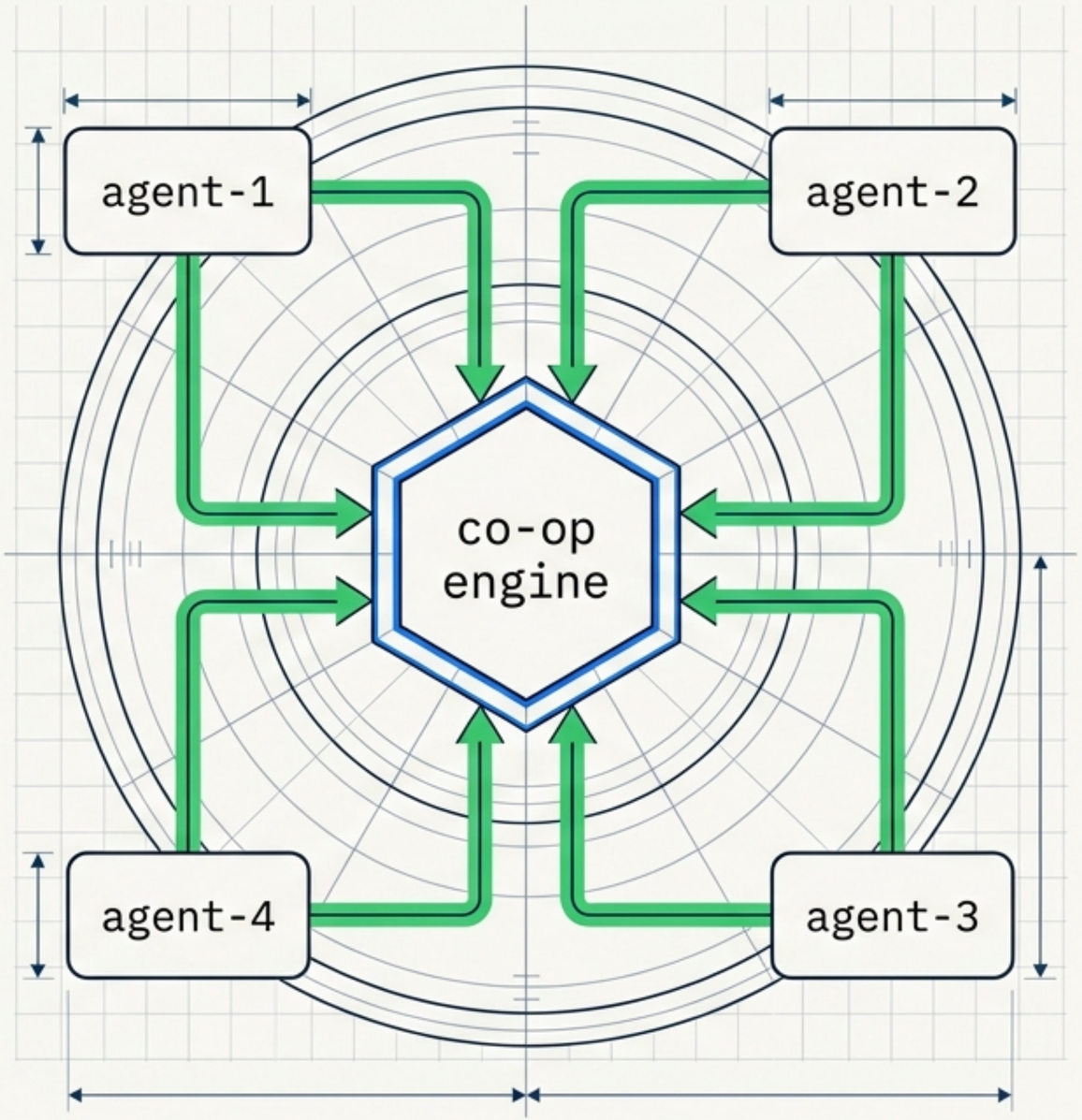
co–op: The Invisible Switchboard for Multi-Agent Coding

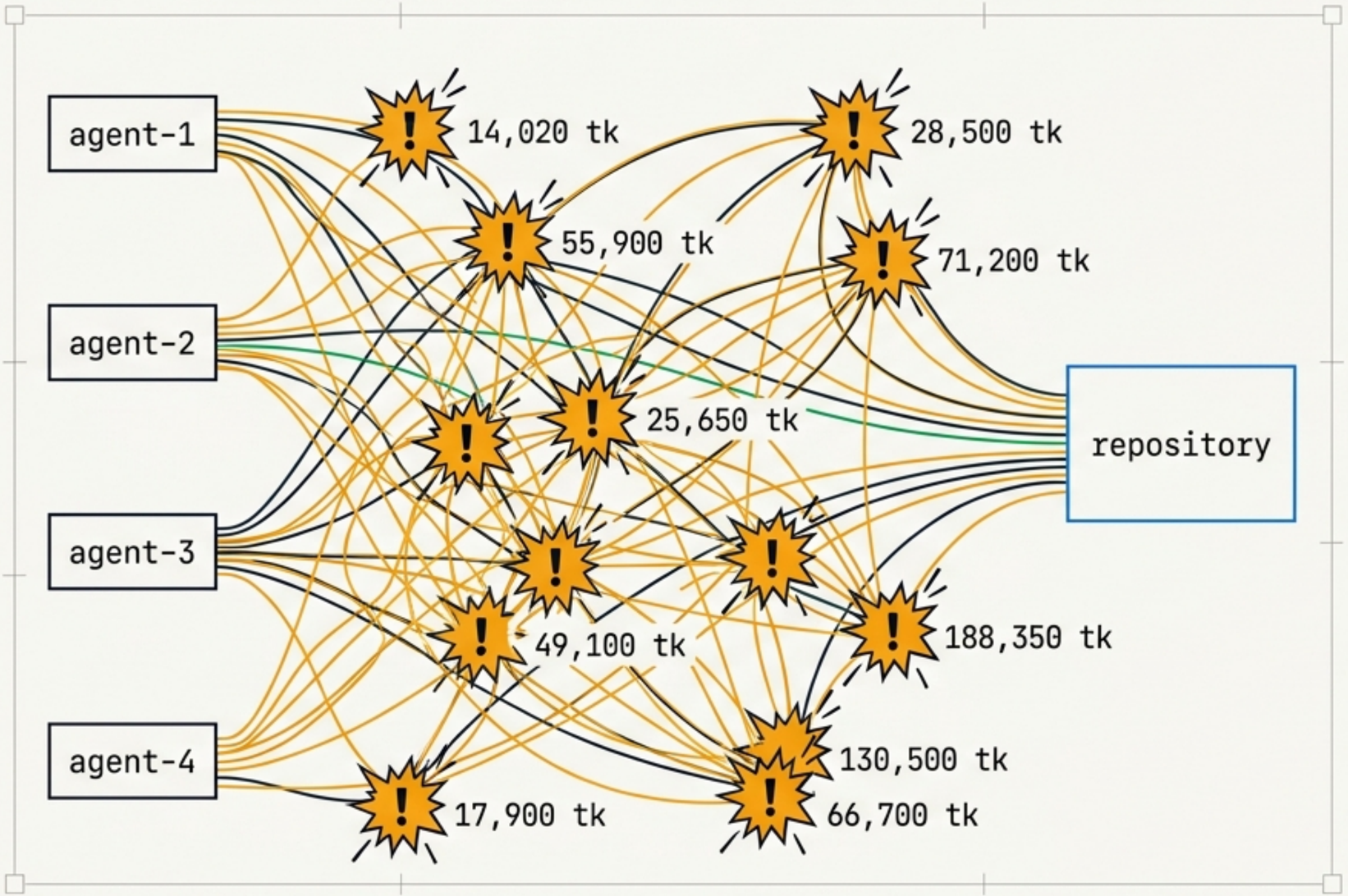
A portable, low-token coordination primitive for AI coding harnesses.

The Paradigm Shift: Agents do not inspect files, grep logs, or mutate coordination state. They call /co-op, and a Python engine handles the rest.

- **Target Goal:** Less than 5% token overhead.
- **Primary Interfaces:** `/co-op` slash command + `co-op` Python CLI.



The Multi-Agent Collision Problem



The Cost of Unmanaged Agents

Duplicate Work: Agents re-solving the same problems and making repeated decisions.

State Corruption: File edit collisions across independent agent instances.

Context Hemorrhaging: Poor handoffs when agents hit context limits or fail.

Token Bloat: Wasting massive context windows searching transcripts, grepping logs, and passing huge status objects.

Friction: High manual operator burden to start, join, or pause sessions.

The Missing Primitive: A Temporary Coordination Layer

What it IS

- ✓ A Python-mediated, SQLite-indexed, file-mirrored coordination skill.
- ✓ Temporary (1-7 days).
- ✓ Budget-capped.

What it is NOT

- ✗ Not Git.
- ✗ Not a general chat app.
- ✗ Not a project management suite.
- ✗ Not a long-term memory system.

The Inversion of Control: Managed vs. Forbidden Actions

	Agent Responsibilities (Old Way)	co-op Python Engine (New Way)
Coordination State	~~Agents manually edit markdown logs~~	**Python updates SQLite directly**
Information Retrieval	Agents grep directories and parse huge files	Python executes cursor-based delta reads
Memory Management	Agents read full transcripts	Python enforces token-capped summaries
Conflict Resolution	Overwriting live files	Python verifies SQLite claim conflicts pre-edit

Design Invariant 1: Python owns all heavy lifting. The agent only receives token-capped, task-relevant slices of coordination state.

Dual-State Architecture: Speed Meets Durability

SQLite Operational Layer

The Operational Authority queried by agents.

Structural index

Write Sequence

1) Validate Schema

2) Update SQLite

- Fast selective retrieval
- Cursor tracking
- Token budgeting
- Claim conflict checks

File Artifacts (.co-op/)

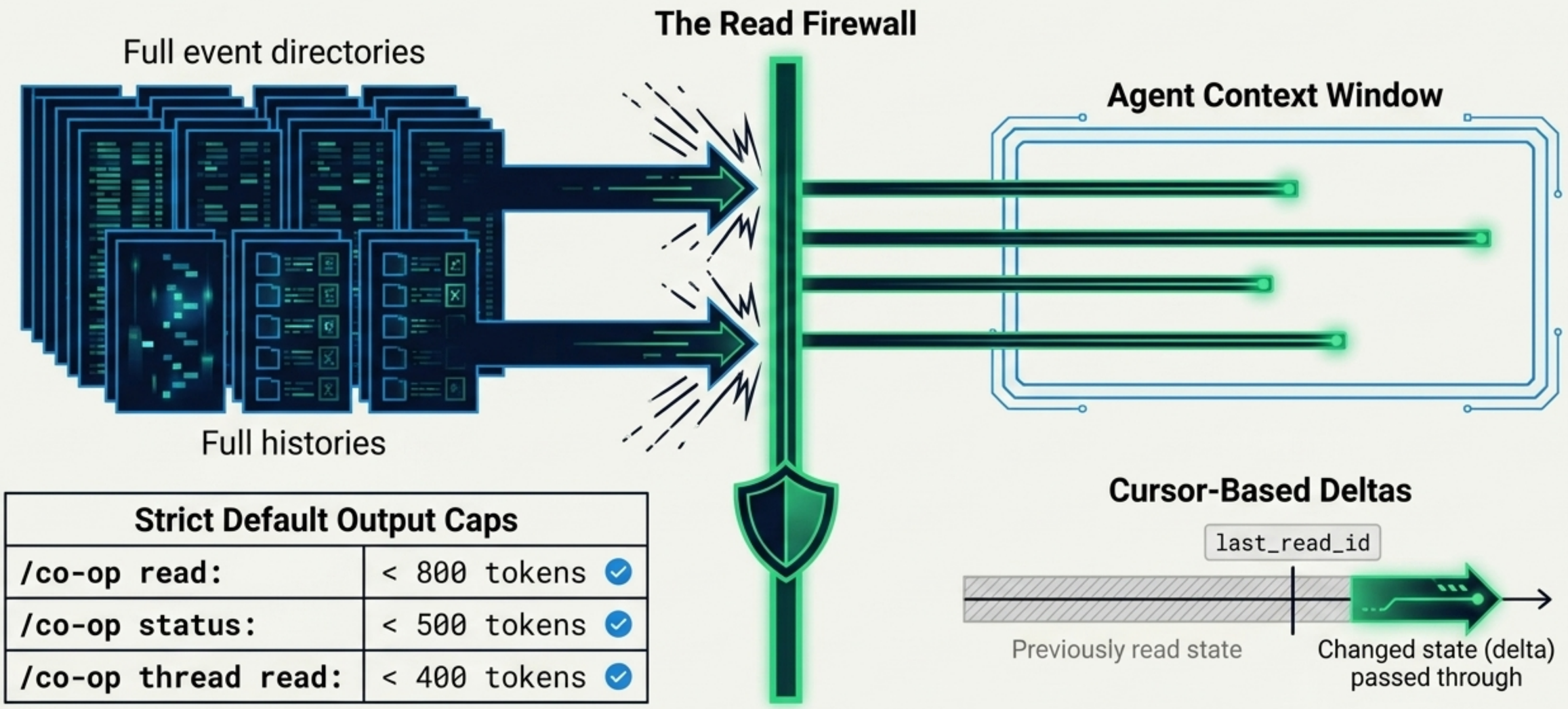
The Durable Audit Trail for operators.

3) Atomic rename/write to File

4) Return capped response

- Human-readable markdown
- Durable JSONL exports
- Recovery source

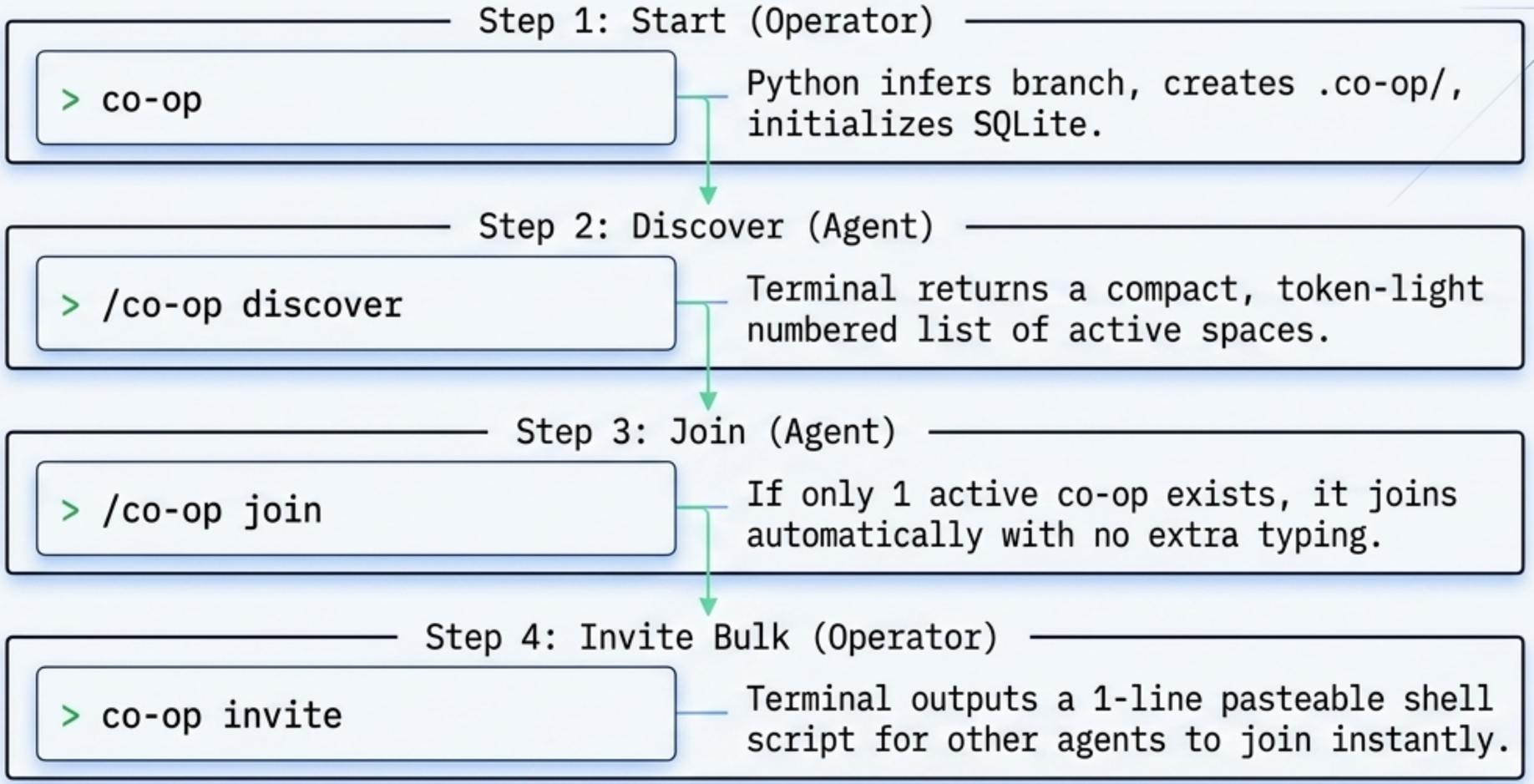
Token Economics & The Read Firewall



Strict Default Output Caps	
`/co-op read:`	< 800 tokens
`/co-op status:`	< 500 tokens
`/co-op thread read:`	< 400 tokens

Target co-op overhead: less than 5% of total agent token usage.

Minimal Typing: The One-Command Flow



The product minimizes typing for the human operator and the agent. The system auto-detects environment, branch, and tool context.

The Coordination Primitives Toolkit

`/co-op task`

Tasks

Lightweight work items.

Key Fields: status, owner, summary.

`/co-op claim`

Claims

Collision prevention for files, directories, and conceptual components.

Prevents overlapping edits.

`/co-op ask / decide`

Blockers & Decisions

Resolves impediments without repeated debate.
Tracks unresolved questions.

`/co-op update`

Check-ins

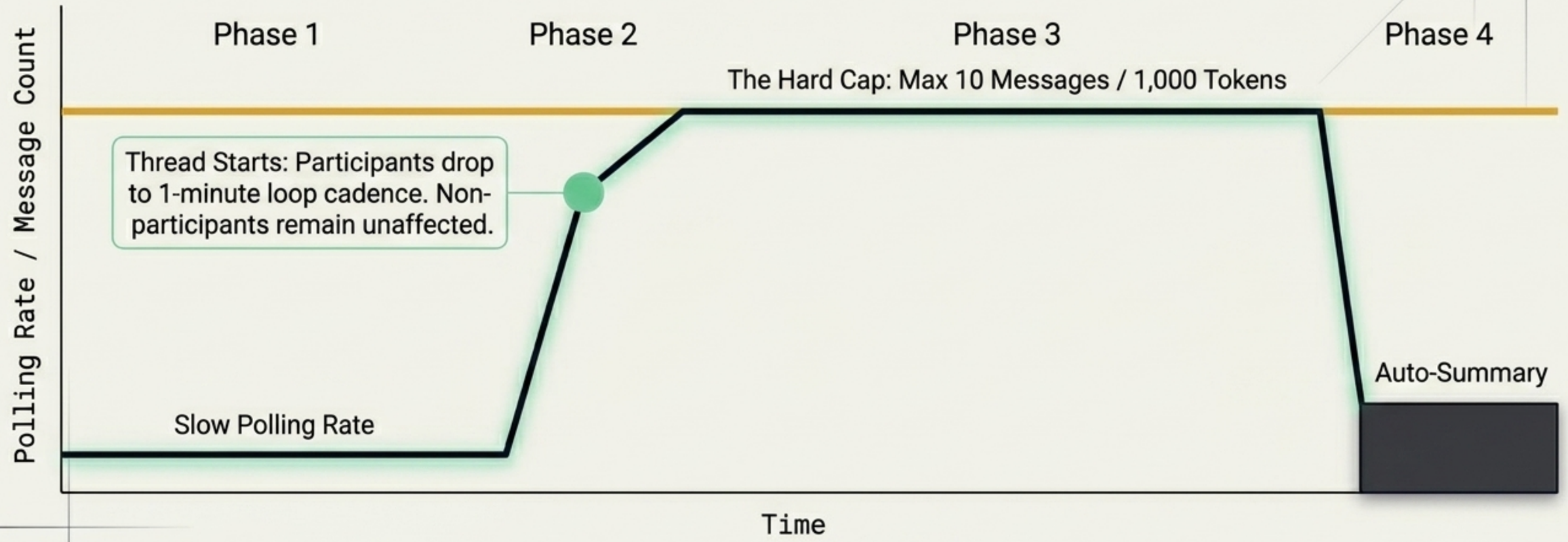
Manager-style status updates.

Key Fields: One-line summary, completed tasks, open items.

All primitives are written directly to SQLite, entirely skipping the need for agents to format complex Markdown files.

Bounded Live Communication

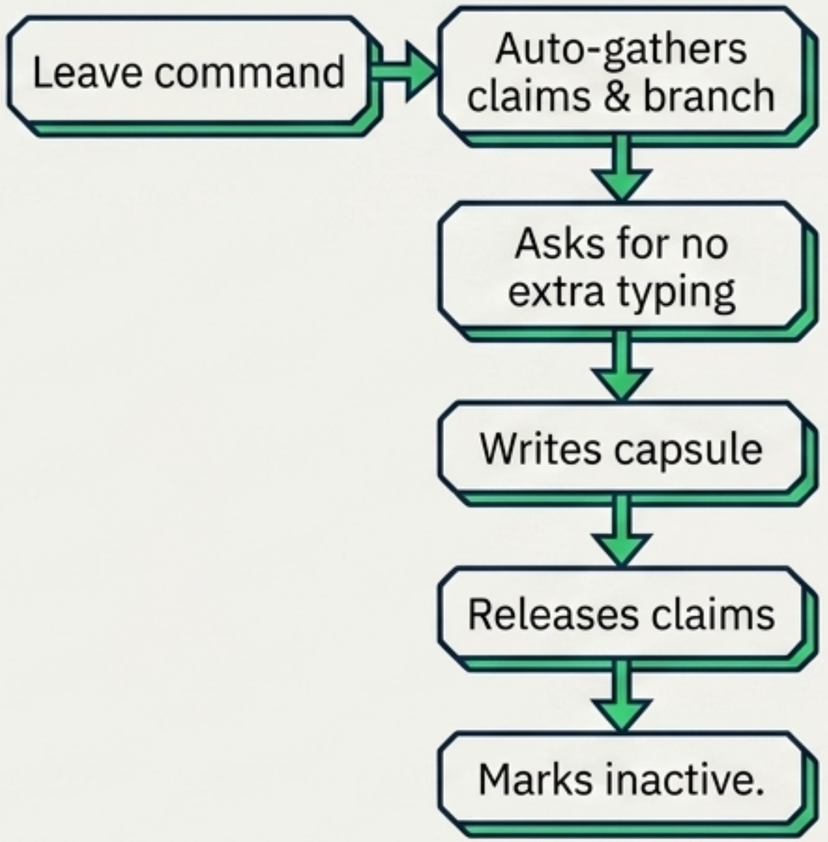
Activation Curve



Live threads are for tactical coordination, not architecture essays. Once closed, Python instantly generates a compact summary. Future agents read the summary, never the full transcript.

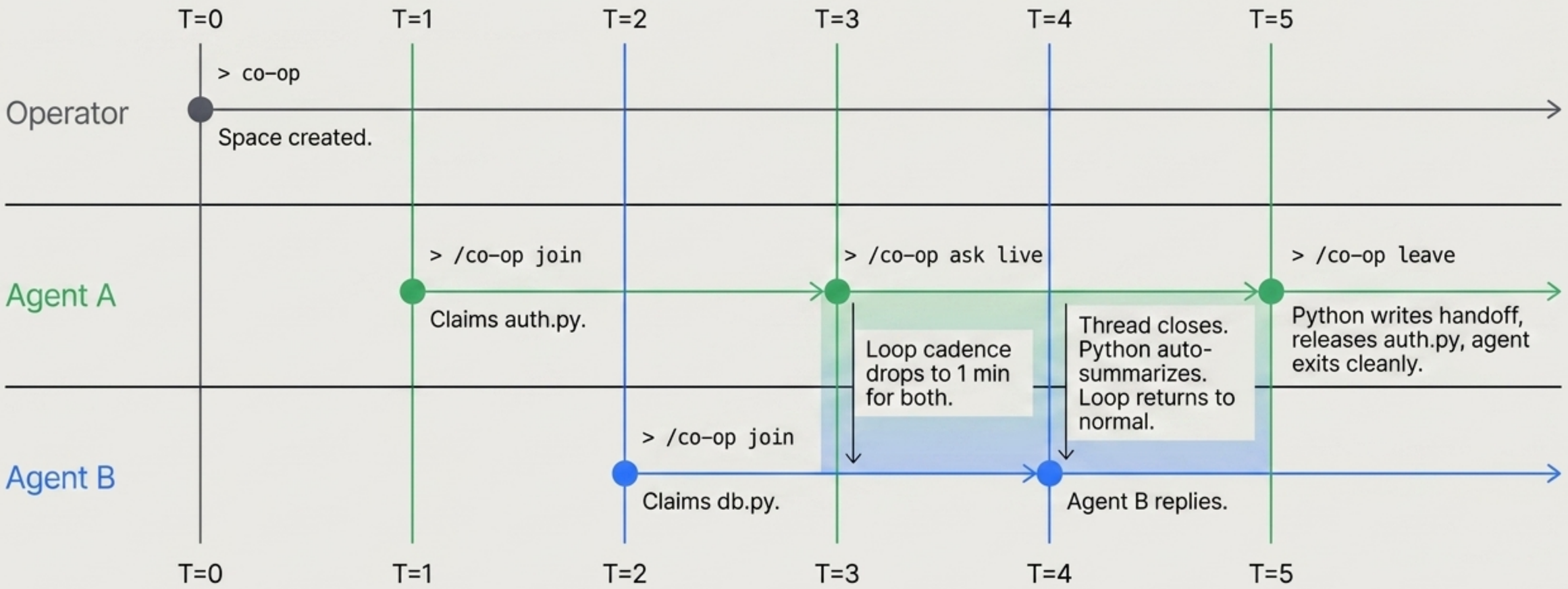
The Auto-Handoff Capsule

```
> /co-op leave
```



Leaving without a handoff defeats the system's value.
co-op `leave` makes context preservation automatic.

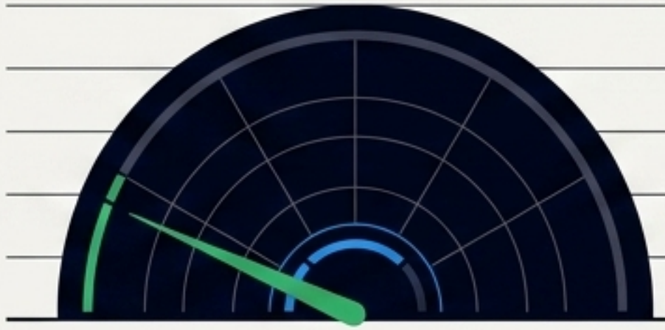
End-to-End Orchestration Flow



Complex mechanics—SQLite updates, token budgeting, loop drops—are completely invisible to the users.

Success Metrics & Dashboards

Token Constraints



<5% overhead

Median /co-op read
< 800 tokens

Live-thread reads
< 300 tokens

Frictionless UX

1-Command

Start, Discover, Join, and Leave must all execute via a single command or ultra-short alias.

Reliability & Safety

0 Collisions

Claim conflicts detected before overlapping edits.

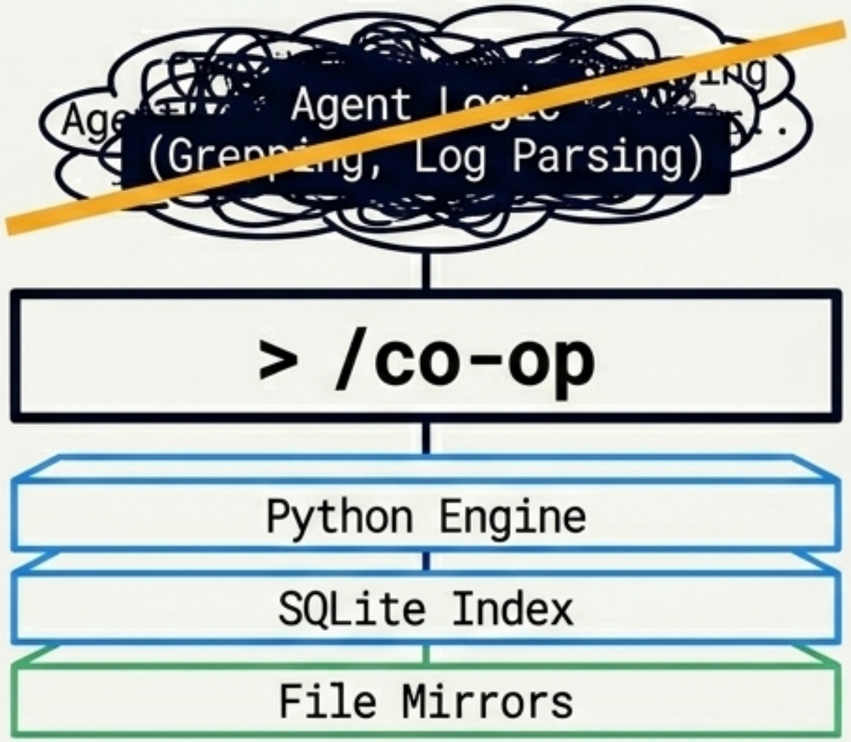
SQLite rebuildable purely from file artifacts.

Auto-stop executed after 6 hours of no meaningful updates.

Implementation Roadmap

```
├── Phase 0: Skeleton
│   (Python package, CLI entry, SQLite schema)
├── Phase 1: Lifecycle
│   (Start, discover, join, leave, invite)
├── Phase 2: Token-Controlled Reads
│   (Cursor tracking, budget enforcement, hot-state compiler)
├── Phase 3: Primitives
│   (Tasks, claims, blockers, handoffs)
├── Phase 4: Live Comm
│   (1-min loop, 10-msg hard caps, auto-summary)
├── Phase 5: Export & Repair
│   (Markdown/JSONL export, index rebuild)
└── Phase 6: Advanced (Stretch)
    (Merge-risk radar, Git diff summaries, DuckDB export)
```

Synthesis: The Agentic Future is Managed



1. The Agent Never Cares

Agents are freed from parsing massive logs. They query data like SQL.

2. The Operator Barely Types

Complex state management is abstracted behind 1-line aliases.

3. Python Protects the Budget

Token limits, hard caps, and auto-summaries guarantee scaling without financial bloat.

A small, temporary, low-token primitive that makes multi-agent coding reliable, auditable, and scalable.