The Internet of Objects Protocol (Io2p)

Principles and Blueprints

Version 0.7

Table of Contents

1. What is Internet of Objects Protocol (io2p)

Internet of Objects is an open, federated identity and discovery protocol for built-environment objects, enabling trusted data exchange across their full lifecycle.

2. Core Principles

The core principles of io2p constitute the philosophical baseline that makes io2p what it is today but also what it is going to be in the future. Any change in these principles dramatically affects the sustainability of the protocol and the eco-system that it supports.

2.1 Every Object Has a Unique Digital Identity

Each physical object (like a building, material, or component) receives a globally unique identifier (UUID), ensuring authenticity, traceability, and consistency across systems.

2.2 Open Participation for All Stakeholders

Anyone can join a network by running a node. Participation is permissionless for reading data and open to any organization or developer who meets basic technical and quality standards.

2.3 Data is Interoperable and Vendor-Neutral

The Io2p protocol enables efficient data exchange between different software systems (e.g., BIM, ERP) through open standards and APIs, preventing vendor lock-in.

2.4 Agnostic in terms of use cases, data structures and applications

Io2p uses flexible data models that allow any type of real-world object to be digitally identified, described, and linked - regardless of the software, format, or domain. This enables seamless integration across diverse use cases.

2.5 Monetization Flexibility

The Io2p protocol does not have a monetization strategy and does not reward the participants directly. Monetization and access control are traits of the Io2p networks. This guarantees preservation of the openness and neutrality of the core infrastructure. The Io2p protocol allows any stakeholder to build commercial services on top of the protocol without altering its foundational rules.

2.6 Networks

The Io2p protocol does not define a central or a single network of connected nodes. The nodes are independent on their own. The way to discover a node may be a subject of treating a node as part of a network. The protocol provides a locator service which can connect nodes and constitute a sovereign network. An Io2p network may facilitate the interconnectivity of a single or multiple business cases through unique UUIDs.

2.7 Federation and Discovery

A hierarchical network topology of connected nodes within a network is achieved through the Internet's default way of addressing - Domain Name System (DNS), enabling scalable discovery of assets based on UUIDs and internet domain-bound nodes.

2.8 Access

The protocol itself is permissionless. Anyone can spawn a network and attract data sources. Every network has the opportunity to shape the access control to nodes and other resources. This includes the choice to select specific Authentication Authorities (AAs) which are eligible for issuing valid for the specific network certificates.

2.9 Source Code and Concrete Implementations

The Io2p protocol is published and developed as Free and Open Source Software (FOSS) under a permissive and OSI (https://opensource.org/) - recognized open source license (MIT - https://opensource.org/license/mit).
The Io2p protocol can be adopted through direct deployment of the reference implementation code that is located in the Io2p source code set of repositories. The

protocol itself can be implemented by any 3rd party in any programming language and on any computing platform. It can be called and marketed as a compatible lo2p protocol implementation as long as it realizes completely and correctly the specification of the lo2p interfaces and keeps the core principles intact.

2.10 Ways to Contribute to The Source Code

Commercial entities are in position to adopt and extend the protocol with tools, services, source code, or other activities or assets that may reflect the reference source code base. The canonical and preferred way to achieve this is through development of functionalities first on network-level. Starting as an initial fork of the reference implementation, then extending the code, deploying it on a network-level and then submitting merge requests (MR) to the main source code repository. Merge of the proposed code is a subject of community review. The contributors may also announce extensions independently and publish them in the open Internet before MR as a way to get community feedback. Contributions are accepted as long as they address the core protocol mechanics and not domain-specific extensions/modifications.

3. Architectural Blueprint

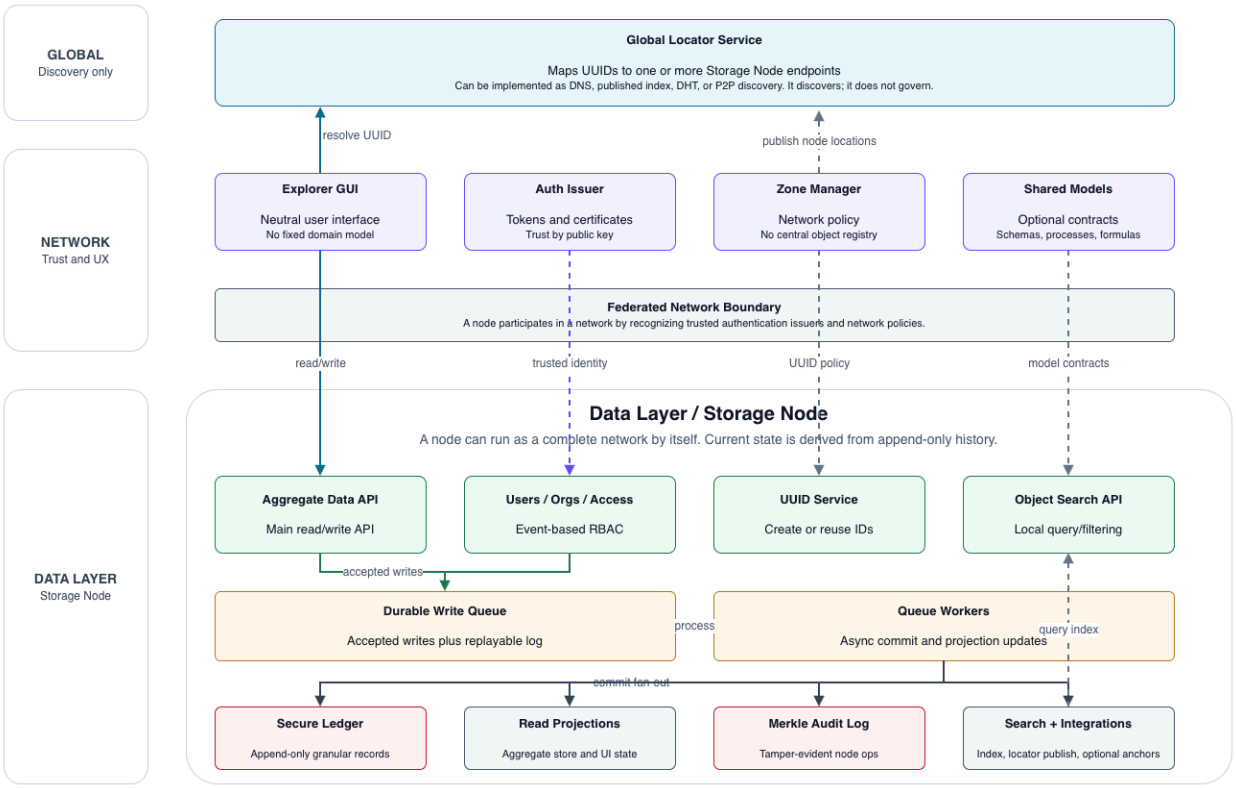

Figure 1.

3.1 Architectural View

The core pattern that Io2p employs is **Append-Only Event Sourcing**. All changes are recorded as immutable, chronologically ordered events in the **Secure Ledger**, never mutating existing data. The current state is dynamically derived for fast reading via **Read Projections**. Writes are accepted synchronously into a **Durable Write Queue** for resilience, and integrity is secured by a mandatory **Merkle Audit Log** for tamper-evident logging of node operations. The architecture strictly separates identity, discovery, data storage, trust, and interpretation into distinct components.Distribution and Sovereignty[1]

Io2p utilizes a **Node-Sovereign Federation** model. Each **Storage Node** is fully autonomous and capable of operating as a complete network alone, with the critical principle that a node must not depend on a global service or central registry to operate locally. **Federation** is achieved through **Public-Key Federation**, where nodes trust one another by accepting approved public keys or certificates from authenticated **Auth Issuers**, rather than relying on a central membership database. **Discovery** is handled by the **Global Locator Service**, which maps UUIDs to candidate Storage Nodes, but its role is strictly discovery and routing; it must not govern object state, ownership, or truth. Dependencies and Execution Semantics[1]

The system operates with a mix of synchronous and asynchronous paths:

- **Synchronous Dependencies (Immediate Acknowledgment):** This path includes UUID lookups, all read requests (**Aggregate Data API**, **Object Search API**), authentication/token issuance, and the initial **Write Acceptance** into the **Durable Write Queue**.[1]
- **Asynchronous Dependencies (Permanent Persistence):** This path handles permanent commitment to the **Secure Ledger**, updates to **Read Projections** and search indexes, **Locator Publication**, and execution of all external integrations (like webhooks or optional blockchain anchoring) via **Queue Workers**.

3.2 Component Details

3.2.1 Global Locator Service

Category	**Details**
Role	The Global Locator Service maps a UUID to one or more candidate Storage Nodes where data for that UUID may exist. It is a discovery mechanism only.
Interface (In)	UUID lookup requests, node publication records, node endpoint metadata, optional node public keys or signatures.
Interface (Out)	Candidate node endpoints, optional metadata about freshness, source node, publication time, and trust hints.
Runtime (Execution)	UUID lookup should be synchronous from the client perspective. Node publication and locator index updates can be asynchronous.
Runtime (Connectivity)	Public, global, or federated endpoint. It must be reachable by clients and nodes that need discovery. It must not be required for a node to function locally.
Important constraint	The locator must not become a source of ownership, truth, permission, or canonical object state.

3.2.2 Explorer GUI

Category	**Details**

Role	The Explorer is a neutral user interface for browsing, searching, modeling, and interacting with Io2p data.
Interface (In)	User actions, UUIDs, search terms, selected nodes, credentials, certificates, model definitions, files, object properties.
Interface (Out)	Calls to node APIs, rendered object views, queued write requests, model interactions, search results, user feedback states.
Runtime (Execution)	Reads feel synchronous. Writes are accepted synchronously into the node queue, then committed asynchronously. The UI must distinguish accepted/queued from permanently committed.
Runtime (Connectivity)	Connects to Storage Node APIs, Auth Issuers, and optionally the Global Locator Service. It should not require direct access to internal node storage.
Important constraint	The Explorer must not impose a domain model such as passport, material, building, process, or ownership by default.

3.2.3 Auth Issuer

Category	**Details**
Role	Auth Issuers authenticate users/organizations and issue signed tokens or certificates that Storage Nodes may trust.
Interface (In)	User credentials, organization credentials, certificate requests, revocation requests, issuer configuration.
Interface (Out)	Signed access tokens, client certificates, public keys, revocation events, issuer metadata.
Runtime (Execution)	Login and token issuance are synchronous. Certificate lifecycle changes and revocations must be represented as append-only events and may propagate asynchronously.

Runtime (Connectivity)	Exposed as a network or organization service. Nodes trust it by public key or certificate chain, not by a central membership database.
Important constraint	A node joins or recognizes a network by accepting trusted issuer keys/certificates.

3.2.4 Zone Manager

Category	**Details**
Role	The Zone Manager manages network-level operational policy: trusted issuers, node participation rules, locator publication behavior, and related configuration.
Interface (In)	Network policy changes, trusted issuer public keys, node declarations, DNS/locator settings, operator decisions.
Interface (Out)	Policy records, trusted issuer lists, locator configuration, node participation metadata.
Runtime (Execution)	Mostly administrative and asynchronous for changes. Runtime policy reads by nodes should be synchronous or locally cached.
Runtime (Connectivity)	Network-scoped management endpoint. Nodes may read or sync network policy, but should retain enough local configuration to operate independently.
Important constraint	It must not act as a central UUID registry or canonical object database.

3.2.5 Shared Models

Shared Models are optional application-level contracts that describe how participants interpret data, including base models, process templates, formulas, schemas, passports, or equivalence rules. They interface with model definitions, schema versions, formulas, and group sharing rules to provide versioned model contracts for applications. While model creation and versioning are asynchronous, lookup during reads can be synchronous. They may be stored locally, shared in a network, or published publicly, but they must always live above the protocol to ensure the core remains model-agnostic.

3.2.6 Data Layer / Storage Node

Category	Details
Role	The Storage Node stores, serves, and audits Io2p data. It can operate as a complete standalone network or participate in a federation.
Interface (In)	API requests, trusted tokens/certificates, write commands, model references, files, object properties, access events.
Interface (Out)	Queued write acknowledgements, ledger records, read projections, search results, audit proofs, locator publications.
Runtime (Execution)	Mixed. Reads are generally synchronous. Writes are accepted synchronously into a durable queue and committed asynchronously.
Runtime (Connectivity)	Exposes node APIs. May connect to trusted Auth Issuers, locator services, and optional external anchors/integrations.
Important constraint	The node must not depend on a global service or central registry to operate locally.

3.2.7 Aggregate Data API

Category	Details
Role	The Aggregate Data API is the main read/write entry point for clients. It hides granular ledger complexity from the UI and applications.
Interface (In)	Read requests, write commands, object data, files, model references, access token/certificate.
Interface (Out)	Read responses, object summaries, queued write acknowledgements, operation IDs.
Runtime (Execution)	Synchronous for reads and write acceptance. Permanent writes happen asynchronously through the queue.

Runtime (Connectivity)	Public or private node API endpoint, depending on node deployment.
Important constraint	It must not mutate existing records directly. Every accepted write becomes a new append-only event.

3.2.8 Users / Orgs / Access

Category	**Details**
Role	Manages node-level users, organizations, certificates, roles, preferences, and authorization decisions.
Interface (In)	Trusted tokens, certificates, login identifiers, role assignment events, revocation events, preference updates.
Interface (Out)	Authorization decisions, user/org projections, access state, audit events.
Runtime (Execution)	Authorization checks are synchronous. User/org/access changes are append-only write events processed through the queue.
Runtime (Connectivity)	Internal node service, exposed through admin/API operations. It may validate tokens/certificates against trusted Auth Issuers.
Important constraint	CRUD must be implemented as versioned events, revocations, and deactivations, not destructive database edits.

3.2.9 UUID Service

Category	**Details**
Role	Creates, accepts, and records UUIDs for protocol objects. It supports both generating new UUIDs and using an existing UUID.
Interface (In)	Create UUID request, reuse UUID request, optional object metadata, caller identity, model context.

Interface (Out)	UUID acceptance/creation event, object identity references, optional locator publication signal.
Runtime (Execution)	UUID acceptance can be synchronous. Permanent recording and locator publication are asynchronous.
Runtime (Connectivity)	Local node service. It may publish UUID-location mappings to the locator through Search + Integrations.
Important constraint	UUIDs have no owner and do not guarantee physical uniqueness or truth.

3.2.10 Object Search API

Category	**Details**
Role	Provides local search across node data, projections, models, and indexed metadata.
Interface (In)	Search queries, UUIDs, filters, source/trust constraints, model filters, pagination.
Interface (Out)	Matching UUIDs, object summaries, facets, references to source records.
Runtime (Execution)	Synchronous read path over derived search indexes and projections.
Runtime (Connectivity)	Node API endpoint. It may be combined with locator queries by clients, but local search is node-scoped.
Important constraint	Search results should identify source and trust context. Search must not imply objective truth.

3.2.11 Durable Write Queue

Category	**Details**
Role	The Durable Write Queue receives all accepted write operations before permanent processing.

Interface (In)	Write commands from Aggregate Data API, UUID Service, access management, model management, or other node services.
Interface (Out)	Ordered jobs, operation IDs, replayable log entries, processing status.
Runtime (Execution)	Synchronous acceptance into durable storage, followed by asynchronous processing by Queue Workers.
Runtime (Connectivity)	Internal node component. It should not be directly exposed to external clients.
Important constraint	If a write is accepted, it must be replayable after interruption or crash.

3.2.12 Queue Workers

Category	**Details**
Role	Queue Workers process accepted jobs, commit records, update projections, update indexes, and emit integration events.
Interface (In)	Jobs from Durable Write Queue, current processing checkpoints, service configuration.
Interface (Out)	Secure ledger records, read projection updates, Merkle audit entries, search index updates, locator publication events, optional external anchors/webhooks.
Runtime (Execution)	Asynchronous.
Runtime (Connectivity)	Internal node component. May use outbound connectivity to locator services, webhooks, or optional blockchain anchors.
Important constraint	Workers must be idempotent or replay-safe so repeated execution does not corrupt state.

3.2.13 Secure Ledger

Category	Details
Role	The Secure Ledger is the primary append-only integrity store for granular protocol records.
Interface (In)	Committed records from Queue Workers.
Interface (Out)	Historical records, verification material, replay source, record references.
Runtime (Execution)	Append-only asynchronous commit. Reads for verification or replay may be synchronous.
Runtime (Connectivity)	Internal storage component, exposed only through node APIs.
Important constraint	It replaces the need for a separate network-level UUID registry in the current architecture.

3.2.14 Read Projections

Category	Details
Role	Read Projections are derived views of current state used for fast reads and UI interaction.
Interface (In)	Secure ledger records, queue worker updates, model interpretation rules.
Interface (Out)	Aggregate object views, current user/access state, object summaries, UI-ready data.
Runtime (Execution)	Asynchronous updates, synchronous reads.
Runtime (Connectivity)	Internal node storage, exposed through Aggregate Data API and Object Search API.
Important constraint	Projections are rebuildable and must not be treated as the source of truth.

3.2.15 Merkle Audit Log

Category	Details
Role	The Merkle Audit Log provides tamper-evident auditability for node-level operations.
Interface (In)	Node operation events, write processing events, access changes, administrative actions.
Interface (Out)	Merkle roots, audit proofs, checkpoints, optional external anchor material.
Runtime (Execution)	Asynchronous append with synchronous proof verification.
Runtime (Connectivity)	Internal node component. Roots may optionally be published or anchored externally.
Important constraint	This is mandatory node-level audit infrastructure, separate from optional blockchain anchoring.

3.2.16 Search + Integrations

Category	Details
Role	Maintains derived indexes and handles outbound integrations such as locator publication, webhooks, optional blockchain anchors, and external notifications.
Interface (In)	Ledger events, projection updates, queue worker events, locator publication rules.
Interface (Out)	Search index updates, locator publication records, webhook calls, optional blockchain anchor transactions.
Runtime (Execution)	Asynchronous.
Runtime (Connectivity)	Internal indexing plus outbound network access to locator services and configured integrations.
Important constraint	External integrations must be derived from committed or accepted events and must not become the source of truth.

3.3 Execution Semantics

Synchronous Paths

- UUID lookup through the Global Locator Service.
- Read requests through Aggregate Data API and Object Search API.
- Authentication/token issuance from Auth Issuer.
- Authorization decisions inside the Storage Node.
- Write acceptance into the Durable Write Queue.

Asynchronous Paths

- Permanent write commit to Secure Ledger.
- Projection rebuild/update.
- Search index update.
- Locator publication.
- Merkle audit log append.
- External webhooks or optional blockchain anchoring.
- Model, access, certificate, and formula lifecycle changes after acceptance.

3.4 Connectivity Principles

- A Storage Node must be able to run without the Global Locator Service.
- The Global Locator Service may discover nodes, but must not govern object state.
- A node federates by trusting issuer public keys/certificates.
- Internal storage components should not be directly exposed to external clients.
- Outbound integrations must be optional and replay-safe.

3.5 Data Integrity Principles

- All writes are append-only.
- Current state is derived from history.
- Every accepted write must be replayable.
- Derived data must be distinguishable from authored data.
- Trust is attached to sources, issuers, signatures, nodes, and shared contracts, not to the protocol itself.

4. Technical Principles

Io2p is append-only, replayable (event sourced), node-sovereign, federated by trust, UUID-addressed, locator-discovered, ledger-secured, source-trust-based, and model-agnostic.

4.1 Append-only Changes

The protocol MUST be append-only.

This core principle dictates that no operation is ever allowed to overwrite or permanently delete existing data within the protocol. Instead, all changes, including updates, corrections, access revocations, or formula changes, must be recorded as new, chronologically ordered records. This creates an immutable history, which is fundamental for ensuring complete auditability and data integrity.

4.2 State of Objects

State MUST be derived, not mutated.

The current, active state of any object is a result of a chain of historical records with the latest record representing the up-to-date state. The entire, unaltered write log serves as the single source of truth for the system. This ensures that the full evolution of an object’s data is preserved and can be verified at any point in time.

4.3 Node-bound Replayable History

Every write MUST be replayable.

All accepted writes must contain sufficient information to allow for their complete reconstruction and processing. Should a node experience a crash or lose its currently derived state, the system must be capable of rebuilding perfect data consistency by replaying the entire accepted write log against the secure ledger. This guarantees high resilience and data integrity across the network.

4.4 Queue-based Storage

Every write MUST pass through a durable queue.

Incoming client requests must first be placed into a persistent queue before final ledger processing occurs. A client may receive an "accepted" status once the write is safely queued, but the definitive "committed" status is only returned after the write has been permanently

processed and secured by the ledger. This mechanism separates immediate acknowledgment from guaranteed persistence.

4.5 Node Sovereignty

A node MUST be able to operate as a complete network alone.

Each individual node in the Io2p ecosystem is designed to be fully sovereign and functional on its own, with federation being an optional capability. This architecture prevents dependency on any single point of failure, such as a central registry, global authority, or mandatory global service, ensuring local operational autonomy.

4.6 Public-Key Federation

Federation MUST be trust-by-public-key, not membership-by-database.

Node federation is achieved through cryptographic trust, meaning a node recognizes and joins a network by explicitly accepting approved authentication issuers, digital certificates, or specific public keys. This decentralized method avoids the need for a centralized database of members, instead basing trust on verifiable cryptographic credentials.

4.7 UUID Role

UUIDs MUST identify protocol objects, not truth or ownership.

The Universally Unique Identifiers (UUIDs) serve strictly as opaque handles for protocol objects. A UUID does not inherently confer proof of data correctness, global uniqueness, or legal ownership of the physical object it represents. These identifiers are simply a standardized way to reference an object within the system.

4.8 Explicit Equivalence

Object equivalence MUST be explicit.

The system operates under the assumption that two UUIDs, even if referencing similar objects on different nodes, are distinct until proven otherwise. A relationship must be explicitly asserted by a user, an application, or a domain model to declare that two or more UUIDs correspond to the same physical object. This prevents accidental merging of identities and requires clear, auditable linking.

4.9 Data Separation

The protocol MUST separate identity, location, and data.

The architecture distinctly separates the three fundamental components for an object. The UUID is dedicated to answering “which object” is being referenced, the locator service determines “where it is” located in multiple networks (not necessarily single result), and the specific node is responsible for answering “what data is available” for that object. This modular separation enhances flexibility and scalability.

4.10 Locator Function

The locator MUST discover; it MUST NOT govern.

While global locator services are essential for mapping UUIDs to their serving nodes, their role is purely one of discovery and routing. These services are strictly forbidden from acting as a source of authority for ownership, truth, permissions, or canonical state for any object. This ensures that core governance remains with the sovereign nodes.

4.11 Primary Integrity Layer

Secure ledger MUST be the primary integrity layer.

The core protocol relies on a secure ledger to maintain data integrity and verification as its primary layer of trust. Although anchoring data to an external blockchain may be an option, the correctness and functionality of the Io2p protocol must remain independent and not rely on any external blockchain system.

4.12 Cryptographic Log

Every node MUST keep a cryptographic operation log.

Each sovereign node is required to maintain a comprehensive and tamper-evident log of all its internal operations. This log should be auditable through structures like Merkle trees or similar cryptographic techniques, allowing external verification of node-level actions and ensuring that no internal data tampering has occurred.

4.13 Model Agnosticism

The protocol MUST remain model-agnostic.

The core protocol must be designed without hardcoding any specific domain models, such as those related to buildings, materials, ownership, or ESG standards, into its foundational layer. This agnostic approach ensures that Io2p can be flexibly applied across a vast range of sectors and use cases.

4.14 Application-Level Models

Models MUST live above the protocol.

Shared data structures, base models, digital passports, and complex processes are considered application-level agreements and contracts. These domain-specific models are built and managed on top of the generic protocol layer, allowing users or networks to define and share schemas without altering the core infrastructure.

4.15 Source-Based Trust

Trust MUST be source-based.

The protocol's function is to cryptographically prove data provenance and integrity—showing *who* published data and verifying that it hasn't been altered. It does not certify the objective truthfulness of the data itself. Trust is therefore applied to the verifiable source, such as specific nodes, issuers, organizations, or users who sign the claims.

4.16 Event-Based Access Control

Access control MUST be event-based and auditable.

All changes related to access, including user permissions, certificate updates, role changes, and revocations, must be treated as historical events and recorded in the append-only log. This approach ensures that access control modifications are fully auditable and visible, rather than being managed through hidden, mutable database edits.

4.17 Data Distinction

Derived data MUST be distinguishable from authored data.

Any calculated or generated properties, such as data rollups, formula results, composite hashes, or thumbnails, must be clearly and technically marked as derived information. This differentiation is critical for users to understand what data was manually authored versus what was computed by the system.

4.18 Immutable Formulas

Formulas MUST be versioned as immutable objects.

Formulas used for calculations must be treated as immutable data objects and cannot be edited in place once created. To make a change, a new version must be copied and saved, ensuring that previous calculations remain fully auditable and verifiable against the formula version used at that time.

4.19 Explorer Neutrality

The Explorer MUST stay neutral.

The Io2p Explorer, a viewing tool, must maintain a neutral perspective and not implicitly force any single canonical object model or hierarchy onto the data. Its role is to enable inspection, search, modeling, and visualization, offering flexible ways to interpret the data without enforcing specific passports or processes.

4.20 Historical Readability

Protocol evolution MUST preserve historical readability.

As new versions of the protocol are released, they are permitted to extend system behavior but must not compromise the integrity of legacy data. A critical requirement is that older records must remain fully interpretable, verifiable, and replayable by any compliant node, ensuring long-term data sustainability.