TECHNICAL WHITEPAPER · v0.1 DRAFT · FOR OS BUILDERS

Mastermind City Operating System

Architecture, Implementation Options & Builder Reference

This document is a technical specification and architectural reference for developers building the Mastermind City Node Operating System. It describes the system's required behaviours, data models, and integration contracts, then presents competing implementation architectures for each major subsystem. Where multiple credible approaches exist, the tradeoffs are analysed and a recommended path is identified — but alternatives are fully documented to give builders agency over their implementation choices. This is a working document. Sections marked [OPEN] indicate design decisions not yet finalised.

Document Status	Draft v0.1 — For Builder Review
Project	Mastermind City
Audience	OS Builders, Smart Contract Developers, Protocol Architects
Repository	github.com/Jabroni777/mastermind-city
Branch	concept-build
License	[OPEN] — See Section 9

"The OS must be light enough that a node of five people running a monastery and a node of five people running a software studio can both use it — without modification."

— Core Design Constraint, Mastermind City

TABLE OF CONTENTS

SECTION 01

System Overview & Design Philosophy

What the OS Is

The Mastermind City Node OS is a lightweight, offline-capable application that provides structural scaffolding for small autonomous communities (nodes). It manages the operating rhythm of a node — its monthly cycle, role assignments, decision records, and contribution ledger — without prescribing what that community does or how it lives. The OS is infrastructure, not ideology.

It must be deployable on modest hardware, function completely without internet access, and produce cryptographically signed records that can optionally be shared with a federation layer. When federation connectivity exists, records replicate to the federation storage network and contribution proofs are anchored on-chain. When it does not, the OS keeps running. Connectivity is a feature, not a requirement.

Non-Negotiable Design Constraints

Offline-First

All core functionality — cycle management, decision recording, role rotation, contribution tracking — must work with zero network connectivity. Federation features are additive.

Minimal Footprint

The application must run on a Raspberry Pi 4 (4GB RAM) or equivalent modest hardware. Binary size and runtime memory are first-class concerns.

Local Data Sovereignty

All node data is stored on the node's own hardware. No telemetry, no cloud sync, no external dependency for core function.

Cryptographic Integrity

Every formal record produced by the OS is signed by the node's keypair. Records are verifiable without trusting the OS itself.

Cultural Agnosticism

The OS must make zero assumptions about the community's culture, beliefs, economics, or purpose. Role names are configurable. Cycle emphasis is configurable. The underlying protocol is invariant.

Graceful Federation

When a node connects to a federation, the OS should handle synchronisation, record replication, and on-chain anchoring without requiring technical expertise from node members.

Exit Without Loss

A node must be able to exit the federation at any time without losing its records, identity, or history.

What the OS Does Not Do

The following are explicitly out of scope for the OS layer. They belong either to the federation layer, companion projects, or the community itself:

— Prescribe culture, values, religion, or economic model
— Govern what a community produces or how members live
— Replace human judgment in any decision
— Operate as a social network or communication platform
— Manage The Manual — that is a separate companion project
— Handle legal entity formation or tax compliance
— Store private personal data of members

The Core Operating Loop

The OS runs a 30-day cycle. Each cycle has four phases: Opening (days 1–3), Planning (days 4–7), Build (days 8–21), and Close (days 22–30). The OS surfaces prompts and collects structured input at phase transitions. Between phase transitions, it is silent unless a member raises a tension or logs a contribution.

Every 90 days, the OS initiates a role rotation. Roles are: Navigator, Steward, Chronicler, Connector, and Builder(s). Role rotation is invariant — it cannot be disabled or overridden in the base OS. This is the single most important structural enforcement the OS provides.

SECTION 02

Core Data Models

The following data models are canonical. Implementations may add fields but must not remove or rename required fields, as records must be interoperable across different OS builds within the same federation.

Node Identity Record

> **REQUIRED FIELDS**

```
// NodeIdentity — generated at genesis, immutable thereafter
NodeIdentity {
node_id: String, // human-readable name, e.g. "cedar-7"
did: String, // DID:key or DID:web identifier
public_key: [u8; 32], // Ed25519 public key
genesis_date: ISO8601, // date of node formation
node_type: NodeTypeEnum, // homestead|studio|guild|monastery|lab|agora|enterprise|custom
charter_hash: [u8; 32], // SHA-256 of the signed charter document
federation_ids: Vec, // optional: federation(s) this node belongs to
version: SemVer, // OS version at genesis
}
```

Cycle Record

The cycle record is the primary output of the OS. It is generated at the end of each 30-day cycle, signed by the Navigator, and optionally anchored to the federation chain.

```
// CycleRecord — produced at Close phase, immutable once signed
CycleRecord {
node_id: String,
cycle_number: u32,
period: ISO8601Range, // e.g. "2025-11-01/2025-11-30"
phase_logs: Vec, // one entry per phase
decisions: Vec, // all formal decisions this cycle
role_assignments: RoleMap, // member_id -> role for this cycle
contribution_totals: HashMap,
tensions_raised: Vec, // logged, resolved or unresolved
rotation_due: bool, // true if day 90+ since last rotation
navigator_signature: Signature, // Ed25519 sig over record hash
created_at: ISO8601,
ipfs_cid: Option, // populated if federated
chain_tx: Option, // populated if anchored on-chain
}
```

Decision Record

```
Decision {
id: String, // "d{cycle}-{seq}", e.g. "d14-003"
decision_type: DecisionType, // consent | tension_resolved | role_change | charter_amendment
summary: String, // human-readable, max 280 chars
proposer_id: MemberId,
objections: Vec, // empty if passed cleanly
counter_proposals: Vec, // required if any objection raised
result: DecisionResult, // passed | withdrawn | deferred
timestamp: ISO8601,
assigned_to: Option,
due_date: Option,
}
```

SECTION 03

Subsystem A: Node OS Application Layer

The application layer is the shell that node members interact with. Three credible implementation architectures exist. Each presents different tradeoffs across performance, developer accessibility, offline capability, and binary size.

Option A1 — Tauri (Rust + Web Frontend)

Tauri is a framework that packages a Rust backend with a web-based frontend (HTML/CSS/JS or any web framework) into a native desktop application. The Rust core handles cryptography, local storage, P2P networking, and system operations. The web frontend handles all UI. Communication between them happens via Tauri's command/event bridge.

Factor	Assessment
Binary size	3–8 MB (vs Electron's 80–150 MB)
Memory footprint	~30–80 MB idle (vs Electron's 200–400 MB)
Offline capability	Full — no network dependency by design
Crypto / P2P	Native Rust — best-in-class performance and library ecosystem
Developer accessibility	Requires Rust knowledge for backend; frontend is standard web
Platform support	macOS, Windows, Linux, iOS, Android (Tauri v2)
Hardware floor	Raspberry Pi 4 (ARM64) — confirmed viable
Complexity	Higher — two languages, IPC boundary to manage

> **RECOMMENDED FOR RUST BUILDERS**
> Tauri is the recommended architecture for the production OS. The performance and footprint advantages are significant for a tool expected to run on low-power hardware in off-grid environments. The Rust ecosystem for libp2p and cryptography is mature. The primary cost is developer accessibility — the team needs Rust competency.

Option A2 — Electron (Node.js + Web Frontend)

Electron bundles Chromium and Node.js into a native desktop app. It is the most widely used cross-platform desktop framework and has the largest developer ecosystem. VS Code, Slack, and Figma use it. The tradeoffs are well-understood.

Factor	Assessment
Binary size	80–150 MB — significant for low-bandwidth environments
Memory footprint	200–400 MB idle — likely too heavy for Raspberry Pi
Crypto / P2P	Via Node.js libraries — functional but slower than Rust

Developer accessibility	High — JavaScript/TypeScript ecosystem
Offline capability	Full
Hardware floor	Struggles on Raspberry Pi; fine on modern hardware
Complexity	Lower — single language, large community

> **VIABLE FALLBACK**
> Electron is a reasonable choice if the development team lacks Rust expertise and the initial deployment targets are not Raspberry Pi-class hardware. A migration path to Tauri should be planned from day one — the web frontend code is fully portable between the two frameworks.

Option A3 — Progressive Web App (PWA)

A PWA runs in the browser but can be installed as a home-screen app, use service workers for offline capability, and access local storage. No native binary required. Lowest possible barrier to entry for new nodes.

> **NOT RECOMMENDED FOR CORE OS**
> PWA is appropriate for the demo/prototype site but not for the production OS. The limitations are significant: no access to system keychain for secure key storage, severely limited persistent storage, no ability to run libp2p daemon processes, and browser sandbox restrictions that conflict with P2P networking requirements. PWA may be appropriate as a companion interface for federation dashboards.

A: Recommendation Summary

Primary: Tauri (Rust + Astro/HTML frontend). Build the backend in Rust using Tauri v2, expose commands to the frontend via Tauri's IPC bridge. Frontend in Astro for the static pages + vanilla JS for interactive components. This gives the best long-term foundation.

Fallback: Electron with a planned Tauri migration. If the initial builder team is JS-first and timeline is critical, start in Electron. Keep all business logic in a separate module layer (not tied to Node.js APIs) so the migration surface is minimised.

SECTION 04

Subsystem B: Local Storage & State Management

The OS must persist cycle records, decisions, member data, and contribution ledgers locally without any network dependency. Three architectures are credible.

Option B1 — SQLite (Embedded)

SQLite is the most battle-tested embedded database on earth. It runs inside the application process, stores data in a single file, requires zero configuration, and has bindings in every relevant language. The Rust ecosystem has excellent SQLite support via sqlx or rusqlite.

Strengths:

+ Zero configuration, zero server
+ Single file — trivially backed up or migrated
+ ACID compliant — no data corruption on crash
+ Rust: sqlx or rusqlite, both mature
+ Well-understood query model for contribution analytics

Considerations:

— Not designed for multi-writer concurrent access (single node = fine)

— Schema migrations require careful versioning as the OS evolves

Option B2 — LMDB (Memory-Mapped B-Tree)

LMDB (Lightning Memory-Mapped Database) is an embedded key-value store used in high-performance systems. Faster than SQLite for write-heavy workloads. Used in Bitcoin Core and many blockchain systems.

> **NOTE**
> LMDB's performance advantage is not relevant at Mastermind City's data volumes. A node produces perhaps 500 records per year. SQLite is more than sufficient and significantly simpler to work with for structured relational data.

Option B3 — Local-First CRDT (cr-sqlite or Automerge)

CRDTs (Conflict-Free Replicated Data Types) are data structures designed for distributed systems where multiple writers may update the same data concurrently and need to merge without conflict. cr-sqlite is a CRDT extension for SQLite. Automerge is a document-oriented CRDT used in collaborative editors.

The appeal: if two OS instances ever need to merge their state (e.g., two devices used by the same node, or federation record sync), CRDTs provide automatic conflict resolution. The cost: significantly higher complexity, less mature tooling, and tricky semantics around ordered data like decision logs.

B: Recommendation

> **RECOMMENDED**
>
> SQLite via sqlx (Rust async) for Phase 1. Schema: separate tables for cycles, decisions, members, roles, contributions, and tensions. Migrations managed with sqlx migrate. Revisit cr-sqlite in Phase 2 when federation sync requirements are better understood — it may be the right answer for the federation record replication layer specifically.

SECTION 05

Subsystem C: P2P Networking & Inter-Node Communication

Inter-node communication is a Phase 2 feature — individual nodes in Phase 1 operate in isolation. However, the networking architecture must be selected early because it shapes the identity and cryptography layers that are needed from day one. Three architectures are credible.

Option C1 — libp2p (rust-libp2p)

libp2p is a modular networking stack built for peer-to-peer systems. It is the networking layer beneath IPFS, Ethereum, Filecoin, and Polkadot. The Rust implementation (rust-libp2p) is production-grade.

Capability	Status
Peer discovery (mDNS, DHT, bootstrap nodes)	`Built-in`
NAT traversal (hole punching, relay)	`Built-in`
Pub/Sub messaging (gossipsub)	`Built-in`
Transport encryption (Noise protocol)	`Built-in`
Private network support (pnet)	Built-in — key to federation isolation
Stream multiplexing (yamux)	`Built-in`
Language	Rust (primary), Go, JS, Python available

The pnet (private network) module is particularly important: it allows the federation to run an isolated P2P network using a shared swarm key. Nodes outside the federation cannot participate. This is the correct model for Mastermind City.

Option C2 — Nostr Protocol

Nostr is a simple, open protocol for censorship-resistant communication. Each user has an Ed25519 keypair. Messages are signed events broadcast to relays. No central server. Clients can run their own relays.

Nostr is not a full P2P network — it is a message-passing protocol with a relay model. It is much simpler than libp2p and already has production client libraries. Its keypair model is directly compatible with Mastermind City's identity requirements.

> **IMPORTANT ARCHITECTURAL NOTE**
> Nostr and libp2p are not competing alternatives — they can be layered. Nostr handles high-level messaging (inter-node communications, federation announcements, record distribution metadata). libp2p handles low-level transport, peer discovery, and the private swarm. This hybrid is the recommended approach.

Option C3 — Hypercore / Holepunch

Hypercore is a secure append-only log protocol. Holepunch builds on it to provide full P2P application capabilities including DHT-based peer discovery and hole punching. Used in Keet (a P2P video chat app). Strong for append-only record structures — which cycle records essentially are.

The limitation: the primary ecosystem is JavaScript/Node.js. The Rust bindings are experimental. If the OS is built in Rust (Tauri), Hypercore integration is awkward.

C: Recommendation

> **RECOMMENDED: HYBRID**
> Phase 1: No P2P required. Nodes operate in isolation. Identity keypairs generated using Ed25519 (compatible with both libp2p and Nostr). Phase 2: rust-libp2p for the private federation swarm (transport, peer discovery, private network isolation). Nostr protocol for higher-level messaging between nodes — inter-node proposals, record distribution, federation announcements. Federation operators run lightweight Nostr relays on their infrastructure.

SECTION 06

Subsystem D: Federation Storage

Federation storage is how cycle records, charter documents, and contribution proofs are preserved across the network such that no single node failure can destroy them. This section addresses a critical architectural clarification: the Mastermind City federation must NOT depend on the public IPFS network or any external storage service.

> **CRITICAL DESIGN PRINCIPLE**
> The federation's storage network IS its member nodes. Records replicate across the nodes that are members of the federation. No external pinning service, no Filecoin, no Pinata. The federation is self-contained. The public IPFS network is irrelevant to this architecture — only the content-addressing protocol is borrowed.

Option D1 — Private IPFS Swarm (Recommended)

IPFS supports private swarms via a "swarm key" — a shared secret that all nodes in the swarm must possess to communicate with each other. Nodes outside the federation swarm key cannot join the network, cannot discover its nodes, and cannot access its content. The content-addressing protocol (CIDs) still works internally, giving all the benefits of IPFS without public network exposure.

How it works:

Federation genesis:
The federation generates a swarm key (32 random bytes). This key is distributed to all member nodes via a secure channel during node admission.

Node runs IPFS daemon:
Each federated node runs a go-ipfs or iroh daemon configured with the swarm key. It will only connect to other nodes presenting the same key.

Record pinning:
When a node finalises a cycle record, it adds the document to its local IPFS node. The record's CID (content identifier) is broadcast to the swarm via the Nostr/libp2p messaging layer.

Automatic replication:
Other federation nodes receive the CID announcement and pin the record locally. The record is now held by every online federation node.

Minimum replication factor:
A cycle record is not considered durably stored until it has been pinned by at least N federation nodes (recommended N=5, configurable by federation DAO vote). The OS reports replication status to the Chronicler.

Disaster recovery:
If a node's local hardware is destroyed, its records exist on all other federation nodes. On OS reinstallation, the node presents its keypair and re-syncs from the swarm.

Option D2 — Syncthing

Syncthing is a mature, battle-tested P2P file synchronisation tool. It is simpler than IPFS, has excellent documentation, and works reliably over local networks with zero internet dependency. It uses TLS with self-signed certificates for transport encryption and device ID whitelisting for network isolation.

Factor	IPFS Private Swarm	Syncthing
Complexity	Higher	Lower
Content addressing	Yes (CIDs — links to on-chain)	No (path-based)
On-chain integration	Native	Requires adapter layer
Local network only	Supported	Native
Internet optional	Yes	Yes
Operational maturity	High	Very high
Rust library support	iroh (good)	No native Rust library

> **SYNCTHING AS FALLBACK**
> Syncthing is a valid fallback for federations that want simpler operations and do not need on-chain provenance for their records. It lacks content-addressing, which means the on-chain hash anchoring must use a separate hash of the file at time of creation — workable but less elegant. Recommend for pilot federations to validate the concept before committing to IPFS operational complexity.

Option D3 — Hypercore Protocol

Hypercore's append-only log structure is a natural fit for cycle records — each cycle appends to the node's log, and the log is cryptographically signed and verifiable. Beaker Browser demonstrated this model for decentralised publishing. The limitation remains the JavaScript-first ecosystem, which is awkward for a Rust/Tauri primary build.

Why Not Public IPFS or Filecoin

For completeness: public IPFS without pinning is not reliable storage — content disappears when no node is pinning it. Filecoin is a decentralised storage market where you pay storage providers to persist your data. Both introduce external dependencies and potential censorship surfaces that are incompatible with Mastermind City's sovereignty requirements. Do not use either for the federation storage layer.

D: Recommendation

RECOMMENDED

Phase 1 (Pilot): Syncthing for simplicity. Run a shared Syncthing folder across all pilot nodes. Record hashes generated locally and stored in the cycle record. Phase 2: Migrate to private IPFS swarm using iroh (Rust IPFS implementation by n0). iroh is production-ready, Rust-native, and designed for exactly this use case. Minimum replication factor: 5 nodes before cycle close is confirmed. On-chain anchoring uses the IPFS CID as the canonical content reference.

SECTION 07

Subsystem E: Blockchain & DAO Layer

The blockchain layer is Phase 2–3 infrastructure. It is not required for individual nodes to operate. It becomes relevant when nodes federate and want cryptographic proof of their records, shared treasury governance, and contribution-weighted decision-making that no single party controls.

Chain Selection

Three EVM-compatible chains are credible for Mastermind City's requirements. The requirements are: low transaction costs (records are anchored frequently), stability and longevity, open-source tooling, and ideological alignment with decentralisation.

Chain	Tx Cost	Throughput	Decentralisation	Tooling	Verdict
Ethereum L1	High ($2–$50)	15 TPS	Highest	Best	Too expensive
Base (L2)	Low (<$0.01)	1000+ TPS	High (via ETH)	Excellent	Recommended
Optimism (L2)	Low (<$0.01)	1000+ TPS	High (via ETH)	Excellent	Strong alt
Gnosis Chain	Very low	70 TPS	High	Good	Good alt
Solana	Very low	65k TPS	Moderate	Good	Ideological mismatch

Recommendation: Base (L2). Base is an Ethereum L2 operated by Coinbase, using the OP Stack (same as Optimism). Transaction costs are sub-cent. It inherits Ethereum's security. The EVM tooling (Hardhat, Foundry, OpenZeppelin) works unchanged. It has significant institutional backing without compromising decentralisation at the protocol level.

Smart Contract Architecture

Four contracts are required for Phase 2–3:

NodeRegistry.sol

Stores node identities (DID + public key + metadata). Called once per node at genesis. Emits NodeRegistered event. Nodes can update metadata but not their identity keypair.

RecordAnchor.sol

Accepts a (node_id, cycle_number, ipfs_cid, record_hash) tuple and writes it to an immutable log. Called by the Chronicler at each cycle close. Anyone can verify a record against this contract.

FederationTreasury.sol

Multi-sig treasury with DAO-controlled distribution rules. Built on OpenZeppelin's TimelockController + Governor. Receives contributions, distributes rewards on verified cycle completion, manages capital access programs.

GovernanceToken.sol

Non-transferable (soulbound) ERC-20 variant representing governance weight. Minted by the treasury contract on verified contribution proof. Burned on node exit. Not tradeable — purely for governance weight.

Option E1 — Full On-Chain Governance (Base L2)

All federation governance happens on-chain via OpenZeppelin Governor. Proposals, votes, and execution are recorded on Base. Treasury distributions are automatic smart contract executions triggered by contribution proofs.

Pros: Maximum transparency and auditability. No trusted intermediary. Rules cannot be changed unilaterally. **Cons:** Every governance action costs gas. Slower — on-chain voting has mandatory timelock periods. Requires members to hold crypto wallets.

Option E2 — Minimal On-Chain + Off-Chain Governance

Only record anchoring and treasury custody happen on-chain. Governance votes are conducted off-chain via Snapshot (signed messages, no gas cost) and executed by a small multisig group of trusted federation nodes. This is the Snapshot + Safe{Wallet} model used by most real-world DAOs today.

Pros: Gasless voting — much better user experience. Faster decisions. No crypto wallet required for most members. **Cons:** The multisig signers are a trusted party. Not fully trustless. Snapshot votes are advisory, not binding code.

E: Recommendation

> **RECOMMENDED: PHASED APPROACH**
>
> Phase 2: Option E2 (Minimal on-chain). Use Snapshot for governance votes and a 3-of-5 Safe multisig for treasury execution. This is operationally realistic for a young federation. RecordAnchor.sol and NodeRegistry.sol are live from day one. Phase 3: Migrate treasury to full Governor-based on-chain governance as the federation matures and members become comfortable with on-chain participation. The GovernanceToken (soulbound, non-transferable) is the key design decision that prevents the DAO from becoming speculative — implement this correctly from the start.

SECTION 08

Integration Architecture & Reference Implementation

The following describes the recommended integrated architecture — the combination of the options above that best satisfies all design constraints simultaneously.

Phase 1: Sovereign Node (No Federation)

Component	Implementation
Application Shell	Tauri v2 (Rust backend + Astro/HTML frontend)
Local Storage	SQLite via sqlx — cycle records, decisions, members, contributions
Cryptography	Ed25519 keypair (ring or ed25519-dalek crate) — generated at genesis
Record Signing	Node keypair signs SHA-256 hash of each cycle record JSON
Backup	Manual export to encrypted file. No network required.
Networking	None. Fully offline.
Blockchain	None. Keypair compatible with future on-chain registration.

Phase 2: Federated Node

Component	Implementation
Application Shell	Tauri v2 (unchanged from Phase 1)
Local Storage	SQLite (unchanged) + iroh IPFS node for distributed storage
Federation Transport	rust-libp2p with pnet private swarm key
Messaging	Nostr protocol over libp2p transport — inter-node events and announcements
Record Storage	Private IPFS swarm (iroh) — records pinned across federation nodes
Replication Requirement	Min 5 pins before cycle close confirmed. Chronicler sees status.
On-Chain Anchoring	RecordAnchor.sol on Base L2 — CID + hash written at cycle close
Node Registry	NodeRegistry.sol — node DID registered on first federation join
Federation Governance	Snapshot (off-chain votes) + Safe 3-of-5 multisig for execution

Critical Implementation Notes

Ed25519 keys are foundational

Generate Ed25519 keypairs from day one. This key IS the node's identity — it signs records, registers on-chain, and authenticates in the P2P network. Key generation, secure storage (OS keychain or encrypted file), and backup are the most security-critical operations in the entire system.

Never store private keys in the database

SQLite records should contain only public keys and signatures. Private keys go in the OS keychain (via Tauri's secure storage API) or an encrypted keyfile with a user-supplied passphrase.

Records are append-only

The cycle record log is conceptually append-only. A signed record should never be mutated. Corrections are new records that reference the record they amend. This preserves the integrity of the history.

The soulbound governance token

Design the GovernanceToken.sol as non-transferable from the start. Transferability is what turns DAO governance tokens into speculation vehicles. Override all ERC-20 transfer functions to revert. This is a single design decision that prevents a class of failure modes.

Federation exit must be clean

A node exit protocol must: (1) settle outstanding contribution claims, (2) revoke the node's governance weight, (3) retain all records (records belong to the node, not the federation), (4) remove the node from the active swarm key rotation schedule. Design this before you need it.

SECTION 09

Open Questions & Build Priorities

The following questions are unresolved and require decisions before or during Phase 2 implementation. They are listed in rough priority order.

[OPEN-01] Open-Source Licence

Which licence best serves the project's values? AGPL-3.0 prevents proprietary forks and requires that any hosted version publish its source. MIT maximises adoption but permits capture. The Commons Clause addendum to MIT/Apache is a middle path used by some projects. Decision needed before first public release.

[OPEN-02] Swarm Key Distribution

How is the federation swarm key distributed to new member nodes securely? Options: (a) manual out-of-band (simple, insecure), (b) encrypted via the admitting node's public key (better), (c) threshold secret sharing requiring M-of-N existing nodes to approve distribution (most secure, most complex). This is an attack surface.

[OPEN-03] Contribution Unit Definition

The contribution ledger tracks "units" — but what is a unit? Hours? Deliverables? Capital? A hybrid? Different node types will weight these differently. The OS needs a pluggable contribution schema — the federation sets the base weights, nodes can add local categories. This must be designed before the treasury distribution logic.

[OPEN-04] Offline Role Rotation Enforcement

Role rotation is invariant at 90 days. But what happens if a node has been offline and reconnects after day 93? What if the Navigator refuses to facilitate rotation? The OS must have a clear policy for enforcing rotation without a human override path.

[OPEN-05] Legal Entity for Federation DAO

The federation DAO needs a legal wrapper for real-world contracts, tax, and liability. Candidates: Wyoming DAO LLC (US-accessible, statutory DAO recognition), Swiss Foundation (neutral, established for international orgs), Marshall Islands DAO LLC (purpose-built for DAOs, no income tax). Legal counsel required.

[OPEN-06] Smart Contract Audit

The treasury and governance contracts are security-critical. An audit is required before Phase 3 launch. Budget: $30,000–$80,000 for a reputable auditor (Trail of Bits, OpenZeppelin, Certora). Funding source for the audit needs to be identified early.

[OPEN-07] The Manual Interface Spec

The Chronicler role is the interface between the OS and The Manual (companion project). What does the OS produce that The Manual uses? A structured export format? A ceremony prompt? This interface needs to be agreed between the two projects before either builds their side of it.

SECTION 10

Appendix: Key Dependencies & Licences

Key libraries and their licences at time of writing. Verify current licence status before production use.

Library	Language	Purpose	Licence
tauri	Rust	Application shell	MIT / Apache-2.0
sqlx	Rust	Async SQLite	MIT / Apache-2.0
rust-libp2p	Rust	P2P networking	MIT
iroh	Rust	IPFS (n0)	MIT / Apache-2.0
ed25519-dalek	Rust	Cryptographic signatures	BSD-3
ring	Rust	Crypto primitives	ISC/OpenSSL
nostr-sdk	Rust	Nostr protocol	MIT
OpenZeppelin Contracts	Solidity	DAO / token frameworks	MIT
Hardhat / Foundry	JS / Rust	Smart contract toolchain	MIT
Astro	JS	Frontend framework	MIT
Syncthing	Go	P2P file sync (fallback)	MPL-2.0
go-ipfs / kubo	Go	IPFS node (alt to iroh)	MIT / Apache-2.0

This document is a living specification. As pilot nodes provide feedback and implementation decisions are made, it will be updated. Comments, corrections, and alternative architecture proposals should be filed as issues on the GitHub repository: github.com/Jabroni777/mastermind-city

Mastermind City · Technical Whitepaper v0.1 · Draft