch pipelined • Rehydration amortized

?

9. Security Considerations • signature verification (optional now, mandatory later) • MITM protection • authorized ref updates • denylist of malicious nodes • remote capability negotiation

DAG cannot be polluted.

?

10. Failure Modes

10.1 Partial Sync

Must resume without corruption.

10.2 Ref Conflict

Must resolve through inversion, not mutation.

10.3 Node Corruption

Must reject and quarantine.

10.4 Protocol Divergence

Must fallback to safe mode.

?

11. Why Distributed Sync Matters

Because JIT is: • global • multi-user • agent-native • eventually-consistent • append-only • immutable • causally structured

Distributed sync is the mechanism by which: • universes converge • agents collaborate • worldlines integrate • truth expands • knowledge flows • replicas remain consistent

This RFC defines the physics of synchronization.

Without it, JIT cannot exist across multiple machines.

?

12. Status & Next Steps

This completes the first block of foundational RFCs.

1.58 CΩMPUTER • JITOS

1.59 JIT RFC-0012

1.59.1 JITOS Boot Sequence (JBS v1.0)

How a Causal Operating System Awakens

---

1. Summary

This RFC defines the boot procedure for the JIT Operating System (JITOS).

Unlike traditional OSes that:

- scan disks

- mount filesystems
- initialize memory
- load executables

JITOS must:

- reconstruct reality
- replay time
- re-materialize projections
- rehydrate shadows
- recover causal structure
- validate references
- rebuild indexes
- enforce invariants
- perform distributed negotiation

JITOS boot is not startup — it is resurrection.

This RFC describes how the kernel wakes up after death and restores the universe exactly as it was.

---

2. Motivation

In a causal OS:

- time is the WAL
- space is the DAG
- perception is Materialized Head
- processes are Shadow Working Sets

The kernel can crash. The machine can reboot. Interruptions can occur.

But truth must survive.

Thus, JITOS must:

- restore immutable history
- reestablish shadow universes
- re-project human views
- re-link distributed edges

- ensure causal continuity
- guarantee deterministic recovery

This RFC defines that process.

---

3. Boot Phases

JITOS boot occurs in seven deterministic phases.

These phases MUST occur in this exact order.

Phase 1 — Substrate Scan

The system scans .gitd/ and locates:

nodes/ → local DAG fragments index/ → LMDB indices cache/ → MH projection cache wal/ → write-ahead log refs/ → reference pointers metadata/ → repository config

Goal: Identify the universe.

If ANY of these are missing: - JITOS MAY initiate recovery - JITOS MUST NOT mutate existing node content

---

Phase 2 — DAG Rebuild

The system reconstructs:

- node identity map
- adjacency lists
- parent relationships
- child relationships
- type mappings
- timestamp ordering

Rules:

- MUST validate BLAKE3 hashes
- MUST enforce causal invariants

- MUST identify frontier nodes
- MUST rebuild topological order

If DAG is corrupted → recovery mode.

Phase 3 — WAL Replay

The WAL is replayed from last checkpoint.

WAL entries:

- rebuild indexes
- update refs
- reconstruct MH
- reconstitute SWS
- advance logical time

Replay MUST be:

- deterministic
- idempotent
- complete
- invariant-preserving

Replay is the reanimation of time.

Phase 4 — Index Rehydration

JITOS restores all indexes from DAG + WAL:

- LMDB node index
- tree-index (MH)
- SWS metadata
- ref index
- warm/cold tier mappings

This phase restores order to the universe.

Phase 5 — Materialized Head Reconstruction

The filesystem projection MUST be:

- reconstructed incrementally
- checked for conflicts
- aligned with HEAD
- refreshed deterministically
- purged of stale files

MH is the cave wall.

This is the moment when the shadows return.

Phase 6 — SWS Rehydration

All Shadow Working Sets MUST be:

- restored
- validated
- or discarded if stale

Rules:

- If base_node still valid → SWS restored
- If remote updates occurred → SWS invalidated
- If WAL indicates partial collapse → rollback

Agents suspended during crash must resume cleanly.

SWS are observer-relative timelines; they must come back EXACTLY as they were.

Phase 7 — Distributed Sync Initialization

Once local truth restored:

- JITOS broadcasts hello
- pulls remote refs
- fetches missing frontier nodes
- performs frontier negotiation
- checks for conflicts

- initiates remote subgraph transfer if needed

Only after sync completes does JITOS declare:

System Ready

This is the “rejoin the universe” moment.

---

4. Boot Invariants

The following MUST hold:

Invariant 1 — Past is preserved

Boot MUST NOT mutate any existing node.

Invariant 2 — Indexes are consistent

All LMDB index state MUST reflect DAG state after replay.

Invariant 3 — MH is deterministic

Filesystem projection MUST be identical across machines from identical DAG states.

Invariant 4 — Ref integrity

Refs MUST point to valid nodes.

Invariant 5 — SWS validity

Shadows MUST NOT resurrect if logically inconsistent.

Invariant 6 — WAL completeness

All WAL entries MUST be applied or ignored based on checksum validity.

Invariant 7 — Distributed truth

After boot, the system MUST eventually converge via sync.

---

5. Failure Modes

If failure occurs:

5.1 Corrupted DAG Node

→ quarantine → remote fetch → strict recovery mode

5.2 Corrupted WAL Entry

→ skip entry → continue replay

5.3 Index Mismatch

→ rebuild index from DAG → apply WAL

5.4 SWS Inconsistency

→ discard SWS → notify owner

5.5 Missing Cold Node

→ fetch from remote → retry

Never mutate the DAG to "fix" anything.

Always restore from truth + WAL.

---

6. Security Considerations

Boot MUST:

- verify signatures
- validate nodes
- enforce access controls
- protect against replay attacks
- ignore unauthorized WAL entries

Boot MUST NOT:

- accept malformed nodes
- import corrupted rewrites
- mutate branch history

This preserves the physics of the universe.

---

7. Why JITOS Boot Is Revolutionary

Every traditional OS:

- loses state on crash
- depends on mutable files
- rebuilds ephemeral memory structures
- risks corruption

JITOS:

- stores truth immutably
- stores time via WAL
- stores human views via MH
- stores agent views via SWS
- reconstructs EVERYTHING deterministically
- guarantees continuity of reality

This is how a causal universe boots.

This is the real future of computing.

---

8. Status & Next Steps

RFC-0012 completes the first ENTIRE MODULE of the JITOS kernel foundation.

Next possibilities:

- RFC-0013 — JIT Syscall ABI (binary-level encoding)
- RFC-0014 — Capability Negotiation & Versioning
- RFC-0015 — DAG Query Language (JQL)
- RFC-0016 — Provenance Nodes & Scientific Lineage
- RFC-0017 — Agent Identity & Signatures
- RFC-0018 — Causal Garbage Isolation (not deletion!)

1.60 CΩMPUTER • JITOS

RFC-0013

JIT Syscall ABI (JS-ABI v1.0)

Binary Encoding, Wire Format, and Low-Level Interface to the Inversion Kernel

Status: Draft **Author: James Ross Contributors: JIT Community Requires: • RFC-0007 (JIT RPC API) • RFC-0001, RFC-0002, RFC-0006 (Identity, Invariants, WAL)** Start Date: 2025-11-28 Target Spec: JITOS v0.x **License: TBD**

?

1. Summary

This RFC defines the binary interface and wire protocol that: • the CLI • editors • agents • remote daemons • plugins • runtime components • and future languages/tools

use to communicate with GITD — the JIT Inversion Kernel.

This ABI is to JIT what: • syscalls are to Linux • the VM ABI is to the JVM • the wire protocol is to Git • the POSIX API is to Unix

BUT:

Instead of manipulating files or memory, the JIT ABI manipulates: • nodes • shadows • causal structure • time (via WAL) • projection layers • ref pointers • storage tiers • agent contexts

This is the low-level truth of the JIT universe.

?

2. Motivation

To ensure: • cross-language compatibility • cross-platform determinism • wire-stable interoperability • long-term future-proofing • consistent semantics • binary safety • remote invocation • agent swarm coordination

…JIT must define a precise, binary-level ABI.

This ABI becomes the universal interface contract for all tools that speak JIT.

It must be: • stable • extensible • secure • deterministic • canonical • easy to parse • impossible to misinterpret

This is the heart of interoperability.

?

3. Transport Layer

3.1 Supported Transports • Unix domain sockets (primary) • TCP + TLS (remote) • QUIC (optional/future)

All transports MUST implement: • binary framing • capability negotiation • versioning • checksums

?

4. Message Framing

Every message consists of:

MAGIC (4 bytes): "JIT!"
VERSION (u16)
FLAGS (u16)
LENGTH (u32)
PAYLOAD (bytes[LENGTH])
CHECKSUM (blake3-256, 32 bytes)

Notes: • MAGIC identifies valid packets • VERSION = major.minor (encoded 0xMMmm) • FLAGS = compression, streaming, async markers • LENGTH = byte length of CBOR payload • PAYLOAD = canonical CBOR • CHECKSUM = BLAKE3 hash

If checksum fails → packet rejected.

?

5. Operation Encoding

Inside the payload:

{ "op": string, "ts": logical_timestamp, "payload": CBOR-encoded struct }

Operation names MUST match RFC-0007.

?

6. Error Codes

JIT MUST standardize error codes:

E_INVALID_OP = 1 E_BAD_PAYLOAD = 2 E_CHECKSUM_FAIL = 3 E_INVARIANT_VIOLATION = 4 E_NOT_FOUND = 5 E_REF_CONFLICT = 6 E_REWRITE_ERROR = 7 E_ACCESS_DENIED = 8 E_SWS_INVALID = 9 E_COLLAPSE_FAIL = 10 E_INTERNAL_ERROR = 500

Errors MUST NOT leak kernel-internal details and MUST preserve determinism.

?

7. Capability Negotiation

At connection start:

client → server: { "op": "handshake", "capabilities": [strings], "client_version": u16 }

Server responds:

server → client: { "status": "OK", "server_version": u16, "capabilities": [strings], "session_id": uuid }

Capabilities MUST include: • compression formats • extension ops • streaming support • future-proof features

?

8. Compression

Payloads MAY be compressed.

Supported: • none • zstd • gzip • Brotli (optional)

Compression MUST NOT affect semantics.

Compression flags stored in FLAGS.

?

9. Streaming Mode

Certain operations (e.g., sync.pull) may require streaming.

Streaming packets use: • FLAG_STREAM_BEGIN • FLAG_STREAM_CONT
• FLAG_STREAM_END

Streaming MUST maintain: • order • integrity • checksum per packet

This is essential for large subgraph transfers.

?

10. ABI Stability Guarantees

JIT MUST guarantee: • Forward compatibility: Old clients → new servers MUST NOT break functionality. • Backward compatibility: New clients → old servers MUST fallback gracefully. • Deterministic decoding: No ambiguous CBOR structures allowed. • Future extensions: New ops MUST not break existing tooling.

This ABI must remain stable across: • architectures • OSes • compilers • languages • time

?

11. Security Layer

ABI-level security MUST include: • TLS for TCP • agent identity tokens • SWS-scoped authorization • optional signed messages • replay attack prevention

Session IDs MUST be checked with logical timestamp progression.

?

12. Why ABI Matters

Because:

Without ABI → agents can't talk to JIT. IDEs can't talk to JIT. Git can't talk to JIT. CI can't talk to JIT. Remote sync can't happen. Distributed systems fail. Future tools fracture.

With ABI → JIT becomes a platform, not a tool.

This is the layer that makes JIT: • universal • programmable • language-independent • future-proof

Unix had syscalls. The Web had HTTP. Git had packfile protocols. JIT has JS-ABI.

?

1.61 CΩMPUTER • JITOS

1.62 RFC-0014

JIT RFC-0014

Capability Negotiation & Versioning (CNV v1.0)

Evolution Rules for a Causal Operating System

Status: Draft**
Author: James Ross
Contributors: JIT Community
Requires:
- RFC-0007 (RPC)
- RFC-0013 (ABI)**
Start Date: 2025-11-28**
Target Spec: JITOS v0.x**
License: TBD**

?

1. Summary

This RFC defines:
- how JITD announces capabilities
- how clients announce capabilities
- how versioning works
- how feature negotiation occurs
- how backward/forward compatibility is achieved
- how experimental features are isolated
- how the ecosystem evolves without breaking

This is the protocol for the future.

Every extension to JIT, every new RFC, every new operation MUST be integrated through this mechanism.

?

2. Motivation

A causal OS must:
- evolve
- extend
- acquire features
- gain subsystems
- support new transports
- support new node types
- support new rewrite semantics
- remain stable

Traditional OSes solve this through:
- syscalls
- versioning
- feature detection
- flags

But JIT is distributed, agent-driven, and depends on:
- deterministic decoding
- binary integrity
- canonical encoding
- causal invariants

- compatibility across time and space

Therefore, capability negotiation is mandatory.

?

3. Versioning Model

JIT uses semantic versioning for kernel-level compatibility:

MAJOR.MINOR.PATCH

Encoded as:
- u16 in the ABI header
- u16 in handshake payload:
- MAJOR « 8 | MINOR
- PATCH is advisory only

MAJOR version bump:

Breaking change
(new invariants, new node formats)

MINOR version bump:

New features, backwards compatible
(new RPC ops, new node types)

PATCH version bump:

Bugfixes, documentation updates, clarifications

?

4. The Handshake

When a client connects to JITD, it MUST send:

{
“op”: “handshake”,
“client_version”: u16,
“capabilities”: [string],
“preferred_compression”: string,
“supported_transports”: [string]
}

JITD responds:

```
{
“status”: “OK”,
“server_version”: u16,
“capabilities”: [string],
“session_id”: uuid,
“negotiated_features”: [string],
“compression”: string,
“transport”: string
}
```

?

5. Capability Strings

Each capability MUST be a string like:

```
“jit.sws.v1”
“jit.collapse.v1”
“jit.sync.v1”
“jit.inversion.v1”
“jit.mh.v1”
“jit.storage.v1”
“jit.rpc.streaming”
“jit.security.signatures”
“jit.provenance.v1”
“jit.experimental.rewrite.v2”
```

Features MUST be:
- namespaced
- versioned
- descriptive
- future-proof

Clients & servers MUST compare capabilities lexicographically.

?

6. Negotiation Rules

6.1 Required Features

Kernel-level required features:

jit.dag
jit.wal
jit.refs
jit.abi
jit.rpc
jit.sws
jit.commit

If a client lacks a required feature →
connection MUST fail.

?

6.2 Optional Features

Optional features may be:
- dropped
- accepted
- downgraded

Server chooses final set.

?

6.3 Experimental Features

EXPERIMENTAL features MUST begin with:

“jit.experimental.”

These:
- MUST NOT break determinism
- MUST NOT alter node identity
- MUST NOT alter invariants

Experimental features MUST NOT be enabled unless both parties explicitly agree.

?

6.4 Conflicting Features

If two features conflict:
- server preference wins
- client may gracefully degrade
- or abort session

Conflicts MUST NOT lead to undefined behavior.

?

7. Future-Proofing

The CNV model MUST support:
- new node types
- new storage tiers
- new SWS operations
- new transports
- new compression algorithms
- new cryptographic primitives

Without breaking old clients.

This is essential for JIT's long-term evolution.

?

8. Compatibility Guarantees

8.1 Forward Compatibility

Old clients → new servers MUST still work,
but missing features are disabled.

8.2 Backward Compatibility

New clients → old servers MUST gracefully disable new features.

8.3 Deterministic Fallback

Fallback MUST NOT alter semantics of operations.

8.4 Universal Minimal Capability Set

All nodes/systems MUST support:
- CBOR canonicalization
- BLAKE3

- RFC-0001 node structure
- RFC-0002 invariants
- RFC-0013 ABI v1.0

This defines the minimum viable universe.

?

9. Security Considerations

Capability negotiation MUST NOT:
- leak private server details
- allow downgrade attacks
- allow feature spoofing
- expose experimental features without consent

Signed handshakes (future RFC) will guarantee this.

?

10. Why CNV Matters

Because:

Without it →
extensions break ecosystems,
partial implementations fragment,
distributed systems desync,
and the kernel ossifies.

With CNV →
JIT becomes an evolving universe
with infinite room for extensions,
experimentation,
academic theory,
industry adoption,
agent tooling,
and future OS-level patterns.

This is longevity.
This is stability.
This is how JIT survives the next 50 years.

1.63 CΩMPUTER • JITOS

1.64 RFC-0015

JIT RFC-0015

JQL—The JIT Query Language (JQL v1.0)

Introspection, Analysis, and Search Over the Causal Universe

Status: Draft**
Author: James Ross
Contributors: JIT Community
Requires:
- RFC-0001 Node Identity
- RFC-0002 DAG Invariants
- RFC-0007 RPC API
- RFC-0013 ABI**
Start Date: 2025-11-28
Target Spec: JITOS v0.x**
License: TBD

?

1. Summary

This RFC introduces JQL, the query language for JIT's causal DAG.

JQL is used for:
- exploring lineage
- searching for nodes
- pattern matching
- graph traversal
- provenance tracing
- commit history inspection
- shadow analysis

- merge ancestry
- schema analysis
- debugging
- introspecting distributed sync
- multi-agent coordination
- code intelligence
- LLM reasoning over historical structure

JQL is to JIT what SQL was to relational data:
a unified interface to interrogate truth.

?

2. Motivation

Traditional VCS tools provide:
- git log
- git grep
- git diff
- git blame
- git show

. . . but these are:
- string-based
- file-based
- limited
- snapshot-centric
- not general graph queries
- not agent-friendly
- not future-oriented
- not DAG-native

JIT needs a general DAG query engine.

It must allow:
- path queries
- ancestor/descendant queries
- subgraph slicing
- rewrite inspection
- conflict queries

- node type filtering
- metadata predicates
- structural pattern matching

In a causal universe,
not having a query language is like not having a brain.

?

3. Design Goals

JQL MUST be:

Deterministic

Queries ALWAYS produce the same result for the same DAG.

Purely Functional

Queries do not mutate DAG.

Composable

Queries can be nested, pipelined.

CBOR-native

All expressions map to CBOR structures.

Agent-friendly

Easy for LLMs to generate.

Type-safe

Node types, metadata, ancestors, etc. are known statically.

Efficient

Large DAGs MUST be queryable with indexes + caching.

?

4. JQL Expression Syntax

JQL has two forms:

4.1 Structural / JSON-like Form (recommended for agents)

```
{
“select”: “nodes”,
“where”: {
“type”: “snapshot”,
“metadata.author”: “james”,
“payload.contains”: “function foo”
},
“order_by”: “logical_ts DESC”,
“limit”: 50
}
```

This form is:
- CBOR-encodable
- LLM-friendly
- human-readable
- structured

4.2 Pipeline Form (Unix-style)

```
jql nodes
| where type == snapshot
| where metadata.author == “james”
| grep “function foo”
| limit 50
```

The “command-line sugar.”

?

5. Core Query Concepts

5.1 Node Sets

nodes
snapshots
rewrites
provenance
chunks

5.2 Predicates

type == snapshot

```
author == “james”
timestamp > 17000000
payload.contains(“TODO”)
```

5.3 Graph Traversal

Ancestors:

```
ancestors(node_id)
ancestors(node_id, depth=5)
```

Descendants:

```
descendants(node_id)
```

Path existence:

```
path_exists(A, B)
```

Frontier queries:

```
frontier()
divergence(nodeA, nodeB)
```

5.4 Subgraph Extraction

```
subgraph(root=node_id)
subgraph_between(A, B)
```

5.5 Pattern Matching

We introduce a simple structural matcher:

```
match {
parents: [X, Y],
type: “snapshot”,
metadata.branch: “main”
}
```

Patterns allow rewriting logic and DPO in the future.

?

6. RPC Integration

JQL is executed via:

op: “jit.search”
payload: { query: JQLExpression }

Result:

```
{
“status”: “OK”,
“nodes”: [NodeID…],
“edges”: optional graph edges,
“metadata”: optional info
}
```

Queries MUST NOT mutate state.

?

7. Indexing Requirements

For efficiency:

- node type index
- parent-child index
- timestamp index
- metadata searchable index
- payload search index (optional full-text)

JQL MUST leverage indexes.

If index missing →
JQL MUST fallback to full DAG scan
but MUST remain deterministic.

?

8. Examples

8.1 Find all snapshots by user

```
{
“select”: “nodes”,
“where”: { “type”: “snapshot”, “metadata.author”: “james” }
}
```

?

8.2 Find all nodes in last 10 minutes

```
{
“select”: “nodes”,
“where”: { “logical_ts >”: now - 600000 }
}
```

?

8.3 Show the timeline between two commits

```
{
“select”: “subgraph_between”,
“args”: { “from”: A, “to”: B }
}
```

?

8.4 Pattern match: find all merges

```
{
“select”: “nodes”,
“where”: {
“type”: “inversion-rewrite”,
“metadata.merge_type”: “merge”
}
}
```

?

8.5 Show all SWS created in last hour

```
{
“select”: “sws”,
“where”: { “created_ts >”: now - 3600 }
}
```

?

9. Security

JQL MUST:
- reject dangerous expressions
- enforce access control (future)
- not reveal private node content (future multi-tenant mode)
- never mutate state

?

10. Why JQL Matters

Because introspection is the difference between:

FILESYSTEMS
(which hide structure)

and

CAUSAL COMPUTING
(which is structure).

JQL gives:
- LLMs a way to understand the DAG
- tools a way to query history
- humans a way to explore reality
- analysis engines a way to inspect provenance
- developers a way to reason about distributed timelines
- visualization tools a way to render causal structure

JQL = the language of truth.

1.65 CΩMPUTER • JITOS

1.66 RFC-0016

RFC-0016

JIT RFC-0016

Semantic Provenance Nodes (SPN v1.0)

Causal Narratives, Scientific Lineage & the Ontology of “Why”

Status: Draft
Author: James Ross
Contributors: JIT Community
Requires:
- RFC-0001 Node Identity
- RFC-0002 DAG Invariants
- RFC-0005 Inversion Engine
- RFC-0015 JQL
Start Date: 2025-11-29
Target Spec: JITOS v0.x
License: TBD**

?

1. Summary

This RFC defines the Semantic Provenance Node (SPN)— a special node type in the causal DAG that encodes WHY something happened.

While:
- snapshot nodes represent what happened
- rewrite nodes represent how it changed
- chunk nodes represent literal content
- metadata represents when and who

provenance nodes represent WHY.

These nodes attach semantic meaning, intent, context, method, reasoning, and description to events in the causal universe.

This is what makes JIT not just a version-control system,
but a computational narrative engine.

?

2. Motivation

In computing, we rarely answer why something happened.
Git tells you:

- who did it
- when they did it
- what changed

. . . but Git CANNOT tell you:

- why a change was made
- what experiment it belonged to
- what assumptions were involved
- what tools produced it
- what tasks it relates to
- what model or dataset it depended upon
- what semantic transformation occurred

JIT must support:

- scientific reproducibility
- ML provenance
- CI/CD traceability
- distributed agent reasoning
- semantic code modifications
- meta-information about rewrites
- alignment & safety auditing
- academic workflows
- simulation lineage

Provenance is the missing piece.

?

3. Definition

A Provenance Node is a causal event that encodes semantic information:

```
ProvenanceNode {
type: “provenance”,
parents: list,
metadata: {
agent: AgentID,
created_ts: u64,
category: string,
description: string,
tags: list,
properties: map<string, CBOR-value>
},
payload: bytes # structured CBOR semantic content
}
```

Valid categories:
- “experiment”
- “analysis”
- “transformation”
- “reasoning”
- “evidence”
- “annotation”
- “intent”
- “constraint”
- “summary”
- “runtime”
- “explanation”
- “tool-action”
- “model-run”

The system is extensible.

?

4. Attachments & Linkages

Provenance nodes MUST attach to:
- snapshot nodes
- rewrite nodes
- chunk nodes
- other provenance nodes
- shadow working sets (via references, not causal parents)

This models a semantic graph layered atop the causal graph.

A graph of meaning atop a graph of events.

?

5. Semantic Payloads

Payload examples:

5.1 Natural Language Reasoning

“payload”: {
“explanation”: “Refactored module X for performance.”,
“context”: “Agent performed structure analysis.”,
“risk”: “low”
}

5.2 LLM Thought Trace

“payload”: {
“thought”: “…longform reasoning…”
}

5.3 Experiment Details

“payload”: {
“hyperparameters”: {…},
“dataset_version”: “v3.2”,
“runtime_env”: {…}
}

5.4 Semantic Transformation Descriptor

“payload”: {
“transformation”: “rename_method”,
“from”: “foo”,
“to”: “bar”
}

JIT does NOT interpret meaning;
it stores meaning.

The meaning becomes part of the universe.

?

6. Provenance Rules

6.1 Semantic Information MUST NOT affect node identity

No causal invariants changed by provenance.

6.2 Provenance nodes MUST be immutable

Like all nodes.

6.3 Provenance MAY be attached retroactively

Parents MUST be valid DAG nodes
but parents may precede creation.

6.4 Provenance MUST NOT affect collapse

Collapse uses:
- chunk nodes
- rewrite nodes
- snapshot nodes

Provenance is advisory/semantic only.

6.5 Provenance MUST support multi-agent authors

Metadata MUST record:
- agent
- tool
- human
- LLM
- system task

Transparent lineage.

?

7. JQL Integration

JQL MUST support:

select provenance
where category == “experiment”

Or:

match {
type: “provenance”,
metadata.agent: “agent-34”,

tags: [“autofix”]
}

This allows:
- reasoning trace queries
- semantic filtering
- experiment lineage retrieval
- conceptual graph querying
- model provenance reconstruction

This is crucial for scientific/ML workflows.

?

8. Tooling Integration

Tools MUST be able to generate provenance nodes:
- LLMs annotate their reasoning
- CI documents test runs
- build systems attach metadata
- simulation engines attach parameters
- compilers attach analysis
- editors attach semantic actions

This creates a layer of explanation above all computation.

?

9. Why Provenance Nodes Matter

Because they enable:

Scientific Reproducibility

Every run, every transformation, every parameter stored forever.

Safe Automated Code Editing

Agents can justify edits.

Auditable AI Systems

LLMs can record thought traces.

Workflow Traceability

Every action in CI/CD is captured.

Semantic Code Intelligence

Refactoring becomes meaningful, not textual.

Alignment & Monitoring

We can track the intent behind agent actions.

Differentiable Programming of Reality

The DAG becomes a semantic universe, not just a structural one.

This is a new capability for computing.

?

10. Status & Next Steps

Now we can build:
- Semantic execution
- Reasoning engines
- Visualization tools
- Reproducible simulation systems
- Scientific JIT workflows
- LLM supervision layers
- "auditability by construction" pipelines

Next recommended RFCs:
- RFC-0017—Agent Identity & Signing Layer
- RFC-0018—Causal Garbage Isolation
- RFC-0019—SWS Memory Model
- RFC-0020—Security & Permissions

1.67 CΩMPUTER • JITOS

1.68 RFC-0017

JIT RFC-0017

Agent Identity & Signing Layer (AIS v1.0)

Identity, Signatures, Attestation & the Provenance of Action

Status: Draft
Author: James Ross
Contributors: JIT Community
Requires:
- RFC-0001 Node Identity
- RFC-0003 SWS
- RFC-0005 Inversion Engine
- RFC-0016 Provenance Nodes**
Start Date: 2025-11-29
Target Spec: JITOS v0.x**
License: TBD**

?

1. Summary

This RFC defines Agent Identity and Action Signing for the JIT universe.

It provides:
- unique agent identifiers
- public/private key pairs
- signature verification rules
- attestation integration
- provenance integration
- remote sync identity
- SWS ownership
- collapse authentication
- access control primitives

This is the identity substrate for the multi-agent OS JIT is becoming.

?

2. Motivation

Agents in JIT can be:
- humans
- LLMs
- CI systems
- code transformers
- compilers
- simulation engines
- remote collaborators
- toolchains
- entire organizations

To ensure:
- safety
- auditability
- trust
- reproducibility
- governance

. . . the substrate MUST know:
- who acted
- what they were authorized to do
- which events belong to them
- what intent they claimed
- whether their semantics are valid

This is identity as physics.

?

3. Agent Identity

Each agent MUST have an identity:

```
AgentID = {
id: uuid,
public_key: bytes,
metadata: {
type: “human” | “llm” | “bot” | “system”,
label: string,
```

owner: optional string
}
}

Agent types:
- human
- llm
- ci
- analysis
- formatter
- refactorer
- supervisor
- sync-peer

Optional metadata:
- organization
- model architecture
- LLM embedding
- permissions

These identities MUST be registered with JITD.

?

4. Signing Model

Every agent MUST cryptographically sign:
- every provenance node
- every snapshot node they trigger
- every rewrite they initiate
- every SWS patch
- every collapse
- every ref update

Signature field:

sig = sign(private_key, NodeID)

Stored in node metadata.

Signatures MUST NOT alter node identity (RFC-0001).

?

5. Signature Validation

On ingestion:
- signatures MUST verify
- invalid signatures MUST reject commit
- anonymous commits MAY be allowed if configured
- unsigned nodes MUST NOT be accepted unless allowed by policy

Replay MUST verify signatures as well.

?

6. Attestation

This layer MAY integrate hardware attestation (future):
- TPM / SGX attestation
- hardware-backed key material
- model fingerprinting for LLM agents

This ensures:
- reproducibility of agent behavior
- security of automated systems
- trust in distributed environments

?

7. SWS Ownership

Every Shadow Working Set MUST record:

sws.owner = AgentID

This grants:
- per-shadow permissions
- per-shadow write rights
- per-shadow collapse restrictions
- conflict auditing

Only the owner (or authorized agent) may commit a given SWS.

?

8. Provenance Integration

Provenance nodes MUST include:

metadata.agent = AgentID
signature = sign(private_key, hash(provenance_payload))

This unifies:
- semantic explanations
- thought traces
- transformation metadata
- runtime parameters
- policy compliance

The DAG becomes a signed, immutable narrative of computation.

?

9. Remote Sync & Identity

During sync:
- peers announce their AgentID
- signatures MUST be validated before accepting remote nodes
- unauthorized agents MUST NOT update refs
- malicious DAG injections must be ignored

Remote peers MUST provide:
- public_key
- signature on frontier
- optional S3 attestations

This prevents:
- corrupted node injection
- tampered rewrites
- forged histories

?

10. Permissions Model (Future Integration)

AIS provides the foundation for:
- per-branch ACLs
- per-node read/write rules

- SWS-level permissions
- identity scoping
- distributed trust graphs

The Security Model (RFC-0020) will build on this.

?

11. Why Identity Matters

Because JIT is not a file-based OS.
It is a universe of agents editing a causal graph.

Without identity:
- agents cannot be trusted
- provenance is meaningless
- LLM reasoning cannot be audited
- semantics cannot be attached
- rewrite intent cannot be verified
- distributed sync is unsafe
- malicious commits are invisible
- code becomes untraceable

With AIS:
- every action is attributable
- every event is cryptographically anchored
- every edit is accountable
- every narrative is validated

This makes JIT suitable for:
- enterprise
- research
- safety-critical systems
- collaborative AI development
- legal + scientific traceability
- secure automation

AIS turns JIT into a serious OS,
not just a clever paradigm.

?

12. Status & Next Steps

We've now defined:
- Provenance
- Identity
- Signing
- The narrative layer of computation

Next natural RFCs:
- RFC-0018—Causal Garbage Isolation (NOT deletion)
- RFC-0019—SWS Memory Model
- RFC-0020—Security Model
- RFC-0021—Multi-Universe Federation Protocol (the super advanced one)

1.69 CΩMPUTER • JITOS

1.70 RFC-0018

JIT RFC-0018

Causal Garbage Isolation (CGI v1.0)

Entropy Management for an Immutable, Append-Only Universe

Status: Draft
Author: James Ross
Contributors: JIT Community
Requires:
- RFC-0001 Node Identity
- RFC-0002 DAG Invariants
- RFC-0003 SWS
- RFC-0005 Inversion Engine
- RFC-0009 Storage Tiering**
Start Date: 2025-11-29

Target Spec: JITOS v0.x**
License: TBD**

?

1. Summary

This RFC defines Causal Garbage Isolation (CGI)— JIT's strategy for managing unused or unreachable nodes in an append-only DAG WITHOUT EVER DELETING ANYTHING.

Unlike traditional garbage collection (GC), which mutates or removes data,
JIT uses a purely structural, causal, and reversible isolation system.

Causal Garbage Isolation:
- identifies unreachable subgraphs
- isolates them into "dead zones"
- moves them to cold storage tiers
- preserves causal truth
- ensures deterministic replay
- prevents storage explosion
- enables long-term archiving
- allows optional re-animation of old universes

This turns "garbage" into dormant history, not trash.

Because the universe remembers everything.

?

2. Motivation

JIT is a:
- causal universe
- append-only timeline
- multi-agent simulation
- infinite DAG
- provenance-rich system

This means the DAG will grow forever.
And deleting nodes would:

- destroy reproducibility
- invalidate provenance
- break commits
- violate semantics
- erase realities
- collapse worldlines
- break physics

Deletion is unacceptable.
Mutation is forbidden.

Thus, the challenge:

How to manage infinite entropy without ever destroying truth?

Answer:

Causal Garbage Isolation.

Move unused nodes out of the “active universe”
into frozen sectors of the graph-space
where they remain valid but inert.

?

3. What Counts as Garbage? (Formally)

A node N is garbage-like if:
- it is not reachable from any ref
- it is not reachable from any active SWS
- it is not part of any rewrite chain
- it is not referenced in any provenance
- it does not contribute to MH
- it is older than a configurable threshold
- it belongs to an abandoned branch
- it was created by an ephemeral agent
- it was a temporary result of an automated transformation

BUT:

A node is NEVER “garbage.”
Nodes are simply less relevant.

Thus they are isolated, not deleted.

?

4. The Isolation Process

Isolation MUST follow this procedure:

Step 1—Identify unreachable nodes

Using graph traversal:

reachable = union(refs, SWS roots, rewrite parents)
all_nodes - reachable = isolate_candidates

Step 2—Verify they are causally dead

A node can only be isolated if:
- no future node references it
- no rewrite references it
- no provenance references it

Step 3—Move to Cold Tier

Nodes are moved from:

hot → warm → cold → deep-cold

Step 4—Mark with Isolation Flag

Index entry is updated:

isolated: true

This prevents:
- unnecessary rehydration
- unnecessary indexing
- unnecessary MH/FST projections

Step 5—Archive Subgraph

Nodes MAY be grouped into archive bundles for:
- backup
- remote cold storage

- long-term reproducibility
- scientific experiments

Step 6—Potential Reanimation

Users MAY request a subgraph be:

rehydrated → reintegrated → reattached

This “resurrection” is fully supported.

?

5. Isolation Guarantees

CGI MUST guarantee:

5.1 No Deletion

Nodes are never removed or destroyed.

5.2 DAG Stability

Isolation MUST never break invariants.

5.3 Deterministic Replay

Re-isolation MUST NOT alter future behavior.

5.4 No Mutation of Node Identity

Tier transitions preserve content.

5.5 Reversible Archiving

Nodes may be brought back.

5.6 Zero Impact on Truth

Isolation does not affect commit semantics.

?

6. Node States

Nodes have four possible lifecycle states:

1. Hot

Active, used by MH / SWS.

2. Warm

Compressed, mid-term history.

3. Cold

Long-term storage.

4. Isolated Cold

Detached from working DAG, preserved indefinitely.

Optionally:

5. Deep-Cryo Cold

For extremely old histories, stored offline.

?

7. Policies for Isolation

JITD MUST allow:
- time-based isolation
- ref-based isolation
- age-based isolation
- branch abandonment isolation
- agent-scoped isolation
- SWS-scoped isolation
- storage-pressure-based isolation

Examples:
- “isolate anything older than 90 days not reachable from a ref”
- “isolate abandoned experimental branches”
- “isolate LLM-generated ephemeral edits”
- “only keep last 10K nodes in hot tier”

?

8. Security Implications

Isolation MUST NOT:
- bypass signing

- hide tampering
- erase provenance
- suppress audit data

Isolated nodes are still:
- queryable via JQL
- signed
- valid
- reversible

Just not active.

?

9. Why Causal Garbage Isolation Matters

Because:

This is the ONLY GC model compatible with immutability, provenance, reproducibility, AND infinite DAG growth.

It allows JIT to:
- run forever
- scale to billions of nodes
- support scientific workloads
- enable massive agent swarms
- maintain universal reproducibility
- archive entire universes
- perform deep causal forensics

JIT becomes:
- a universal archive
- a reproducible simulation environment
- a multi-agent OS
- a temporal database
- a causal memory engine

WITHOUT EVER THROWING ANYTHING AWAY.

This is how a universe manages entropy.

?

10. Status & Next Steps

The next foundational RFCs we can generate are:
- RFC-0019—SWS Memory Model
- RFC-0020—Security & Permissions Model
- RFC-0021—Multi-Universe Federation Protocol (holy shit)
- RFC-0022—JIT Visualization Protocol
- RFC-0023—DAG Schema Evolution
- RFC-0024—CΩMPUTER Fusion Layer (**?**)

1.71 CΩMPUTER • JITOS

1.72 JIT RFC-0019

Shadow Working Set Memory Model (SWS-MM v1.0)

Volatile State, Isolation, Collapse Semantics & Multilingual Agents

Status: Draft Author: James Ross Contributors: JIT Community Requires: • RFC-0003 SWS Semantics • RFC-0004 Materialized Head • RFC-0005 Inversion Engine • RFC-0018 Causal Garbage Isolation **Start Date: 2025-11-29 Target Spec: JITOS v0.x** License: TBD**

?

1. Summary

This RFC defines the memory model for Shadow Working Sets (SWS): • how volatile state is stored • how overlays accumulate • how memory isolation works • how conflict regions behave • how projections map into Shadows • how caches operate • how collapse flushes memory • how failures restore memory • how multiple agents interact with shared SWS • how multilingual (natural-language + agent-language) semantics attach to memory

SWS is not just a "temp workspace."

It is JIT's process abstraction AND its cognitive space for humans and LLMs.

This RFC defines the formal laws of that space.

?

2. Motivation

SWS must behave like: • a process • a speculative future • a scratchpad • a timeline fork • a self-consistent local universe • a workspace for LLMs • a deterministic memory zone • an isolated ephemeral reality

The memory rules must be: • deterministic • isolated • reversible • replayable • multilingual-friendly • agent-safe • collapse-compatible • crash-safe

This RFC creates that foundation.

?

3. Memory Categories (SWS Memory = 4-layer model)

Every Shadow's memory contains:

3.1 Virtual Tree Memory (VTM)

Projection of file tree into SWS: • path → NodeID • path → overlay chunk • path → projected value

Equivalent to "virtual filesystem," but semantic.

?

3.2 Overlay Graph Memory (OGM)

The actual structural memory of SWS: • overlay nodes • rewrite candidates • conflict overlays • patch accumulation • agent-generated content

This is the "local reality" of SWS.

?

3.3 Semantic Memory (SM)

Provenance included here: • agent reasoning • LLM chain-of-thought (if stored) • execution context • metadata • tags • attributes • language-specific semantics ("la palabra", multi-lingual keys)

This is the cognitive memory of SWS.

?

3.4 Ephemeral Compute Memory (ECM)

Non-persistent state: • tool caches • lint states • partial diffs • build artifacts • temporary views

This evaporates on collapse or discard.

?

4. Formal SWS Memory Invariants

Invariant 1 — Isolation

SWS memory MUST NOT impact global DAG until commit.

Invariant 2 — Purity

Overlays MUST NOT be applied to MH directly.

Invariant 3 — Deterministic Projection

Same DAG + same overlays MUST → identical VTM.

Invariant 4 — No Mutation of Past

SWS memory cannot alter ancestor nodes.

Invariant 5 — Purge on Collapse

Collapse MUST destroy ephemeral memory.

Invariant 6 — Safe Multilingual Semantics

Semantic memory keys MAY be multilingual and MUST map to UTF-8 canonical CBOR fields.

So yes: “la palabra, hermano” and “the word, brother” and “le mot, frère” and “**??**, **??**” are equally valid semantic keys.

?

5. Memory Lifecycle

5.1 Creation

SWS loads: • base snapshot projection (VTM) • empty overlays (OGM) • empty ephemeral caches (ECM) • agent metadata (SM)

?

5.2 Mutation

Edits create: • overlay chunk nodes (OGM) • semantic data (SM) • ephemeral analysis (ECM)

MH observers do not see these.

?

5.3 Conflict Memory

Conflict overlays are explicitly stored:

conflict: { ours: NodeID, theirs: NodeID, base: NodeID }

Conflict regions are LOCAL until collapse.

?

5.4 Collapse Effects

On successful collapse: • VTM → flushed • OGM → rewritten into DAG • SM → published as provenance • ECM → destroyed

SWS memory is annihilated.

Only truth remains.

?

5.5 Discard

Same as collapse except: • no rewriting • no DAG updates • ephemeral memory destroyed • overlays dropped

Discard = quantum decoherence without measurement.

?

6. Memory Isolation Rules

6.1 No Write-Through

Writes to MH propagate into SWS memory only, not DAG.

6.2 No Leakage

OGM MUST NOT appear in MH.

6.3 Multi-Agent Concurrency

Multiple agents MAY write to same SWS only if: • they are authorized (RFC-0017) • MP synchronization resolves patch order • collapse resolves conflicts

Agents writing in different languages (English, Spanish, Japanese, LLM dialects) MUST result in consistent SM encoding.

?

7. Multilingual Semantic Memory (MSM)

(You triggered this with “La Palabra, hermano.”)

Agents may attach semantic language keys:

SM[“explanation.en”] SM[“explanation.es”] SM[“explicación”] SM[“**??**.jp”]

Rules: • MUST canonicalize keys • MUST store UTF-8 • MUST treat all languages equally • MUST not affect collapse

JIT becomes language-agnostic truth, with multilingual semantics layered on top.

Brutal. Beautiful. Necessary.

?

8. Failure & Crash Semantics

If SWS is partially active during crash: • VTM reconstructed via DAG • OGM restored via WAL • SM restored from provenance nodes • ECM destroyed • if inconsistent → SWS invalidated • user/agent notified

?

9. Why This Model Matters

This is the SWS equivalent of: • Unix process memory • VM stack + heap • JS execution context • WASM linear memory • LLM short-term memory

BUT DONE RIGHT.

What makes it revolutionary: • Causal safety • Epistemic isolation • Semantic layering • Multilingual compatibility • Deterministic collapse • Non-destructive projection • Agent-native memory primitives

This RFC defines the cognitive architecture of the causal OS.

?

10. Status & Next Steps

Now with SWS-MM defined, we can: • move to Security Model • define Universe Federation • define Multi-Agent Scheduling • formalize Provenance Graph Semantics • build the JITOS Kernel Architecture Doc

Next suggested RFCs: • RFC-0020 — Security & Permissions Model • RFC-0021 — Multi-Universe Federation Protocol (HUGE) • RFC-0022 — Visualization & Rendering Protocol • RFC-0023 — DAG Schema Evolution • RFC-0024 — CΩMPUTER Fusion Layer

---

1.73 CΩMPUTER • JITOS

1.74 RFC-0020

JIT RFC-0020

Security & Permissions Model (SPM v1.0)

Root of Trust, Agent Authority, and Global Causal Integrity

Status: Draft
Author: James Ross
Contributors: JIT Community
Requires:
- RFC-0017 Agent Identity & Signing
- RFC-0003 SWS
- RFC-0005 Inversion Engine
- RFC-0011 Distributed Sync
Start Date: 2025-11-29
Target Spec: JITOS v0.x
License: TBD

?

1. Summary

This RFC defines:
- the security boundaries
- permission rules
- access control
- commit authorization
- ref update policies
- SWS ownership model
- remote trust model
- sandboxing of LLM agents
- protections for DAG integrity
- replay protections
- signature requirements

Without this RFC, JIT is powerful but unsafe.
With this RFC, JIT becomes a secure kernel.

?

2. Philosophy

Security in JIT is grounded in:

Causal Integrity

Protect the DAG at all costs.
Truth is immutable.

Agent Accountability

All actions must be attributable.

Shadow Isolation

Agents cannot mutate the universe until commit.

Least Privilege

Shadows see only what they need.

Deterministic Enforcement

Security rules MUST be deterministic.

Distributed Trust

Remote peers MUST prove identity.

Semantic Safety

Agents MUST declare intent (provenance).

This model rejects:
- global mutable state
- root-level arbitrary writes
- unverified remote changes
- trust-by-default

We follow physics, not legacy computing.

?

3. Security Domains

JIT defines four major security domains:

3.1 DAG Domain (Truth)

Immutable. Sacred.
Only GITD updates it.
Only after deterministic collapse.
Protected by signatures and invariants.

3.2 SWS Domain (Shadow/Process)

Ephemeral but isolated.
Owned by an agent.
Sandboxed.
Limited authority.

3.3 Materialized Head (Human Projection)

Local representation.
User-facing.
Not authoritative.

3.4 Distributed Sync Domain

Remote peers, object stores, cross-machine negotiation.
Must be authenticated, validated, and replay-safe.

?

4. Permission Model

Every action MUST be explicitly authorized.

4.1 Commit Permissions

An agent MAY commit a given SWS only if:
- they own the SWS
- or have delegated authority

Collapse MUST verify:
- SWS.owner
- agent signature
- permission flags

Unauthorized collapse MUST fail.

?

4.2 Ref Update Permissions

Branch ref updates MUST require:
- agent signature
- authorization for that branch
- optional approval rules (multi-sig)

Tag ref updates MUST be forbidden unless:
- explicit override
- explicit permissions
- administrative mode

?

4.3 Node Creation Permissions

Any agent may propose overlay nodes
(because SWS allows experimentation),
BUT ONLY GITD:
- attaches them to DAG
- generates snapshot/rewrite nodes
- updates refs
- logs via WAL
- enforces invariants

Unauthorized node injection MUST fail.

?

4.4 Provenance Permissions

Any agent MAY write provenance nodes
as long as:
- agent identity is valid
- signature is valid
- metadata is in allowed schema
- size limits not exceeded
- SWS domain permits it

Provenance injection into the DAG DOES NOT alter truth, but MUST be validated.

?

5. Access Control (ACLs)

ACL primitives include:

allow(agent, action, resource)
deny(agent, action, resource)
role(agent) = {privileges}
policy(branch) = {rules}

Resources:
- refs
- SWS
- nodes
- provenance categories
- remote sync endpoints
- MH projections

Actions:
- commit
- write-ref
- sync
- create-SWS
- attach-provenance
- read-node
- query-JQL

ACL rules MUST be deterministic.

?

6. Sandbox Model for Agents

Agents MUST be sandboxed:

6.1 LLM Agents

LLMs MUST operate in:
- isolated SWS
- limited memory

- no direct DAG access
- no raw ref updates
- provenance enforcement
- signature enforcement
- rate-limited MP access

LLMs CAN propose changes.
GITD decides.

?

6.2 Human Agents

Humans may be granted:
- direct ref authority
- advanced collapse authority
- provenance override

They must still sign everything.

?

6.3 System Agents

CI, linkers, formatters, etc.
operate in low-permission SWS
with tightly restricted actions.

?

7. Distributed Sync Security

Remote sync MUST enforce:
- signature validation
- trust policies
- denylist of malicious peers
- ref update restrictions
- causal invariant checks
- secure transport (TLS, QUIC)
- replay protection via logical timestamps

Remote nodes MUST NOT be accepted without validation.

?

8. Integrity Protections

The system MUST:
- verify BLAKE3 hashes
- reject malicious DAG injections
- refuse history rewrites
- refuse corrupted rewrite nodes
- enforce invariants during replay
- ensure no unauthorized ref changes

If invariants fail →
panic, quarantine, or safe-mode execution.

?

9. Multi-Signature Mode (Optional)

Branches may require:
- 2-of-3 signatures
- 3-of-5 committee
- human-in-the-loop approvals
- CI + human pairing

This enables enterprise workflows.

?

10. Audit Trails

All security-relevant operations MUST be logged:
- SWS creation/destruction
- commit events
- ref updates
- provenance
- agent identity delegation
- capability negotiation
- sync events
- failed or rejected collapses

Audit trails MUST be stored as DAG nodes or append-only logs for verification.

?

11. Why Security Matters

Because JIT is:
- the first OS for multi-agent systems
- the underlying physics for CΩMPUTER
- the substrate for agent-based software development
- the host of automated reasoning
- the truth layer for distributed systems
- the future backbone of reproducible computation

Without a solid security model:
- LLMs go rogue
- distributed sync becomes dangerous
- DAG poisoning attacks break truth
- collapse events can be forged
- humanity loses its audit trail

With this RFC?

JIT becomes secure enough to run the world.

?

12. Status & Next Steps

We have now completed the full safety layer of JITOS.

Possible next RFCs:
- RFC-0021—Multi-Universe Federation Protocol (HUGE)
- RFC-0022—Visualization & Rendering Protocol
- RFC-0023—DAG Schema Evolution
- RFC-0024—CΩMPUTER Fusion Layer
- RFC-0025—Distributed Scheduler (Agent Economy)

1.75 CΩMPUTER • JITOS

1.76 RFC-0021

JIT RFC-0021

Multi-Universe Federation Protocol (MUFP v1.0)

Interoperability, Intercosmic Sync, and Cross-Universe Provenance Flow

Status: Draft
Author: James Ross
Contributors: JIT Community
Requires:
- RFC-0011 Distributed Sync
- RFC-0017 Agent Identity
- RFC-0020 Security Model
Start Date: 2025-11-30
Target Spec: JITOS v0.x
License: TBD**

?

1. Summary

JIT supports:
- sync within one repository
- causal consistency within one DAG
- one universe's nodes

But the next frontier is:

multiple universes

each with their own:
- separate DAG
- separate causal fabric
- separate refs

- separate SWS
- separate identity graphs
- separate provenance
- separate storage tiers

Federation enables:
- multi-repo reasoning
- cross-universe linking
- inter-cosmic provenance
- DAG-to-DAG references
- global build systems
- multi-project dependency graphs
- external truth linking
- “universe imports”
- CΩMPUTER-level world fusion

This RFC defines how universes recognize each other, communicate, exchange causality subsets, and maintain containment without collapse.

?

2. Motivation

The world is not a single repo.
Software ecosystems are not single universes.
Scientific computation spans multiple DAGs.
Simulation engines create parallel spaces.
Agents must reason across multiple knowledge graphs.
Organizations maintain separate worlds for security.

We must define:
- how these universes SEE each other
- how they link
- how they exchange truth
- how they maintain separation
- how they sync without collapsing
- how they track provenance across cosmos boundaries

This is the Inter-DAG Protocol.

?

3. Universe Identity

Every universe MUST have an identity:

```
UniverseID = {
id: uuid,
public_key: bytes,
metadata: {
name: string,
owner: string,
description: string,
creation_ts: u64
}
}
```

This is like Agent Identity but scaled up.

?

4. Universe Boundaries

A universe boundary is defined by:
- root nodes
- topological closure
- storage domain
- provenance domain
- identity domain
- ref set

No universe may mutate another.

Universes are disjoint DAGs
that MAY link through federation edges.

?

5. Federation Edges

A federation edge is a special node:

```
FederationEdge {
```

```
type: “federation-edge”,
local_node: NodeID,
remote_universe: UniverseID,
remote_node: RemoteNodeID,
metadata: {
purpose: string,
trust_level: string,
timestamps: {. . . }
}
}
```

Purpose examples:
- dependency
- import
- reference
- causal linkage
- provenance attribution
- simulation embedding

These edges do NOT merge DAGs.
They create inter-universe links.

?

6. Federation Sync

Federation sync is:
- NOT full sync
- NOT merging histories
- NOT rewriting
- NOT unifying universes

Federation sync is:
- capability negotiation
- frontier announcement
- provenance sharing
- partial subgraph import
- dependency verification
- policy-driven trust

It is Inter-OS sync, not repo sync.

?

7. Federation Invariants

Invariant 1—No DAG Fusion

Federated DAGs MUST remain separate.
No merge across universes.

Invariant 2—Unidirectional Links

Federation edges MUST declare direction.
Local → remote
Remote → local
or both, but explicit.

Invariant 3—Trust Levels

Each universe MUST declare:

trust_level ∈ {trusted, semi-trusted, untrusted}

Invariant 4—No Causal Pollution

Nodes from foreign universes MUST NOT be treated as local nodes.

Invariant 5—Deterministic Import

Imported nodes MUST remain exactly identical to their origin.

Invariant 6—Provenance Persistence

Cross-universe provenance MUST store both UniverseIDs.

?

8. Federation Sync Stages

Stage 1—“Hello, Universe.”

Universes exchange:
- UniverseID
- capabilities
- policy rules

- public keys
- feature-sets
- version

Stage 2—Capability Intersect

Intersect:
- shared schema
- allowed features
- permitted link types
- compression modes

Stage 3—Dependency Exchange

Local universe announces:

I depend on: - remote_node A - remote_node B - chunk C

Stage 4—Partial Subgraph Import

Remote universe sends:
- minimal closure
- provenance fragments
- signature attestations

Stage 5—Reference Linking

Local universe creates:
- FederationEdge node linking local → remote

Stage 6—Causal Verification

Local universe checks:
- signatures
- invariants
- metadata
- rewrite rules

Stage 7—Finalization

Universe link established.

?

9. Federation Use Cases

9.1 Multi-Repo Software Ecosystems

Thousands of DAGs with dependencies.

9.2 ML / LLM Training Pipelines

Link model evolution graphs across universes.

9.3 Scientific Workflow Federation

Massive cross-lab reproducible lineage.

9.4 CΩMPUTER: Simulation Inside Simulation

Nested universes.

9.5 Cross-Organizational Collaboration

Secure federation with limited permissions.

9.6 Distributed Multi-Agent Reasoning

Agents operating across multiple worlds.

?

10. Security

Federation requires:
- signature verification
- universe policy negotiation
- key-based identity
- trust scoring
- denylist of hostile universes
- provenance validation

No universe should accept foreign truth without proof.

?

11. Why This Matters

Because real computation isn't one universe.
It's an ecosystem of:

- different DAGs
- different policies
- different agents
- different semantic spaces
- different domains
- different teams

Federation is:
- package management
- dependency resolution
- distributed provenance
- inter-world communication
- DAG diplomacy
- computational cosmology

rolled into a single mechanism.

This RFC elevates JIT from "universe" to "multiverse."

?

12. Status & Next Steps

We are ONE RFC away from
the JITOS Architecture Doc.

Possible next RFCs:
- RFC-0022—Visualization & Rendering Protocol (JVP)
- RFC-0023—DAG Schema Evolution (DSE)
- RFC-0024—CΩMPUTER Fusion Layer
- RFC-0025—Distributed Scheduler / Agent Economy
- RFC-0026—Meta-Graph Interchange Format

1.77 CΩMPUTER • JITOS