The Single-Writer Kernel:

A Local Transactional Reference Monitor for Agent Swarms

Erich Owens

Curiositech LLC

engineering@portdaddy.dev

September 2026
Version 1.2 (textbook edition)

Abstract

A swarm of autonomous coding agents sharing one machine collides over the same scarce things: network ports, files on disk, mutual-exclusion locks, and the record of who did what. The instinct, trained on a decade of distributed-systems literature, is to reach for consensus — replicate the state, run an agreement protocol. We make the opposite, and we argue correct, move: collapse the entire problem onto a **single writer over a single local SQLite database in write-ahead-log (WAL) mode**, and let the operating system's file lock serialize every mutation. There is one decider, so there is no agreement to reach.

This paper presents that kernel — the **substrate layer** of a four-layer coordination stack — together with the implemented half of the coordination protocol that rides directly on it (a durable message bus, decaying stigmergic markers, violable commitments, a cryptographic delegation chain, and a policy monitor) as a **single-writer transactional reference monitor** in the sense of Anderson and Lampson: a small, tamper-evident mediator of every security-relevant operation, realized locally rather than as an abstract security kernel. We state the kernel's invariants as theorems — mutual exclusion, idempotence, the consistency model (serializable transactions, a linearizable claim history), oracle-bound obligation closure — and we are deliberate about where the promises stop. In particular we *split* durability by fault class: a write that returns success survives a *process* crash but is *not guaranteed* against power loss under the default WAL configuration; and we correct a widespread over-claim, showing that the policy monitor is a post-commit *detector*, not a pre-commit regimenter, with the only genuine pre-commit regimentation credited to a uniqueness constraint and a boot-time admission gate. The kernel is solid where it is local and provisional exactly where a cross-machine economy of agents would need it to become cryptographic and continuous — and we say so in the same words throughout. Companion chapters in the same series (*The Legible Swarm*, *From Spawn to Person*, *The Harbor Economy*) build the operator interface, the identity-and-reputation bridge, and the cross-machine market on the floor this paper specifies.

Keywords: reference monitor, single-writer transactions, SQLite write-ahead logging, durability

by fault class, linearizability, deontic logic, commitments, stigmergy, local-first coordination, multi-agent systems

Reading time: approximately 45 minutes end-to-end; the Reader's Map (§2) routes you to a 15-minute path and to single-section paths. This is Chapter 1 of The Harbor, the Person, and the Economy*, a eight-chapter textbook, and the chapter every later one rests on.* The Legible Swarm *is the operator-facing entry point and a natural place to start;* From Spawn to Person *builds an identity-and-reputation bridge on top of this kernel;* The Harbor Economy *carries the cross-machine market, including cross-machine capability transfer — which belongs to the cross-machine layer and lives there, not here. Any chapter can be read first, but every later layer rests on this one.*

Contents

1 Where this paper sits, and what it promises

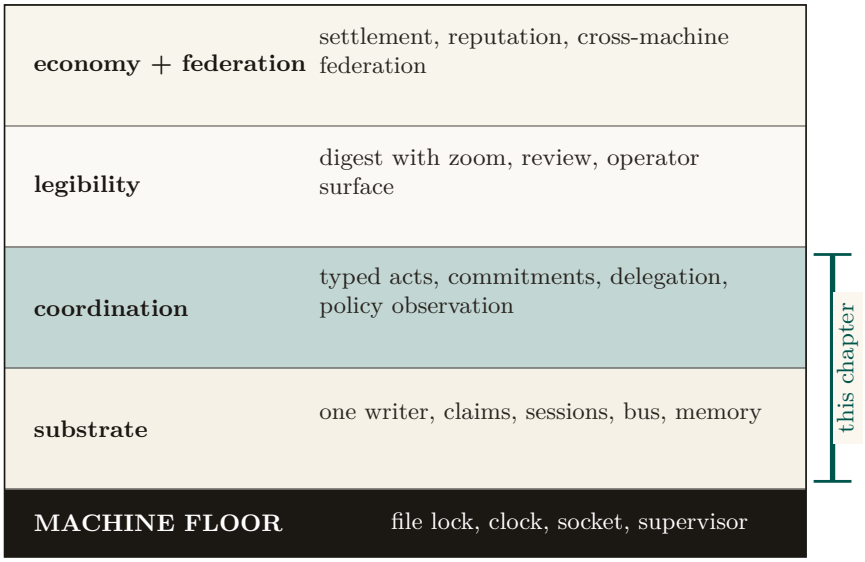


Figure 1: An architectural section through the four-layer coordination stack. This chapter is the second band from the floor: the transactional substrate, providing the one-writer record and claims that every higher layer rests on. The machine floor beneath it is the narrow dependency boundary the substrate assumes: a file lock, a clock, a socket, and a supervisor. [internal]

We model an agent-coordination system as a four-layer stack (Figure 1), each layer serving a different *whom*.

Substrate (the machine).
A single always-on daemon over one local database: durable storage and serialized mutation. *This paper's core.*

Coordination (the agents).
The protocol agents speak over the substrate: ownership claims, a message bus, stigmergic markers, obligations, and policy enforcement. This paper specifies the implemented half of it.

Legibility (the operator).
The human-facing view that renders the swarm intelligible. Treated in an accompanying chapter (*The Legible Swarm*).

Economy (the market between operators).
The cross-machine market in which operators trade work and reputation. Treated in an accompanying chapter (*The Harbor Economy*).

You climb the stack in order, and you never buy a layer before you need it. This paper is the structural floor. The legibility layer reads the kernel's claims, bus, and policy monitor to render the swarm; the personhood argument leans the entire notion of a *durable agent identity* on the kernel's continuity machinery; the economy's settlement ledger is, mechanically, another set of tables under the same single writer, and its cross-machine capability transfer rides the kernel's cryptographic delegation chain. If this floor is unsound, everything above it inherits the fault. So the discipline of this paper is to state each promise precisely and, where the promise has an edge, to draw the edge in ink.

1.1 The one-sentence thesis

> *The kernel is a single-writer transactional reference monitor over a local SQLite/WAL database — the one process that mediates contended resources, and the one place where "true" is decided rather than asserted.*

That is the whole claim, and it has two halves, which are the kernel's entire job; everything else is convenience.

1. **Durability (with an edge):** a write that returned success is durable across the death of any agent — and, we will be careful, across the death of the *daemon process*, though *not* unconditionally across power loss.
2. **Mediation (with a correction):** a state the kernel *regiments* cannot come to exist; a state the kernel merely *enforces* is detected and compensated within a bounded window. These are different guarantees and we will never conflate them.

1.2 Why "reference monitor," and why *not* consensus

The right mental model is the **reference monitor** of Anderson and Lampson [1, 2]: a mediator that is *always invoked* (complete mediation), *tamper-evident*, and *small enough to verify*. The kernel instantiates this not as an abstract security kernel but as a concrete local daemon over one SQLite file. Complete mediation holds *by construction over the kernel's own state*: because every mutation of *coordination state* routes through one writer holding one file lock, the operating system's serialization *is* the mediation primitive — Saltzer and Schroeder's "economy of mechanism" [3] done with one moving part instead of N hand-rolled critical sections. We are deliberate about the scope of the word *complete*: it is complete over writes that *enter* the daemon, not over what a same-user agent can do *beside* it. A classical reference monitor earns "always invoked" from the hardware/operating-system boundary that forces every access through it; this kernel has no such boundary at the substrate layer, so a same-user process can mutate shared state the daemon never sees — edit a file under an advisory claim, run `git` directly, or write the SQLite file itself — without routing through the writer. Mediation is therefore complete over the daemon's *application-program interface* and advisory everywhere else; making it complete over the agent's host capabilities is an out-of-band enforcement problem we scope to the threat model (§13) and credit, when it lands, to a separate-user kernel boundary that does not yet exist (§13.2).

Key idea. The swarm *looks* like a distributed-systems problem — many agents, contention, failover — so the trained reflex is Raft, Paxos, CRDTs. The kernel's defining move is to refuse that problem. One writer, one machine, one file: there is nothing to *agree* on, so the consensus impossibility of Fischer, Lynch, and Paterson [14] does not bite. It is sidestepped, not solved. Distribution is a cross-machine concern that belongs to the economy layer; pushing a consensus protocol down into the local substrate would be the wrong layer.

This is the single most important architectural decision a local-first coordination layer can make, and the field is littered with systems that got it wrong by manufacturing a consensus problem they did not have [12]. We will state the formal payoff of the choice as a conditional theorem in §11: when every read and write is routed through the sole daemon connection and responses follow commit, transactions are *serializable* and claim/lock operations are *linearizable* — the property that lets the coordination layer treat "who holds this" as ground truth without any agreement protocol at all.

1.3 A research-maturity scale, and why the grades matter

A systems paper that mixes "we proved this" with "we hope to build this" is unfalsifiable. So every consequential claim in this paper carries an explicit *maturity grade* on a four-point scale,

plus two refinements introduced here because a single grade cannot formally carry a claim that spans fault classes or interception points. The grade is a property of the *claim*, not a marketing adjective: it says how far the claim is from a verified, running artifact. The concrete artifacts behind each grade are catalogued in Appendix A; the body argues at the level of mechanisms.

Table 1: The research-maturity scale used throughout. The last two rows are refinements this paper introduces (see §5 and §8).

Grade	Meaning
implemented	Realized, exercised, and true under the production runtime.
PARTIAL	Realized but partial: holds under a narrower condition than the prose might invite, or detects rather than prevents.
SPECIFIED	A concrete design exists, but no running artifact realizes it.
proposed	Named as a direction; no design yet.
I1a / I1b	Durability *split by fault class*: I1a (process-crash) is **implemented**; I1b (power-loss) is *not guaranteed* under the default WAL configuration. A claim that conflates fault classes cannot carry one grade.
regimented / enforced	Prohibition split by interception point: *regimented* = rejected pre-commit, truly unreachable (a uniqueness constraint / boot-admission gate); *enforced* = detected post-commit, halted or compensated within a bounded window (the policy monitor). "Physically unreachable" is reserved for the regimented set.

Pitfall. The grades are not decoration. The two most consequential corrections in this paper — the durability split and the regimentation/detection split — are exactly the places where a single optimistic grade would let the kernel's foundational promise be read as stronger than the artifact delivers. When a system claims "the database survives every agent's death," a reader hears "power-loss durable." It is not, by default. Saying so is the difference between a systems paper and a brochure.

2 Reader's Map

Table 2: Reader's Map. Find your row; read those sections first.

If you are...	...read first	...critical artifact
short on time (15 min)	§1 (thesis + stack map), §5 (durability split), §8 (the monitor correction)	Figs. 1, 3, 6
a systems engineer	§4–§11 (substrate, single-writer, the theorems)	Thm. 11.2; Figs. 2, 11
a security reviewer	§8 (deontic split, the monitor), §13 (threat model)	Fig. 6, Table 5; Table 9
a protocol/agent author	§6 (claims), §7 (the bus), §14 (the interface contract)	Fig. 4, Table 4; §14
here for the economy	§9 (continuity organs), then the personhood and market papers	Fig. 9
an instructor	every **Exercises** block; the open problems OP-1 – OP-11 and their status table (§17)	the starred exercises; Table 10

Reading order within the series. The legibility paper is the gentlest on-ramp; this paper, the substrate, is the formal floor; the personhood and market papers build upward. If you only

read one paper to decide whether the foundation is sound, read this one.

3 The kernel as seven organs

We organize the kernel into seven *organs*, each a contract the daemon must hold for the layers above to be coherent (Table 3). The temptation is to describe such a kernel as a flat noun-list of features — ports, claims, sessions, a bus, markers, commitments, a monitor, a memory store — which is the symptom of treating the kernel as a database. It is not a database. It is a system of record *plus* a mediator, and naming the organs surfaces the connective tissue a noun-list hides.

Table 3: The seven organs, each the contract the daemon holds for the layer above it, with the table that is its home where the chapter names one and its maturity grade. All seven are disciplines over one SQLite/WAL file: one commit history, not seven stores. The grades come from the chapter's own status table (§A).

Organ	Contract held for the layer above	Maturity
transactional substrate	one writer, one durable file; every other organ is a table with discipline over the same commit history	**implemented**
resource / exclusion	at most one compatible holder per conflict domain; leases expire and are swept lazily (home: `claims`)	**implemented**
actor continuity	a stable actor identity across sessions, mechanically present but asserted by the actor, not proven	PARTIAL self-asserted
message carriage	an ordered, cursor-addressable, durable envelope delivered at least once (home: `messages`)	**implemented**
obligation & enforcement	no *done* without a receipt the daemon can re-check; the policy monitor detects and cannot always prevent (home: `commitments`)	PARTIAL partial
session memory	notes and events survive process death; execution state does not	**implemented** (memory)
self-attestation	the daemon reports its own coverage as enforced, degraded or stubbed, never silently	**implemented**

The **substrate organ** is the floor; the other six are, mechanically, tables *with discipline* over the same file. We treat the substrate and the two organs that carry the paper's sharpest corrections (resource/exclusion and obligation/enforcement) in full, and the rest with the depth their novelty warrants.

Exercises 1.1–1.3, page 41.

4 The substrate organ: one writer, one file

The substrate is the bedrock everything else is just a table in. Its job is to make two promises — durability and serialized mutation — and to make them formally. We cover the single-writer discipline here and durability in §5, because durability is where the most consequential correction lives.

4.1 The single-writer discipline

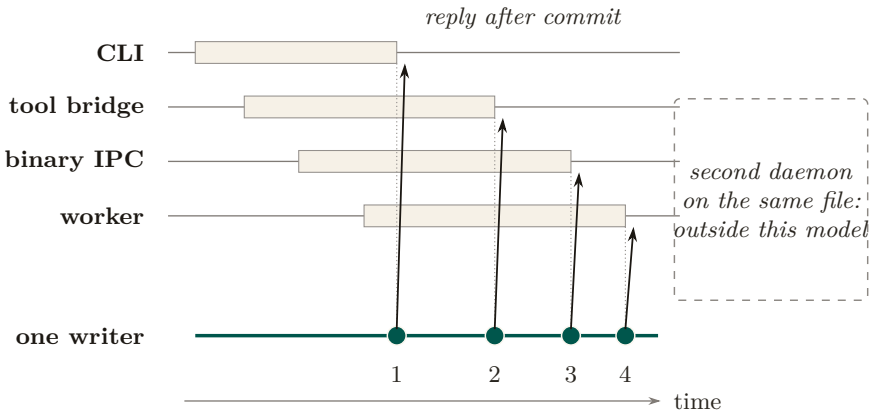


Figure 2: Four concurrent callers and one writer. Each band is a request in flight; its right edge is where the daemon commits it, and the rail shows the resulting order 1–4. A reply is sent only after the commit. A second daemon opening the same file is outside the model, not another lane. [internal]

The kernel's concurrency story (Figure 2) is: dozens of command-line invocations and agent connections funnel through one writer — the daemon, the sole holder of a read–write connection to the database file — and SQLite's operating-system file lock plus a bounded busy-wait serialize all mutations. We must be precise about *what* serializes writers, because there are two stories and only one is true at a time.

Definition 4.1 (Single-writer discipline)**.** All state-mutating operations route through a single daemon process holding one SQLite write connection. Concurrency among the many clients (command-line invocations, agents over the message transports, background workers) is resolved by the daemon's request queue; the SQLite file lock is the *backstop* that preserves correctness if the discipline is ever violated by a second writer (for example a stray direct write), not the primary serialization mechanism.

Key idea. "Single-writer" is an architectural *discipline* (everything goes through the daemon), backstopped — not replaced — by SQLite's file lock. This distinction matters: SQLite permits multiple writer connections (it serializes them, it does not reject them), so the property holds because of where writes are routed, not because the database forbids a second writer. The standing risk is therefore a second *copy* of the daemon coming up against the same file — which is why the substrate also runs self-checks that detect a divergent or stale second instance (§10).

4.2 The configuration *is* the durability claim

The substrate's storage configuration is short and decisive. In WAL mode with the *normal* synchronization level (the WAL is fsync'd at checkpoints rather than on every commit), a bounded auto-checkpoint interval, a bounded busy-wait, foreign-key enforcement on, and a clean-shutdown checkpoint-and-truncate, the kernel inherits exactly the crash semantics that configuration defines. The claim "a write that returned success survives a crash" reduces *entirely* to what WAL with normal synchronization guarantees — which is the subject of §5, and which is not what an unguarded reading assumes.

4.3 Schema evolution: an ordered boot ledger, not yet a sequencer (OP-7)

The initial schema-by-union approach (`CREATE TABLE IF NOT EXISTS`) created vulnerabilities around unsequenced renames and multi-column backfills. The kernel's answer so far is a partial

one, and the honest way to report it is to name the trade it makes rather than the property it saves.

What ships (PARTIAL) is a **boot-time migration ledger**: the daemon applies a numbered, totally ordered set of migrations before it serves any request, and then *verifies the schema it is about to serve from* — probing the actual tables and column sentinels rather than trusting that the statements ran without raising — and refuses to serve on a mismatch. That is a real fail-safe default, and it is the reason a half-applied migration cannot quietly become coordination truth.

What does *not* yet ship is the sequencer that would make that order authoritative. Organ modules still self-initialize their own tables over the shared database handle (Appendix A), so the schema is still, in part, a union. A strict linear ledger managed exclusively by the daemon via SQLite's `pragma user_version`, in which modules *declare* migrations rather than applying them, is SPECIFIED. OP-7 therefore stands PARTIAL: this is Table 1's second question — "is a version table unavoidable?" — answered *probably yes*, at the cost of the "any module, any order" flexibility the schema-by-union design offered, and not yet paid for in full.

4.4 Path and ownership invariants

The database resolves to a single canonical path with owner-only file permissions (historically the file also carried the system's signing key in plaintext; that secret is being migrated to the operating system's keychain, see §13). A fail-closed guard refuses to open the live registry from a test context — the analogue of a framework's protected-environment check. This is Saltzer and Schroeder's *fail-safe defaults* [3] applied at the one place a test could corrupt production truth.

4.5 The second writer: institutional truth above the local harbor

A team runs several harbors and wants one plan across them: which objectives are authorized, who owns what, which transitions were accepted. The product calls that surface **Chartroom**, under the umbrella name Grand Harbor, and its design record (ADR-0137, Proposed, 2026-09-01) is the first place this chapter's invariant meets a second writer. The record exists on an unmerged branch; no Chartroom code is on the main line, and the book grades it SPECIFIED throughout this section.

The design rule is short enough to state before the mechanism. *A second writer above the local harbor is either a projection of the local record or the writer of it, never both, and the switch between the two is an epoch.* A projection may be copied, searched, and rendered anywhere; it carries no authority, and a client that writes to it has written nothing. The writer is the one process whose accepted transitions define the record, and there is one per record per epoch. The design record draws this line itself: the harbor's own signing key is the sole signer of Chartroom intents, the relay verifies and applies an intent but never authors one, and the record rejects reusing the existing roadmap-snapshot route precisely because that route is a full-replace mirror whose contract keeps the daemon authoritative. That is a projection, and the record declines to let a projection masquerade as a writer. This is a *design invariant*: intended to hold, backed by the single-writer implementation for the local case, and not yet exercised across harbors by any code.

Moving the writer is the part no product yet does, and it is a procedure, not a setting. (1) Freeze: the local harbor stops accepting writes and records the freeze as its last transition. (2) Export: the ledger and its root hash leave the harbor as one artifact. (3) Verified import: the receiving authority imports the artifact and proves, against the root, that what it holds is what left. (4) Check: projections and permissions are re-derived on the new side and compared with the old. (5) Epoch: the new writer's epoch is signed and published, and every client refuses a write under any older epoch from that moment. (6) Rollback only by another epoch: there is no return to the old writer except by running the same procedure in reverse, so the record never has two writers, even for the duration of a mistake. The reason the switch must be an epoch

and not a flag is the same reason revocation must be an epoch in *The Anchor Protocol*: the model-checked attack there restores a backup taken before a revocation and thereby resurrects a dead credential, and it is refuted only when state carries the epoch it was valid under. A restored harbor database that predates the cutover is the same attack against authority, and the same defence closes it. That transfer is a *model-checked property* for credentials and a *design invariant* for writers; nothing in the repository yet checks the writer case.

> **Where this stops.** The alternative the procedure exists to forbid is dual writing with reconciliation: both authorities accept writes for a while and receipts reconcile them afterwards. That is two truths, the condition every result in this chapter assumes away, and no receipt repairs it after the fact, because the receipt itself would need a writer. The design record's own deployment plan is consistent with this section: a staged migration, a fixture receipt, a rollback probe, and a root-authorized production write, with no import of existing plans or documents in the first step. Until that plan runs, a remote Grand Harbor is a projection over local authorities, which is exactly what the relay is today.

Exercises 1.4–1.6, page 41.

5 Durability, split by fault class

This is the most important correction in the paper, so it gets its own section. The kernel's foundational promise — "a write that returned success survives" — is true, but only for the fault class a careful reader expects and not for the one the pitch most invites.

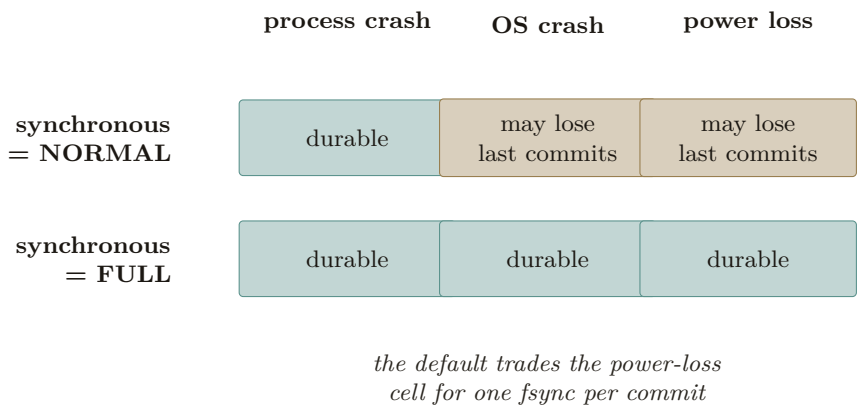


Figure 3: Durability by fault class under the two synchronization levels. The reference implementation ships `synchronous=NORMAL`: a process crash never loses a returned write, but an OS crash or power loss can roll back the most recent commits. `synchronous=FULL` buys the power-loss cell too, at the cost of an fsync on every commit. [internal]

Figure 3 draws the split this section defends: the same write, judged against two different fault classes with two different answers.

5.1 Why one label is a lie here

Gray and Reuter [9] draw the canonical line between *atomicity/consistency* (a transaction is all-or-nothing and never leaves the store corrupt) and *durability* (the "D" in ACID: a committed transaction survives subsequent failures). Under SQLite's WAL with the *normal* synchronization level, the log is *not* fsync'd on every commit — it is synced at checkpoint [11]. Per SQLite's own documentation, in WAL mode the normal level means "commits might lose durability following a power loss or hard reset," though the database remains uncorrupted. So the honest statement splits in two.

Property 5.1 (Durability by fault class)**.** Let a write return success under the default WAL configuration.

- **I1a (process-crash durability).** If the *daemon process* dies (a forced kill, a panic, an out-of-memory termination), the write survives: the data is in the operating system's page cache, which outlives the process. **implemented**.
- **I1b (power-loss durability).** If the *operating system* crashes or power is lost *before the next checkpoint*, the most recent committed writes may roll back. *not guaranteed* under the normal synchronization level; it would require full synchronization or a checkpoint on the commit path.

Worked dramatization: Alice crashes mid-write. Alice is an agent editing a file. At t_0 she commits a claim on a port and a note recording her edit; the daemon returns success. At $t_0 + 5\,\text{ms}$ her host operating system kills the daemon process outright — a forced termination, no clean shutdown. *What survives?* Both writes. The committed pages live in the operating system's page cache, which the daemon process does not own and does not take with it when it dies; when the supervisor restarts the daemon, SQLite finds a dirty log, replays it, and Alice's claim and note are intact. This is I1a, and it is the fault class the design actually targets: agents die constantly, and the record must outlive them. Now change one detail. At $t_0 + 5\,\text{ms}$, instead of the daemon dying, *the power fails*. The committed pages were in the page cache but had not yet reached stable storage, because under the normal synchronization level the log is fsync'd only at the next checkpoint. On reboot, the database is uncorrupted — but Alice's last claim and note may simply be gone, rolled back as if never written. This is I1b, and it is *not guaranteed*. The two faults differ only in whether the page cache survived; conflating them is the single most common over-claim about WAL-backed storage.

Pitfall. The seductive misconception is "the normal sync level is safe in WAL mode because WAL already guarantees crash safety." This conflates crash-*consistency* (true: the database is never corrupt) with *durability* (false for power loss). The two are different ACID guarantees, and only the first is unconditional here. A systems paper cannot ship the unqualified claim. We split it.

5.2 Why the trade is defensible — and where it stops being so

Trading power-loss durability for throughput is a reasonable default for a local-first developer tool: the failure (lose the last few hundred milliseconds of claims after a power cut) is benign and self-correcting — agents re-claim, heartbeats resume. What is *not* defensible is letting the prose imply the stronger guarantee — and for the few paths that genuinely need I1b, the right fix is a per-write-path selector rather than a global change of synchronization level.

The design, **Selective Checkpointing** (SPECIFIED, OP-10), is small: a high-stakes, money-bearing write path — a settlement-ledger entry, say — appends `PRAGMA wal_checkpoint(FULL);` to its commit path, forcing a synchronous flush to disk, while claim, marker, and note writes keep the fast default. Two things must be said plainly about it. First, *no write path in the reference implementation opts in today*: the only checkpoints it issues are the bounded auto-checkpoint, a passive checkpoint on the maintenance path, and the clean-shutdown truncate (Appendix A). Second, even once a path opts in, the guarantee it buys is *per path*: Property 5.1 continues to govern every path that does not, so I1b remains *not guaranteed* for the kernel as a whole, and a table row or a figure that says otherwise is wrong. OP-10 stays *open* until both the selector and an interface that stops a *new* write path from silently defaulting into the fast lane exist.

5.3 A recovery story, not just a durability story

Durability is about what survives; recovery is about what happens next. On daemon restart with a dirty WAL, SQLite replays the log to a consistent state (I1a's mechanism). But the kernel-level questions are larger: who validates the audit chain on boot, and what becomes of a half-applied cross-organ operation (§12) after a crash? The hook exists — the boot-admission gate that refuses to serve when a critical self-check fails (§10) — but boot-time WAL recovery plus integrity check plus chain verification should be a single named invariant, and today it is closer to PARTIAL than **implemented**.

Exercises 1.7–1.9, page 41.

6 The resource organ: claims, locks, and fair exclusion

Network ports are the namesake and the original sin: two agents that each start a development server on the same port collide, and one silently fails. The resource organ generalizes that problem into a single exclusion primitive — over network ports, over named mutex locks, and over edit-surface claims on regions of source files — to which every higher layer's notion of ownership reduces.

6.1 Claim granularity is richer than "claims"

It is tempting to collapse all three of these into one word. The kernel instead distinguishes **whole-file**, **line-range**, and **symbol-path** claims, each keyed by the file path together with, respectively, nothing, a line interval, or a syntactic symbol identifier. This is the substrate for a "reserve regions, not whole files" coordination policy and for semantic-conflict prediction over a syntax-aware symbol index — but at the substrate layer it is simply exclusion at three granularities.

6.2 The claim lifecycle as a finite-state machine

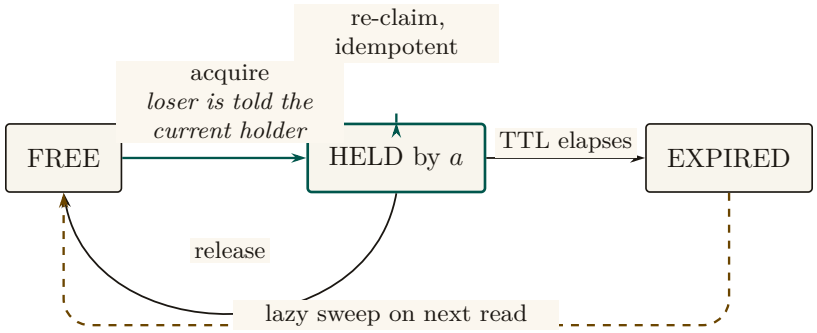


Figure 4: The guarded claim lifecycle. Acquire is one atomic uniqueness decision that also tells a losing caller who currently holds the claim; re-claim and release are idempotent; expiry is removed lazily, on the next read, not by a background sweep. The kernel has no queued state and therefore no fairness or bounded-wait guarantee: a losing caller must retry, and nothing schedules it a turn. [internal]

Figure 4 states, as a state machine, the four properties on which the coordination layer builds typed ownership. The acquire path is small enough to state as an algorithm (Listing 1); we then give the central property as a theorem.

```
function acquire(key, requester, ttl):
  // single atomic statement; the uniqueness constraint is the
     mutex
  try:
    INSERT row (key, holder=requester, expires_at = now()+ttl)
    return GRANTED(holder=requester)          // we won
  catch UNIQUE_CONSTRAINT_VIOLATION:
    existing = SELECT row WHERE row.key = key
    if existing.holder == requester:
      UPDATE row SET expires_at = now()+ttl  // idempotent re-
         claim
      return GRANTED(holder=requester)
    if existing.expires_at < now():           // stale holder
      DELETE row WHERE key = key AND expires_at < now()
      return acquire(key, requester, ttl)     // one bounded
         retry
    return DENIED(holder=existing.holder)     // loser learns
       the winner
```

Listing 1: Claim acquisition. The whole acquire is a single insert against a uniqueness constraint; the loser is handed the current holder, not an error to retry blindly.

Theorem 6.1 (Atomic Mutual Exclusion and Fenced Leases)**.** Within a canonical conflict domain (exact normalized key, transactional interval overlap $[a, b] \cap [c, d] \neq \emptyset$, or path prefix hierarchy), at most one holder can hold an active lease at any logical instant. Furthermore, every granted lease carries a strictly monotonic fencing epoch $e \in \mathbb{N}$; downstream effect brokers reject any operation bearing an expired epoch $e' < e_{\text{current}}$.

Proof sketch. Within an immediate SQLite transaction (`BEGIN IMMEDIATE`), the daemon evaluates the conflict predicate against all currently unexpired leases. For exact keys, this is enforced by primary-key uniqueness; for intervals and hierarchies, by an indexed range check within the same transaction. Because all writes funnel through the single connection (Def. 4.1), predicate evaluation and insert are atomic. Upon grant or reallocation, the daemon increments the resource's fencing epoch e. Downstream effect brokers validate that e matches the active lease at execution time, preventing stale or paused holders from producing side effects after lease expiration. □

Worked dramatization: Alice and Bob race for one port. Alice and Bob are two agents, started a millisecond apart, that both want to bind a development server to port 7421. Each issues `acquire(7421)` (Algorithm 1). Both requests reach the single daemon, which serializes them: one insert lands first. Say Alice's does. Her insert succeeds and she is granted the port. Bob's insert, arriving an instant later, hits the uniqueness constraint on key 7421 and is rejected *atomically* — there is no moment at which both rows exist. Crucially, Bob does not get a bare error. The catch path reads back the row and hands Bob the *identity of the holder*: "port 7421 is held by Alice." Bob can now pick another port, wait, or message Alice, but he can never double-bind. The serialization that makes this clean is not a hand-rolled critical section; it is the operating system's file lock funnelling both inserts through one writer, and the database's primary-key uniqueness deciding the winner. One writer, one constraint, one holder.

Pitfall. Theorem 6.1 is sound for the *fresh-contention* case above. A subtler hazard hides in the *stale-holder* branch: reclaiming an expired lock runs a delete (of the expired row) and *then* an insert as two separate statements rather than one transaction. On the single daemon connection this is serialized by the connection itself and is safe. But if a second writer connection ever existed — the very scenario the single-writer discipline exists to prevent — a deferred

transaction would open a window for a transient "database busy" error mid-sequence rather than clean serialization; the fix is to begin the transaction in immediate mode. So Theorem 6.1 holds because all writes route through the one daemon connection (Def. 4.1), *not* because the expire-then-acquire sequence is itself atomic. State which story you mean; do not tell both.

What the discipline looks like from two terminals: one agent holds a lock on a file, a second asks for the same name and is told who holds it and for how long, and only after the first releases does the second acquire. Nothing here is advisory — the daemon is the single writer for the `locks` row, so the second request cannot race the first.

At the terminal. *two agents, one lock, one holder at a time.*

```
$ export PORT_DADDY_AGENT_ID=alice
$ pd lock lib.webhooks.ts --ttl 600000
SUCCESS: Acquired lock: lib.webhooks.ts
  TTL: 600s

$ export PORT_DADDY_AGENT_ID=bob
$ pd lock lib.webhooks.ts --wait --timeout 2000
Lock 'lib.webhooks.ts' is held by <actor-id>
  Held since: <timestamp>
  Expires in: <n>s

$ pd locks

NAME                          OWNER                    EXPIRES
------------------------------------------------------------------------
lib.webhooks.ts               <actor-id> <timestamp>

Total: 1 lock(s)

$ export PORT_DADDY_AGENT_ID=alice
$ pd unlock lib.webhooks.ts
SUCCESS: Released lock: lib.webhooks.ts

$ export PORT_DADDY_AGENT_ID=bob
$ pd lock lib.webhooks.ts
SUCCESS: Acquired lock: lib.webhooks.ts
  TTL: 300s
```

6.3 Fairness without a scheduler: the Stigmergic Ticket-Lock (OP-1)

The conspicuous gap in the claim lifecycle (Figure 4) is the absence of a QUEUED state. Locks have a time-to-live, but acquisition is racy first-come, so a fast agent churning a hot file can starve a slow agent indefinitely.

The design that would close OP-1 without dragging a background scheduler into the kernel is a **Stigmergic Ticket-Lock**. It is SPECIFIED — specified here in enough detail to build and to attack, but not realized: no `claim_tickets` table exists in the reference implementation, and the lifecycle above is still the one the kernel runs. It would introduce a `claim_tickets` table:

```
CREATE TABLE claim_tickets (
  resource_id TEXT NOT NULL,
  agent_id TEXT NOT NULL,
  seq_num INTEGER PRIMARY KEY AUTOINCREMENT
);
```

When Agent B's claim acquisition failed against the primary-key `UNIQUE` constraint, the error handler would insert an entry into `claim_tickets`. When Agent A then called `release('auth.ts')`, the daemon would execute a single atomic transaction:

1. Delete Agent A's active claim row: `DELETE FROM claims WHERE resource = :r;`
2. Read the lowest ticket:

```
SELECT agent_id, seq_num FROM claim_tickets
WHERE resource_id = :r
ORDER BY seq_num ASC LIMIT 1;
```

3. Atomically re-grant the claim: `INSERT INTO claims (resource, holder, ttl) VALUES (:r, :next_agent, :ttl);`
4. Remove the claimed ticket: `DELETE FROM claim_tickets WHERE seq_num = :next_seq;`

Because the state transition would be driven entirely *reactively* by the releasing agent inside one transaction, FIFO ordering among waiters follows from the ticket table's own monotonic sequence, with no background timer or polling thread in the kernel — which is the property that makes the design worth stating.

Its honest edge is the path it does *not* cover. A holder that never calls `release` — it crashed, or its lease simply expired — returns the resource through the lazy TTL sweep (I4), and the sweep consults no ticket queue. So bounded wait would hold for cooperative release and fail on exactly the failure path the kernel was built to expect. OP-1 stays *open* until the sweep and the queue are the same code path, and until the ticket table exists at all.

Exercises 1.10–1.12, page 42.

7 The communication organ: the bus, stigmergy, and two delegation chains

Table 4: The communication organ: one durable carrier and two provenance traces that must never be merged. The envelope fields are what the bus stores; the two traces answer two different questions and are checked by two different mechanisms.

Piece	What it is	Status
the bus envelope	version tag, time-to-live, message kind, typed act with its conversation id, actor, reason and event reference; ordered by a durable history cursor	**implemented**
authorization chain	the cryptographic object: who authorized whom, hop-bound, attenuation-monotonic, anti-replay	**implemented** (ProVerif)
coordination lineage	the coordination object: what work is already in flight, so loops and upward blocks can be detected	SPECIFIED

The communication organ (Table 4) is where the substrate stops being a lockbox and becomes a medium for conversation. The carrier is **implemented**; the *semantics* that ride on top of it (typed speech acts, protocol state machines) belong to the coordination layer. Three pieces matter here.

7.1 The bus: a durable carrier envelope

The bus is a durable, ordered, *at-least-once* publish/subscribe channel over a messages table, with a versioned envelope (a version tag, a message kind, a body, and an in-reply-to pointer), a history cursor for replay, and a per-message time-to-live. The coordination layer's typed speech act — carrying a performative, a conversation identifier, a coordination lineage, and a reply deadline — is layered *inside* this envelope. The kernel ships the carrier; the coordination layer ships what it carries.

Key idea. At-least-once delivery means a *healthy* bus routinely redelivers and reorders. This collides with a loud-fail honesty discipline: if "every anomaly is loud," benign redelivery drowns the operator in false alarms. The resolution is that benign transport non-determinism must be *deduplicated silently*, and only genuine protocol-state-machine violations surface loudly. That requires an idempotency key in the envelope — currently absent — which is the cheapest high-leverage fix at the coordination layer. The kernel provides the bus; making every speech-act effect idempotent is the coordination layer's debt, flagged here so the reader does not mistake silent deduplication for dishonesty.

7.2 Stigmergic markers: decay as a provided guarantee

The kernel also carries *stigmergic markers* — numeric coordination signals deposited in shared state, after the manner of Grassé's stigmergy in social insects [19] and the generative-communication tuple spaces of Gelernter's Linda [20], and used in ant-colony optimization [21]. Each marker carries per-kind exponential decay $w \cdot r^{\Delta t}$, computed both on a background evaporation tick *and* at read time (so a reader sees the correctly decayed value without waiting for the tick), and pruned below a small threshold. The decay is the structural defense against "blackboard rot": stale coordination signals evaporate on their own. The calibration of the decay rate is genuinely open (OP-8): too slow and signals rot, too fast and they evaporate before they coordinate.

Numbers by hand — The decay arithmetic a calibration will use. Take $w_0 = 1$ and $r = 0.9$ per tick, with a prune threshold of 0.05. Then $w(5) = 0.9^5 = 0.590$, $w(10) = 0.349$, $w(20) = 0.122$, and the marker is pruned at the first tick where $0.9^t < 0.05$: $0.9^{28} = 0.052$ survives and $0.9^{29} = 0.047$ does not, so the marker lives 29 ticks [internal]. Moving the rate half the distance to one ($r = 0.95$) more than doubles that life, to 59 ticks, because $\ln 0.05 / \ln 0.95 = 58.4$. The calibration question of OP-8 is which of these lifetimes matches the time a signal stays true; the arithmetic is the same at every answer.

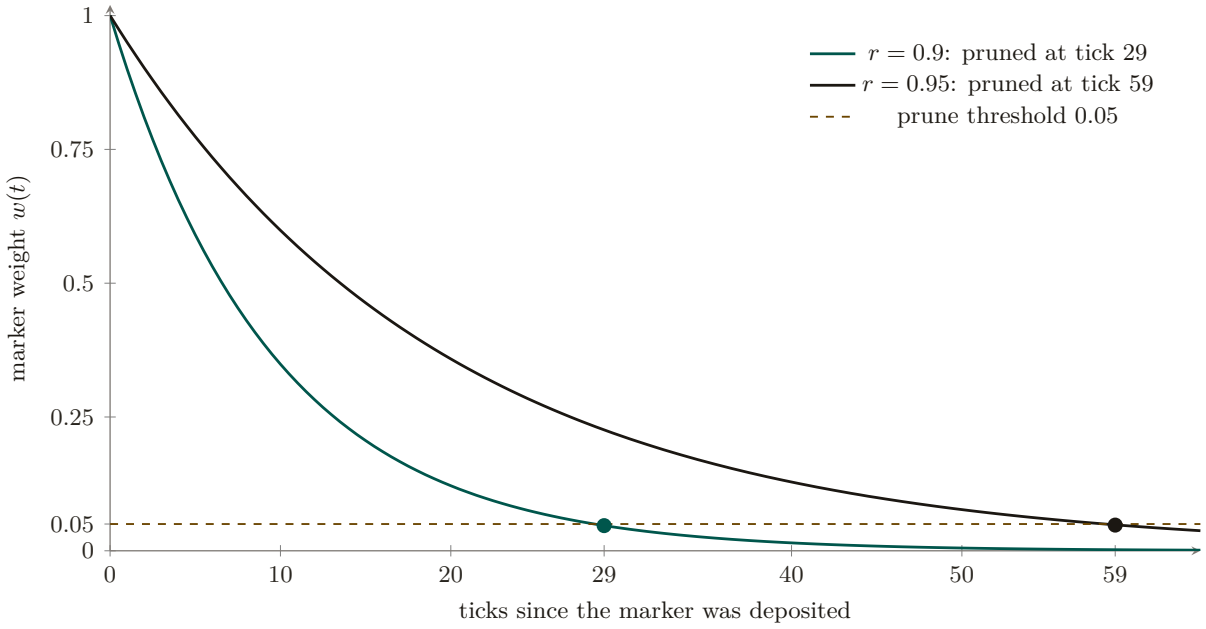


Figure 5: Marker weight $w(t) = r^t$ for two decay rates against the prune threshold of 0.05. At $r = 0.9$ the marker survives 28 ticks and is pruned at tick 29; at $r = 0.95$ it lives until tick 59, since $\ln 0.05 / \ln 0.95 = 58.4$. Moving the rate half the distance to one more than doubles the lifetime, which is the whole content of the calibration question. [internal; analytic]

7.3 Two delegation chains, one dangerous name

A dangerous naming collision lives in this organ. The phrase "delegation chain" names *two different objects*, which we keep rigorously distinct:

- the **authorization chain** — the *cryptographic* object (**implemented**, mechanically verified in a protocol-verification tool): a hop-bound, attenuation-monotonic, anti-replay and anti-splice chain of digital signatures that proves *who authorized whom*. This is what cross-machine capability transfer in the economy layer rides on.
- the **coordination lineage** — the *coordination* object (SPECIFIED): the task-lineage over which loop detection and upward-block run, preventing two agents from delegating a task back and forth forever.

These are distinct objects, and the relationship is that the coordination lineage is carried *inside* the authorization chain — loop detection runs over a lineage whose authenticity the cryptographic chain guarantees. We never again write bare "delegation chain."

Pitfall. The "lineage inside chain" relationship is a slogan until the task-shape field is in the *signed* payload at each hop — and that collides with a rule against keyword-based text classification, because loop detection needs a *semantic* task-shape equivalence (a learned embedding or a structural hash), which is not deterministically signable. This tension is a real open problem; we flag it rather than assert it solved.

7.4 A second transport

The kernel ships *two* transports, not one: a request/response protocol over a Unix domain socket *and* a binary framed transport over a Unix domain socket, the latter with peer-credential authentication and backpressure/draining for high-frequency, tight-loop coordination. We note in §13 that the peer-credential check is a software handshake — the connecting process *states* an agent identity and the daemon checks it against the registration table, backed by owner-only (`0600`) permissions on the socket file — and *not* the kernel-enforced socket-level credential check (`SO_PEERCRED`) it resembles and is named after. That is a real trust-boundary caveat, it holds for both transports, and it is unchanged by the hardening design of §13.4.

Exercises 1.13–1.15, page 42.

8 The obligation & enforcement organ: the sovereign's two arms

This organ is the deontic core, and it is genuinely two distinct mechanisms — in the vocabulary of deontic logic for computer systems (Jones and Sergot [8]): *prohibition* (a policy monitor) and *obligation* (commitments). Getting the distinction right — and being honest about how strong each arm actually is — is the security heart of the paper.

Table 5 lays out the three modalities and the mechanism behind each.

Table 5: The deontic split: three modalities, three mechanisms. Capability (what the sandbox makes possible) is kept apart from permission (what the deontic layer allows) so that "able but forbidden" stays expressible.

Modality	Trigger	Mechanism	Holds / fails when	Example
prohibition	an act the policy forbids is attempted	regimented before commit by a gate when the event is controllable; otherwise detected after commit and compensated	holds when the effect never commits; fails as detect-and-compensate when the gate could not see the trigger	a tool's emergency override flag: the agent is *able* to call it and *forbidden* to
obligation	a commitment is recorded	an oracle-bound commitment: closure only through a receipt the daemon can re-check, before a daemon-set deadline	holds when a typed receipt closes it in time; fails by staying open and escalating, never by free-text assertion	"release the claim on `lib/webhooks.ts` within thirty minutes"
permission	a grant is issued	a recorded capability row, attenuable, never widened downstream	holds while the grant is live; an act without a grant is a prohibition case	"may read the audit log until the session ends"

8.1 The deontic split

Definition 8.1 (The kernel's deontic split)**.** The kernel realizes three deontic modalities with three mechanisms: *prohibition* is regimented (rejected pre-commit) or enforced (detected post-commit); *obligation* is enforced by oracle-bound commitments; *permission* is a recorded capability. The coordination layer's deontic binding is a direct lift of this split.

We keep **capability** (*ability* — what the agent's sandbox makes possible) distinct from **permission** (*normative status* — what the deontic layer allows), so that "able but forbidden" stays expressible. The canonical example is a dangerous override flag that a tool exposes for emergencies: it *exists* (an agent is *capable* of invoking it) but *must not be used* (it is *forbidden*). Collapsing capability into permission makes that state inexpressible — which is exactly the bug a discipline against silently bypassing guardrails cannot afford.

8.2 The policy monitor is a detector, not a regimenter

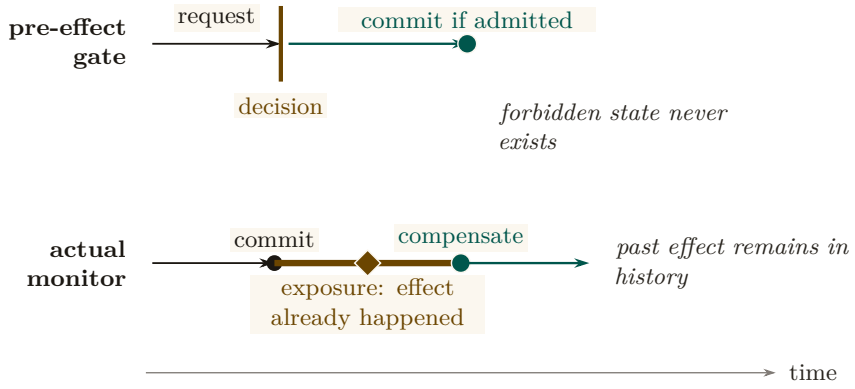


Figure 6: The reference-monitor test as a causal counterfactual. A true mediator decides before the commit point, so a forbidden state never exists. The present policy monitor instead subscribes after commit: the caution segment is the exposure interval in which the effect has already happened, before it can observe, halt subsequent work, and compensate. Only uniqueness constraints and fail-closed boot gates belong to the upper trace; subscriber rules belong to the lower one. [internal]

Figure 6 draws the correction this section makes precise. Here is the correction that the rest of this stack must adopt. The most seductive claim one can make about a coordination policy monitor — that it "makes forbidden states physically unreachable" — is false as stated. The monitor is a *subscriber to an append-only event stream of operations that have already committed*. By the time it observes a violation, the offending write is already durable in the log. Even in its strictest mode, its strongest response is a *compensating* undo, not a prevention. This is not an implementation accident; it is a theorem.

Theorem 8.1 (A prevention bound for post-commit monitoring)**.** A monitor invoked only after a transition commits cannot prevent a safety violation whose bad prefix ends at that transition. It can detect the prefix, halt later actions, and enforce a separate recovery or bounded-response property. Therefore a post-commit subscription must not be credited with making the triggering state unreachable.

Proof sketch. Schneider's execution-monitor model [4] recognizes safety properties through finite bad prefixes and enforces them by suppressing an action before that prefix enters the target execution. A subscriber invoked after commit cannot suppress the committing action. Compensation may establish a different eventual or bounded-recovery property, but it does not retroactively remove the bad prefix. □

Worked dramatization: Bob attempts a forbidden action. Bob is an agent under a policy that forbids editing files outside his assigned task scope. He nonetheless issues a write that records an out-of-scope edit. *What happens?* The write commits. The single writer serializes it, the uniqueness constraint has no objection (it is a well-formed row), and the operation lands durably in the log. Only *then* does the policy monitor, subscribed to that log, observe the out-of-scope edit and raise a violation. Its response is compensating: it can flag Bob's session, revoke his claims, alert the operator, and trigger a salvage of the affected surface — but it cannot pretend the write never happened, because it did. Contrast this with Bob trying to claim a port Alice already holds (the previous dramatization): *that* attempt never commits, because the uniqueness constraint rejects it before the commit line. The two cases are the whole deontic story in miniature: a uniqueness constraint and a boot-time admission gate *regiment* (the bad state is unreachable); the policy monitor *enforces* (the bad state is reached, then detected and compensated within a bounded window).

Key idea. The only genuine pre-commit regimentation is credited to the right mechanism: the uniqueness constraint (no two holders of one resource key; no duplicate process-identity squat) and the fail-closed boot-admission gate (no serving from a test database; refuse to serve when a critical self-check fails). Those reject *before* the commit line and are *truly* unreachable. The policy monitor notices everything else *afterward* and compensates. So: "physically unreachable" for the regimented set; "detected within a bounded window, then halted or compensated" for the rest. Crediting double-claim prevention to the monitor *double-counts* — the uniqueness constraint already did it; the monitor merely logs it.

Theorem 8.2 (Synchronous State Decidability — closing OP-2)**.** A safety invariant ϕ is **regimentable** (strictly preventable pre-commit, making violation states physically unreachable) if and only if $\phi(S, \Delta)$ is a pure, deterministic boolean predicate decidable solely over the daemon's local, synchronous, committed database state S and the incoming transaction payload Δ.

Conversely, any invariant requiring external I/O, asynchronous wall-clock verification, distributed consensus, or non-deterministic grading oracles is strictly **enforceable** (post-commit, detect-and-compensate).

Proof sketch. In SQLite WAL mode under single-writer serialization, transaction commits occur synchronously on the database connection thread. A predicate $\phi(S, \Delta)$ computable in bounded CPU steps without yielding the event loop can be embedded directly into SQLite constraints (`UNIQUE`, `CHECK`) or transaction guard clauses; any transition yielding $\neg\phi$ rolls back atomically before state mutation. If ϕ depends on external network I/O or asynchronous delays, holding the exclusive transaction lock during evaluation starves the single-writer queue; hence evaluation must be deferred post-commit, converting the mechanism from a synchronous gate into an asynchronous event subscriber governed by Theorem 8.1. □

8.3 The general boundary: regimentation is controllability

Express lane: a governance rule can be enforced by prevention exactly when no event the runtime cannot refuse can carry a legal history out of the rule; that is Ramadge–Wonham controllability with the model's own steps as the uncontrollable alphabet. The formal statement is the box below; the machine-checked policy table is Table 6.[1]

The scene. The operations lead reads the governance document aloud. *No force-push to main. No network egress from sandboxed jobs. No writes outside the granted scope.* And then: *no confident falsehoods in status reports.* She asks the question every practitioner senses the answer to: which of these does the runtime *guarantee*, and which does it merely *watch for*? The first three feel airtight. The daemon holds the remote credential, so a push that never happens needs no forgiveness. The fourth is not airtight and never will be: the falsehood is composed inside the model's forward pass, and by the time any mediator sees tokens the falsehood already exists. Theorem 8.2 settled this for one door, the database commit. This section settles it for every door a runtime can stand at, and the instinct turns out to be an instance of a theorem from 1987.

The idea in one breath. *Split the agent's events into what the runtime can refuse and what it cannot; a safety rule is preventable if and only if no unrefusable event can carry a legal history out of the rule.*

Intuition: the bouncer at the one door. A club has one door and a bouncer. The bouncer can refuse *entry*: entry is mediated by the door he stands at. He cannot refuse a patron's

[1] A standalone, submission-form version of this section is *Regimented or Enforced: The Controllability Boundary for Agent Governance* (research paper 2), whose policy table and verdicts regenerate from `skills/harbor-results/scripts/b3_controllability.py`.

thought: nothing the club owns mediates it. A door policy ("no entry after two") is enforceable by prevention; a thought policy ("no ill intent inside") is enforceable only by watching and, if it comes to it, walking someone out. The daemon's effect boundary is the door. Writes, egress, tool execution, pushes, spawns, and API calls cross it, so each can be refused; token emission, in-context reads, plan formation, and hidden activations complete inside the model, so none can. The analogy maps relations rather than surface features, which is why it earns a prediction: a *compound* rule that mentions thought but gates only the door ("no entry for anyone who arrived after the fight started") should still be preventable, because the bouncer need only *observe* the trigger and *refuse* the door. Section 8.3.1 confirms exactly this. The predictable misread, preempted: uncontrollable does not mean invisible. Tokens are eminently loggable after emission. Uncontrollable means *unrefusable before the fact*, not unobservable after it; when an event is also unobservable a second, stricter theory applies (§8.3.2).

The vocabulary, at the point of use. Everything below is a statement about strings and sets of strings. An *alphabet* Σ is the finite set of event types the system can emit: `fs_write`, `net_egress`, `model_emit_token`, and the rest. A *string* over Σ is one possible history of an execution. A set of strings K is *prefix-closed* when every prefix of a member is a member: if the first ten events of a run are allowed, so are the first nine. Prefix-closed languages are exactly Lamport's safety properties: a violation has a finite witness, and once you have violated one you can never un-violate it. The *plant* L is the set of histories that are physically possible before any policy is imposed; we take $L = \Sigma^*$, since nothing about the hardware forbids an interleaving. A *policy* is a prefix-closed $K \subseteq L$. A *supervisor* is a function from the history so far to a set of events to disable next, and the whole theorem turns on its type: it may disable members of Σ_c, the *controllable* events that pass through a mediation boundary before taking effect, and nothing else. The *closed loop* is the set of histories that occur once the supervisor is attached. A supervisor *regiments* K when the closed loop equals K exactly: nothing forbidden happens and nothing permitted is lost. That second half is half the value. A supervisor that achieves safety by disabling everything is not a solution.

Theorem 8.3 Regimentation is controllability. Let $L \subseteq \Sigma^*$ be a prefix-closed plant over $\Sigma = \Sigma_c \uplus \Sigma_u$ and let $K \subseteq L$ be a nonempty prefix-closed safety policy. There exists a regimentation mechanism for K, a supervisor disabling only controllable events whose closed loop is exactly $\overline{K}$, if and only if K is *controllable* with respect to L and Σ_u:

$$\overline{K}\,\Sigma_u \,\cap\, \overline{L} \;\subseteq\; \overline{K},$$

that is, no uncontrollable event, physically enabled after a legal history, exits the policy. When the condition fails, no runtime confined to disabling Σ_c can prevent the violation; the best achievable enforcement is detect-and-compensate, and the largest preventable weakening of the policy is the supremal controllable sublanguage $\sup\mathcal{C}(K)$, which exists and is computable for regular K and L [verified framework: Ramadge and Wonham [6]].

Proof idea. If the policy is controllable, build the laziest possible supervisor: after any legal history, disable exactly the controllable events that would step outside the policy. It never has to disable an uncontrollable event, because controllability says none of those can step outside. If the policy is not controllable, there is a legal history after which the plant can fire an uncontrollable event that leaves the policy; any mechanism that keeps every legal history reachable must let that history happen, and then cannot stop the event.

Proof. The proof is structured; each numbered step can be checked on its own.

⟨1⟩1. ASSUME K controllable. DEFINE the supervisor $f(s) = \{a \in \Sigma_c : sa \in \overline{L} \setminus \overline{K}\}$ for $s \in \overline{K}$. PROVE the closed loop equals $\overline{K}$.

⟨2⟩1. The closed loop is contained in $\overline{K}$, by induction on history length. The empty history lies in $\overline{K}$ because K is nonempty and prefix-closed. If the current history s lies in $\overline{K}$ and the loop executes a, then $sa \in \overline{L}$ and $a \notin f(s)$. If $a \in \Sigma_c$, then $a \notin f(s)$ means $sa \in \overline{K}$ by the definition of f. If $a \in \Sigma_u$, then $sa \in \overline{K}\Sigma_u \cap \overline{L} \subseteq \overline{K}$ by controllability.

⟨2⟩2. Every $sa \in \overline{K}$ is reachable: an uncontrollable a is never disabled, and a controllable a with $sa \in \overline{K}$ is not in $f(s)$.

⟨2⟩3. Q.E.D. Containment both ways is equality, so f regiments K.

⟨1⟩2. ASSUME K not controllable: there are $s \in \overline{K}$ and $\sigma \in \Sigma_u$ with $s\sigma \in \overline{L}$ and $s\sigma \notin \overline{K}$. PROVE no regimentation mechanism exists.

⟨2⟩1. Let M be any regimentation mechanism for K. Because M realizes exactly $\overline{K}$, the legal history s is reachable under M.

⟨2⟩2. At s the plant enables σ, and M cannot disable it: $\sigma \notin \Sigma_c$, so disabling it is outside M's type. Physically, the event completes before the mediation boundary sees it.

⟨2⟩3. Q.E.D. The closed loop under M reaches $s\sigma \notin \overline{K}$, contradicting the assumption that M's closed loop is contained in $\overline{K}$. Any enforcement of K must therefore tolerate the reachable violation $s\sigma$ and respond to it afterwards, which is detect-and-compensate by definition.

⟨1⟩3. The existence, uniqueness, and computability of $\sup \mathcal{C}(K)$ are imported from Ramadge and Wonham [6]: controllable sublanguages of K are closed under arbitrary union, so a supremal element exists, and for regular languages a fixpoint iteration on the product automaton computes it.

⟨1⟩4. Q.E.D.

□

Two remarks on what the proof used. The forward direction is constructive and *maximally permissive*: the supervisor disables nothing except the controllable exits, so regimentation costs no legal behavior. The governance rule and the agent's freedom are not in tension when the rule is controllable at all. The reverse direction used nothing about the mechanism except its type. No cleverness inside the runtime, no better prompt, faster monitor, or smarter policy engine, escapes the theorem, because the obstruction is that σ finishes before any decision point the runtime owns. Theorem 8.2 is the special case $\Sigma_c = \{$synchronous database commits$\}$: when the only mediated channel is the commit, the regimentable policies are exactly those expressible over committed state. The general statement strictly dominates it. The slogan cannot say why a force-push is preventable by a runtime that also owns the credential channel, nor why a confident falsehood stays unpreventable however much state is committed.

Numbers by hand — The policy table, checked by hand. The controllability test is decidable by a product-automaton search: walk the synchronous product of plant and policy automaton and flag any reachable state where the plant enables an uncontrollable event that the policy disables. Run over the reference alphabet, the checker returns Table 6 [internal, `b3_controllability.py`; deterministic, no seed]. The eight single-event rows also follow by hand, which is the right way to first believe the table. Take "forbid `git_push`." The policy automaton has one state with every event except `git_push` self-looping; `git_push` lies in Σ_c, so the only event the policy ever disables is controllable, the premise of a controllability violation is unsatisfiable, and K is controllable: regimentable. Now take "forbid `internal_plan`." The same one-state automaton

disables `internal_plan`, which lies in Σ_u; the universal plant enables it at the start state, so the empty history and $\sigma =$ `internal_plan` witness $\overline{K}\Sigma_u \cap \overline{L} \not\subseteq \overline{K}$: detect-only, for any runtime with this alphabet split. *Now you try:* classify "forbid `api_call` after `spawn_child`" and "forbid `internal_plan` after `fs_write`" (Exercise 1.19).

Table 6: Nine governance rules classified by the product-automaton checker. The verdicts are the checker's output; the eight single-event rows also follow by hand from Theorem 8.3 [internal, `b3_controllability.py`].

Policy	Event class	Regimentable?
forbid `net_egress`	controllable	yes (regiment)
forbid `git_push` (incl. force-push)	controllable	yes (regiment)
forbid `fs_write` (out-of-scope write)	controllable	yes (regiment)
forbid `exec_tool`	controllable	yes (regiment)
forbid `spawn_child`	controllable	yes (regiment)
forbid `model_emit_token`	uncontrollable	no (detect only)
forbid `in_context_read`	uncontrollable	no (detect only)
forbid `internal_plan` ("confident lie")	uncontrollable	no (detect only)
compound: no `net_egress` after a secret `in_context_read`	mixed	yes (regiment)

8.3.1 The compound case: gate the channel, never the token

The last row of Table 6 is the one an entire product line rests on. The policy *no network egress after an in-context read of a secret* mentions an uncontrollable event, so a naive reading of the single-event rows predicts detect-only. The checker says otherwise, and the mechanism is visible in the policy automaton: two states, `ok` and `tainted`. The uncontrollable read is enabled in both. The policy never tries to forbid it; it lets the read happen and records the taint transition. What the tainted state disables is `net_egress` alone, and egress is controllable. Every uncontrollable event the plant enables is enabled by the policy in every reachable state, so the criterion holds and the constructive direction of the theorem hands us the supervisor: track taint, gate the channel. This automaton is Schneider's own Figure 1, a two-state security automaton for "no `Send` after a `FileRead`" [4], with his states relabelled. What the alphabet split adds is not the construction but the grading: under complete mediation the figure is one more enforceable safety policy; once $\Sigma_u \neq \emptyset$ the same two states become the canonical witness that a policy may reference an unrefusable event freely so long as it never tries to refuse it.

Three consequences, in increasing order of consequence.

1. **The design rule.** A compound trigger-to-effect policy is regimentable if and only if its *effect* events are controllable; the trigger may be as uncontrollable as it likes, provided the policy observes rather than forbids it. Uncontrollable events in the policy's condition are free; uncontrollable events in its prohibition are fatal.
2. **The sealed room.** Put an agent alone in a room with a customer's secret and promise that the secret does not leave. It is impossible to prevent the model from *reading* the secret (the read is uncontrollable, and useless to forbid, since reading is the job) and impossible to prevent it from *incorporating* the secret into tokens and plans (uncontrollable again). What is possible, exactly and only, is to gate every controllable egress channel out of the room, with taint recorded from the moment of exposure. Gate the channel, never the token. The companion noninterference result for the sealed room proves from the other direction that the gate suffices; controllability says the gate is the only place a guarantee *can* live. This is why "all spend is recorded" in the economy chapters is precisely complete mediation of Σ_c.

3. **The two dials.** Holding the plant fixed, widening Σ_c (owning more channels) is the only way to grow the regimentable set; no policy-language cleverness does. The plant is the other dial: $\overline{L}$ sits on the left of the containment, so shrinking it weakens the antecedent and strictly more policies pass. Both detect-only verdicts above turn on "the universal plant enables `internal_plan`," and are conditional on the modelling choice $L = \Sigma^*$. An event that never becomes physically possible needs no supervisor, which is why data minimization and capability removal are governance moves and not merely hygiene: a secret never loaded into the room regiments the compound policy with no gate at all.

Numbers by hand — Synthesis by hand: the sandboxed review job. The theorem's constructive direction is the synthesis algorithm the Supremica and TCT tool families implement, and it is small enough to run on paper. A sandboxed code-review job carries three rules: P_1, no `net_egress`, ever; P_2, no `git_push`, ever; P_3, after any `in_context_read` of customer material, no `fs_write`. P_1 and P_2 are one-state automata and P_3 is the two-state taint automaton, so the product specification has $1 \times 1 \times 2 = 2$ states, `ok` and `tainted`. The controllability check visits both states and tests each of the four uncontrollable events in each: all eight (state, event) pairs are enabled by the specification, so there are 0 violations and the composite is controllable [verified: eight cases, checkable by eye]. The maximally permissive supervisor falls out of the proof as a two-row disablement map: in `ok`, disable {`net_egress`, `git_push`}; in `tainted`, disable {`net_egress`, `git_push`, `fs_write`}; one observed transition flips `ok` to `tainted`. That table is exactly the policy engine a competent engineer would have written by hand. The value of synthesis is not surprise; it is that the answer arrives with a maximal-permissiveness certificate and a controllability certificate, neither of which hand-written engines carry.

Synthesis is equally informative when it fails. Ask for the best preventable approximation of "forbid `internal_plan`": any nonempty controllable sublanguage K' contains the empty history, the plant enables `internal_plan` there, so controllability forces `internal_plan` $\in \overline{K'} \subseteq \overline{K}$, contradicting that K forbids it. Hence $\sup \mathcal{C}(K) = \emptyset$ [verified: three lines]. There is no nontrivial preventable weakening of a thought ban, and a governance document that claims to "partially enforce" such a rule by prevention is describing detection with optimistic vocabulary. *Now you try:* add P_4, "no `spawn_child` after `git_push`," to the review job and write the new disablement map (Exercise 1.20).

8.3.2 Partial observation: the second boundary

Theorem 8.3 assumes the supervisor *observes* every event, even the ones it cannot disable: the daemon logs tokens after emission and reads after completion. Realistic deployments break this in a second place. Some events are not only unrefusable but unobservable: activations never surfaced, state internal to a third-party tool, another principal's context. Supervisory control under partial observation is also solved territory. Lin and Wonham [5] characterize enforceability as controllability *plus observability*: the supervisor's decisions must be constant on observation-equivalence classes of histories, so the policy may not demand different gate decisions after histories the runtime cannot tell apart. Observability strictly shrinks the regimentable set. "No egress after the model *considered* exfiltration" has a controllable prohibition and an unwitnessed trigger, so no supervisor can implement its gate condition. The practical reading: a compound policy is deployable only if its trigger is witnessed at the mediation boundary. Taint must be an event the daemon sees, not a state it infers. Figure 7 puts both boundaries on one grid.

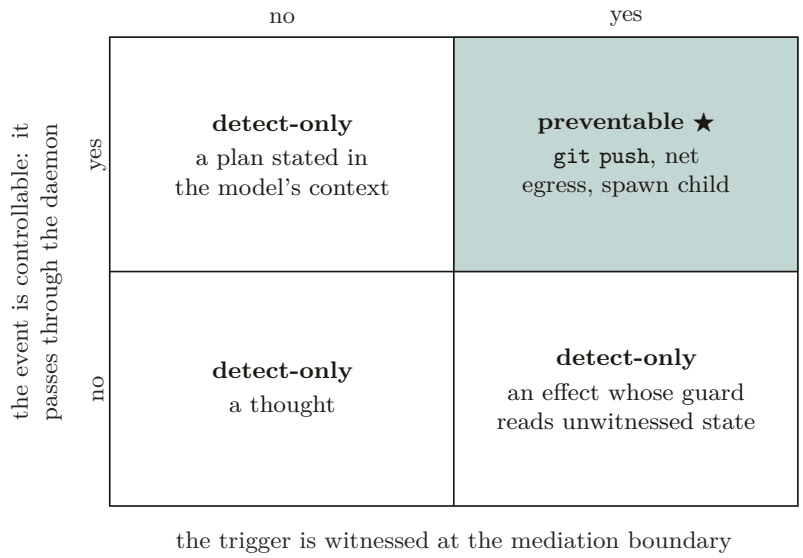


Figure 7: Preventable is one cell, not a spectrum. An event is preventable exactly when it is controllable (top row) *and* its trigger is witnessed at the mediation boundary (right column): the shaded cell, marked ★ and tied to Theorem 8.3. Every other combination is detect-only. Column membership reproduces from the checker [internal, `b3_controllability.py`]; row membership is by inspection of each trigger.

What it buys. The chapter's own vocabulary. *Regimented* and *enforced* (Definition 8.1) are now the two sides of one inequality rather than two adjectives; every row of Table 7 marked "detected, not regimented" is a row whose prohibition names an uncontrollable event or an unwitnessed trigger. Downstream it buys the assurance modes the rest of the book uses to describe what a runtime can see and control, the design rule behind the sealed room, and the pricing stance of the market chapters: detect-only policies are enforced by audit, reputation, and bond-slashing, which must be sized for violations that *will* occur.

> **Where this stops.** (i) The theorem does not say detect-only policies are hopeless. Detection feeding audit, reputation, and slashing is real enforcement; the theorem says only that prevention is off the table. (ii) It certifies no implementation. Its Σ_c is the set of events *actually* mediated, and every unenumerated side channel (raw sockets, DNS, inherited file descriptors, `/proc`, credential helpers, package managers, provider callbacks) silently moves events from Σ_c to Σ_u and shrinks the regimentable set behind the operator's back. Complete mediation of the claimed alphabet is an architectural obligation (Anderson; Rushby) that a red team, not this proof, discharges. (iii) The classification is relative to the alphabet split: a runtime that gates model steps changes Σ_u and re-grades the table. The theorem is a functor from architectures to boundaries, not one fixed verdict. (iv) Full observation is load-bearing: policies whose triggers are unwitnessed internal state fall to the Lin–Wonham refinement even when their prohibitions are controllable. (v) The checker verifies the stated automata over the stated alphabet; it is a faithful implementation of the test, not a mechanized proof of the theorem. An Isabelle or Coq formalization remains open. (vi) Prevention governs effects, never semantics. For "no confident falsehoods in status reports" the obstruction is observability: a report leaves through emission and then through `api_call`, `fs_write`, or `net_egress`, all controllable, so the prohibition can be made controllable; but "this report is false" is not a predicate over the event alphabet, so the gate condition is unwitnessed at every alphabet split. The rule falls to (iv), not to the box's inequality.

Recall.

1. Without looking: which two sets does the controllability criterion mention besides the policy, and which of them is a dial the operator can turn?

2. State the design rule for compound policies in one sentence.
3. Why does the confident-falsehood rule stay detect-only even for a runtime that gates every effect channel?

The monitor is honest about its own coverage, which is itself a genuine security property: it reports each rule as *enforced*, *degraded*, or *stubbed*, so a reader can never mistake a stubbed rule for an active one. We note in §13 that one rule's enforcement (capability escalation) is backed by a separate native component whose evidence is the weakest in the organ, carrying a time-of-check-to-time-of-use surface.

8.4 Commitments: obligation with oracle-bound closure

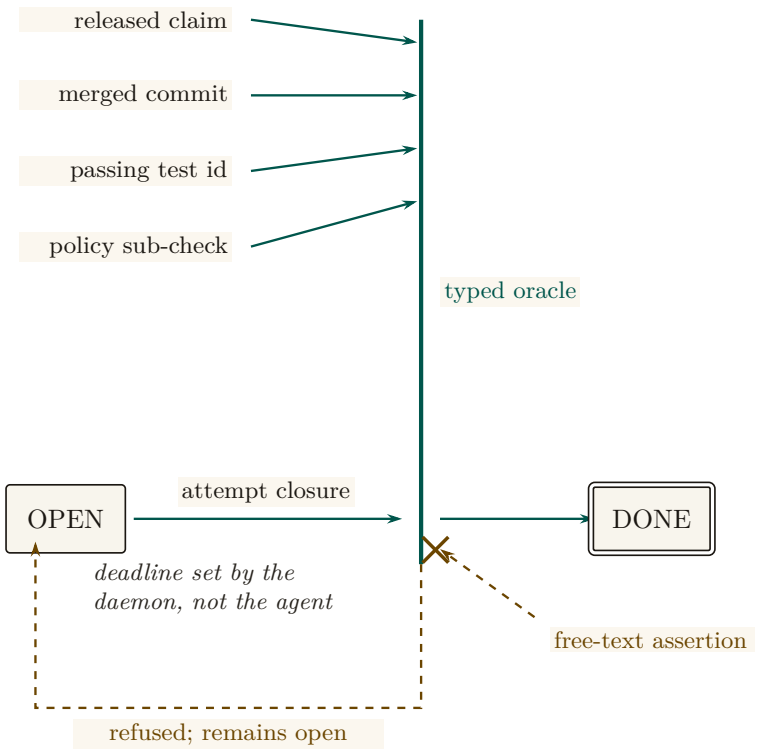


Figure 8: Typed evidence passes through a narrow closure aperture. A commitment closes as done only when the daemon can re-run a check from the finite oracle vocabulary: a released claim, merged commit, passing test id, or satisfied policy sub-check. A free-text assertion is refused and the commitment remains open. The deadline is set by the daemon, not the agent. [internal]

Commitments are durable, *violable* promises — in the sense of Cohen and Levesque's "intention as a persistent goal" [17], with both single-minded and open-minded strategies for when a promise may be dropped. Two hardening laws are implemented, and they are the ones that determine safety (Figure 8).

Property 8.1 (Metric-control separation, Law 1)**.** A commitment's deadline is derived by the daemon from a scope policy and is *never* agent-supplied. The agent picks the work; the daemon picks the deadline. (Invariant I5.) The point is to deny the agent control of the very metric it is judged against. This removes one direct manipulation channel; it is not a general proof of resistance to proxy gaming or misspecified scope policy.

Property 8.2 (Oracle-bound closure, Law 2)**.** Closing a commitment as "done" is refused unless the agent supplies a non-empty reference drawn from a *finite oracle vocabulary*: a released claim, a merged commit hash, a passing test identifier, or a satisfied policy sub-check. Free-text notes cannot close a commitment. (Invariant I6.)

The closure gate is small enough to state as a protocol (Algorithm 2). The crux is the final clause: the supplied reference must not only be drawn from the vocabulary but must *currently hold* when the daemon re-checks it, so an agent cannot name a test that passed yesterday and has since regressed.

```
function close(commitment, status, oracle_ref):
  if status != 'done':
    return apply_drop_strategy(commitment, status)   // abandon / supersede
  if oracle_ref is empty:
    return REFUSED("done requires a verifiable oracle reference")
  kind = classify(oracle_ref)                        // structured field, not NLP
  // the four kinds below are the SHIPPED vocabulary. State-Machine Assertions
  // (OP-5, specified only) would add DELTA_ORACLE and AST_ASSERTION here.
  if kind not in {RELEASED_CLAIM, MERGED_COMMIT, PASSING_TEST, POLICY_SUBCHECK}:
    return REFUSED("oracle reference outside the finite vocabulary")
  if not verify_holds_now(kind, oracle_ref):         // re-check at closure time
    return REFUSED("named oracle does not currently hold")
  transition(commitment, DONE, witnessed_by = oracle_ref)
  return CLOSED
```

Listing 2: The commitment-closure oracle. Closure requires a reference from a finite vocabulary that the daemon re-verifies at closure time; free text is rejected by construction, not by inspection.

Key idea. That finite oracle vocabulary *is* the kernel's honesty boundary. It can mechanically verify "the test you named passed," never "the refactor is good." Naming the oracle set is naming the limit of what the substrate layer can verify — and open problem OP-5 asks which real "done" conditions it fails to capture, and whether the set can be extended without re-opening the Goodhart hole that free-text closure would.

8.5 Extending the oracle vocabulary: State-Machine Assertions (OP-5)

The obvious way to widen what a commitment can close against is to widen the vocabulary, and there is a concrete design for doing so without re-admitting the Goodhart hole. **State-Machine Assertions** (SPECIFIED) would add two kinds to Listing 2's finite set: *delta oracles*, which close against a machine-checkable before/after measurement (a named benchmark improving past a declared threshold), and *AST assertions*, which use a parser such as `tree-sitter` to state a structural property of the resulting code ("no call site of f remains") that the daemon can re-evaluate at closure time.

Both kinds satisfy the property that matters — the daemon can re-check them itself, so they are oracles and not testimony. Neither is in the shipped set: Listing 2's four kinds are the vocabulary the kernel actually enforces, and the listing marks the two proposed additions as exactly that. Two questions also stand between the design and the claim, which is why we do not make it. A delta oracle inherits the noise of whatever it measures, so "improved by a threshold" is a statistical statement dressed as a boolean; and an AST assertion is a proxy for a structural intent, which is the shape of every target that has ever been gamed. *Oracle completeness* — the claim that some finite vocabulary captures the "done" conditions agents actually need — is not established by adding two kinds, and OP-5 stays *open*.

Exercises 1.16–1.21, page 42.

9 The continuity organ: memory, checkpoint, and the foundation of the economy

The through-line of the whole series — memory → checkpoint → continuity → durable identity → reputation → market — *bottoms out here.* If the substrate's continuity is weak, the cross-machine economy is built on sand. This is the section the personhood and market papers lean on hardest, so it carries the canonical statement of the layer's own softest link.

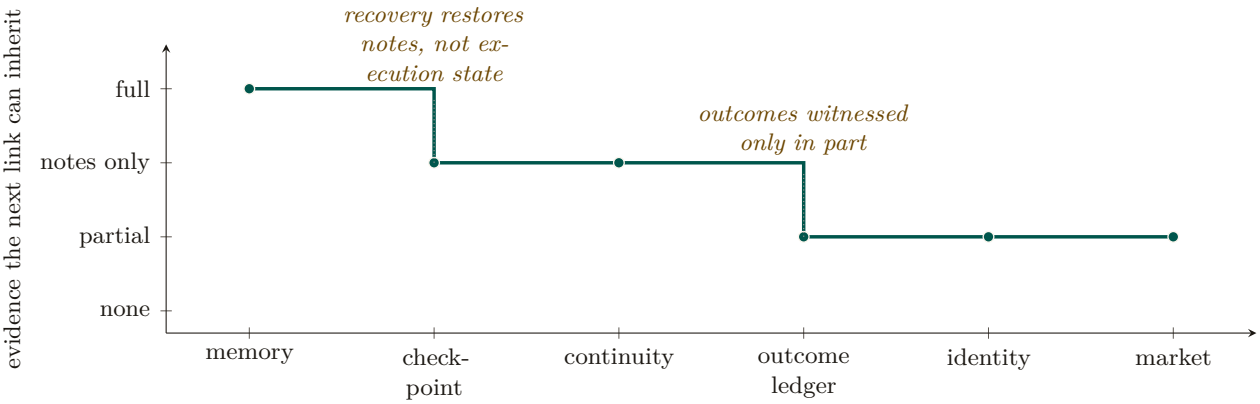


Figure 9: The continuity chain as an evidence-support profile. What checkpoint inherits from memory already drops from full to notes only; what the outcome ledger inherits from continuity drops again, to partial; identity and market inherit only that partial residue. Downstream confidence cannot exceed the weakest upstream link. [internal]

9.1 Three organs, one weak link

Figure 9 names the three continuity organs and states the canonical sentence the rest of the series depends on:

> *The economy rests on three continuity organs — memory (IMPLEMENTED), checkpoint (PARTIAL: notes, not execution state), and a witnessed-outcome ledger (PARTIAL: commitments and oracle-bound closure, not neutral graded outcomes) — and reputation keys on the richer third organ, which does not yet exist; the checkpoint organ is the weakest continuity link, and a checkpoint with teeth (a real execution-state snapshot) is the literal foundation of the cross-machine economy.*

Memory — session records, durable notes, and the operation log — is **implemented**, and the kernel even forbids amnesia: a monitor rule guards that an active session's durable note count never regresses (detect-and-compensate, per Theorem 8.1, but real).

Checkpoint is PARTIAL, and the weakness is precise. The recovery pipeline runs a heartbeat watchdog → stale/dead detection → a recovery queue → a salvage handoff, and it *passes notes*; it does not checkpoint execution state. A checkpoint with teeth — the gap between passing notes and restoring work — is open problem OP-4, the most important unbuilt thing at this layer because it is the literal foundation of any reputation economy.

The most promising direction we have for it is worth naming, at exactly its strength. **Event-sourced rehydration** (SPECIFIED) would treat the successor's starting context as a replay rather than a summary: pin the working tree to the predecessor's Git commit, truncate the durable message log to the recorded failure point, and replay that log into a fresh agent so its prompt prefix is reconstructed from the record instead of paraphrased from it. What this

restores is the *durable* half — a commit identifier, a message log, an open-claim and commitment set — and that half is genuinely worth having. What it does not establish is the half OP-4 is named for. A language model's inference state is bound to a particular model, sampler, and session; a replayed prefix reconstructs the *inputs* to that state, not the state, and the reference implementation ships none of this today. Whether replaying the inputs is close enough to count as "restoring work," and against what test, is precisely what remains open — so we report the mechanism as a direction with an unestablished completeness claim, not as a closure of OP-4. The companion personhood chapter reaches the same boundary from the identity side: a continuity witness attests that the ledger followed the right body, never that the successor inherited the predecessor's experience.

The witnessed-outcome ledger — the durable record of what an agent actually delivered — is the organ on which reputation keys. A PARTIAL substrate now ships: durable commitments, daemon-derived deadlines, oracle-bound closure, API/CLI surfaces, and overdue detection. Neutral graded outcome events, sanctions, identity binding, and reputation updates remain absent. OP-4 (here) and that richer outcome-ledger gap (the personhood paper) are the *same* finding viewed from two ends.

9.2 The operation log and tamper-evidence

The append-only operation log is the kernel's audit organ — the stream the policy monitor subscribes to and the thing that makes "what actually happened" answerable. Hash-chained tamper-evidence over it — the digital-timestamping pattern of Haber and Stornetta [23] that predates blockchains, itself building on Merkle's hash trees [22] — exists but is re-scoped (see §13.4).

Exercises 1.22–1.24, page 43.

10 The self-attestation organ: honest green

Honest self-attestation belongs to the substrate layer, because the kernel is the one component that can formally report its own integrity: it is always-on and owns the only writable truth. The attestation routine evaluates a registry of invariants, each returning *pass*, *fail*, *skipped-with-reason*, or *unknown*. The report is *scoped and conjunctive*: "all good" is printed only when every checked invariant passes, and the report always enumerates what was *not* checkable.

Property 10.1 (Honest self-report, I10)**.** The attestation reports "all good" only if every checked critical invariant passed, and it always lists the unchecked set. There are three triggers: a boot-admission gate (refuse to serve on a critical failure), a command pre-flight (loud refusal that names the fix), and a continuous watchdog (a pass-to-fail transition deposits a coordination marker and a notification). **implemented**.

This is the honesty discipline of the rest of the series turned on the kernel itself, and it is the right posture for a trusted computing base: when integrity is unknown, deny. The drift and liveness self-checks exist precisely because a second copy of the daemon *can* come up against the same file — a packaged install lagging the development tree is a recurring hazard — and the kernel must know which copy of itself is the real one.

11 The invariants, stated as theorems

A completionist treatment names the kernel's properties as falsifiable claims, each with its mechanism and its maturity grade. Table 7 is the formal spine of the layer; the grade column is the maturity discipline turned on the theorems themselves.

Table 7: The kernel invariants. I1 is split by fault class (I1a/I1b) per §5; I7–I9 carry the regimentation/detection correction per §8.

#	Invariant (theorem)	Mechanism	Maturity
I1a	Process-crash durability	WAL + OS page cache	**implemented**
I1b	Power-loss durability	(needs full synchronization)	*not guaranteed*
I2	Mutual exclusion (1 holder/key)	uniqueness constraint + busy-wait	**implemented**
I3	Idempotence of re-claim/re-release	upsert / delete-if-exists	**implemented**
I4	Liveness reclamation (TTL sweep)	lazy expiry on read + ticks	**implemented**
I5	Metric-control separation (Law 1)	daemon-derived deadline	**implemented**
I6	Oracle-bound closure (Law 2)	finite oracle vocabulary	**implemented**
I7	No-amnesia (note monotonicity)	monitor: note-monotonicity rule	PARTIAL (detected, not regimented)
I8	Forbidden-state handling	constraint/boot (regimented) + monitor (detected)	PARTIAL (mostly detect-and-compensate)
I9	Tamper-evidence of the audit log	hash chain	PARTIAL (re-scoped to the economy layer, §13.4)
I10	Honest self-report	scoped attestation report	**implemented**
I11	Runtime parity (interpreter ≡ test bindings)	runtime-shim layer	PARTIAL (OP-3)
I12	Non-forgeable identity	actor-soul id + lookup credential	PARTIAL (bounded gate ships; universal write gating absent)

11.1 The work unit, model-checked

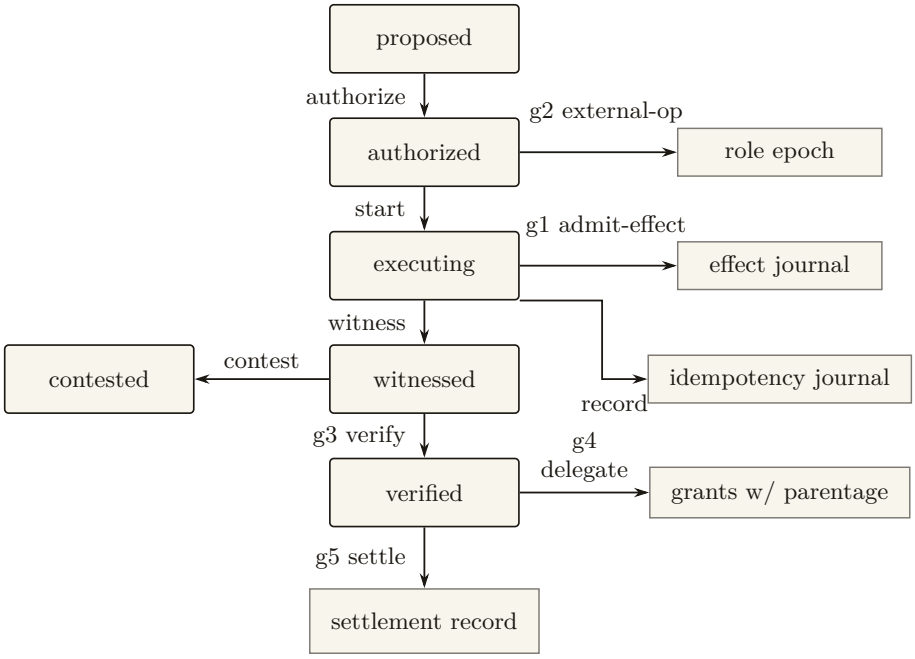
Express lane: the kernel's promises are properties of one object, an evidence-bearing work unit; on a two-principal instance every one of its 536 reachable states satisfies six invariants, and removing any single guard lets a checker construct the crime in at most seven steps. The machine is Figure 10 and the box below; the numbers are Table 8.

The scene. An agent dies mid-task. Its successor, possibly a different model from a different vendor, picks up the work and later says "done." Who checks the paperwork? Where is the record that the port it held was released, that the payment moved once and not twice, that "verified" meant an inspection actually happened rather than a mood? Every organ in this chapter answers a piece of that question. The work unit is the object that holds the pieces together: a case file that outlives every clerk who touches it.

The idea in one breath. *Make the unit of trust a work unit whose every consequential step passes a guard, then prove, by exhausting the machine's reachable states, that its six promises can never be broken, and that every guard is load-bearing, because removing any one lets the checker construct the crime.*

Intuition: the notarized escrow folder. Nothing enters the folder without the right stamp: an active grant carrying the *current* role epoch, so a clerk who was replaced yesterday finds that today's stamp rejects him. No payment leaves twice: each external operation carries an idempotency key, honored at most once. And "verified" is not a feeling: it means the acceptance policy and the inspector's receipt are physically in the folder. Then we do something a paper

folder cannot: enumerate every state the machine can ever reach and check all six promises in each. The misread to preempt: a model checker does not prove the daemon correct. It proves the *specification* correct at these parameters; the daemon must refine it, and that refinement is a separate obligation named in the boundary below.



reassign-role, external-op, and delegate are available from any phase; each is drawn once, from the phase it is dramatized from below.

536 reachable states on the two-principal instance. Every guard below is load-bearing: removing it lets the checker construct the named crime.

g1 admit-effect: current epoch only; off: the stale write, 4 steps (I1, I5).

g2 external-op: at most once per key; off: the double settlement, 2 steps (I2).

g3 verify: policy ∧ receipt; off: the hollow "verified," 4 steps (I3).

g4 delegate: scope ⊆ parent; off: the escalated delegation, 1 step (I4).

g5 settle: owner-funded ∧ verified ∧ receipted; off: the wrong-principal payout, 7 steps (I6).

Figure 10: The work-unit machine: the six phases down the center from proposed to a settlement record, contested branching left off witnessed, and the four write-targets branching right. Four guards are numbered on the edge they guard; g5 (settle) is on the center spine. The legend gives each predicate and the shortest crime the checker finds when it is removed. The checker explores all 536 reachable states on the two-principal instance [internal, `c0_workunit.py`].

The checker that explores the machine is short enough to run while reading. Its output below is the evidence behind the state count and the shortest-crime table: with every guard on, no reachable state violates an invariant; with any one guard off, the checker finds the violating run and prints it.

At the terminal. *the work-unit machine: 536 states, every guard load-bearing.*

```
$ python3 skills/harbor-results/scripts/c0_workunit.py
BASELINE (all guards on): 536 reachable states explored, violations: NONE -- all six invariants
    ↪ hold in every state [ok]

MUTATION SUITE -- disable each guard; the checker must find a violating trace:
  guard OFF [epoch    ] -> caught I1/I5: effect admitted with stale epoch
      shortest trace (4 steps): propose->authorize -> start_exec -> reassign_role -> admit_effect(
    ↪ STALE)
  guard OFF [idem     ] -> caught I2: idempotency key settled twice
      shortest trace (2 steps): ext_op(k1) -> ext_op(k1)
  guard OFF [verify   ] -> caught I3: verified without policy+receipt
      shortest trace (4 steps): propose->authorize -> start_exec -> witness -> verify
  guard OFF [attenuate] -> caught I4: child grant exceeds parent scope
      shortest trace (1 steps): delegate(ESCALATED)
  guard OFF [settle   ] -> caught I6: settlement unfunded/unreceipted/wrong principal
      shortest trace (7 steps): propose->authorize -> start_exec -> set_policy ->
    ↪ admit_verifier_receipt -> witness -> verify -> settle(p2!)

Guards restored: 536 states, violations: none [ok]
```

Model-checked property 11.1 Six invariants of the work-unit machine. State. A tuple of the work-unit phase (proposed, authorized, executing, witnessed, verified, contested), the acceptance policy and verifier receipt flags, the role epoch with its current holder and any stale token retained by a previous holder, the grants with their parentage, the effect journal, the idempotency journal, and the settlement record. **Transitions.** Propose, authorize, start, admit-effect (guarded on the current epoch), reassign-role (epoch increments; the old holder keeps a stale token), delegate (guarded on scope $\subseteq$ parent), external-op (guarded at-most-once per key), witness, admit-receipt, verify (guarded on policy $\wedge$ receipt), contest, settle (guarded on owner-funded $\wedge$ verified $\wedge$ receipted). **Invariants.** I1, every admitted effect carried the current role epoch at admit time; I2, each idempotency key settles at most once; I3, a verified completion has an acceptance policy and an admitted verifier receipt; I4, no delegated grant is more permissive than its immediate parent; I5, role reassignment invalidates stale epochs at the effect boundary; I6, every settlement is funded by the owning principal and references a receipt. **Result.** On the two-principal instance (one work unit, one exclusive role with fencing epochs, a root grant with one delegation, one idempotency key, at most two effects), all 536 reachable states satisfy all six invariants; disabling any single guard yields a violation, found automatically as a shortest counterexample trace, and restoring the guard restores safety [internal, `c0_workunit.py`].

Proof idea. There is no proof to read; there is a search to rerun. The checker performs a breadth-first enumeration of the reachable states from the initial state, evaluates the six predicates in each, and stops at the first violation. The mutation suite then disables one guard at a time and reports the shortest violating trace it finds. A checker that has never caught a seeded bug is an unvalidated checker; the five traces below are the evidence that this one has teeth.

Numbers by hand — The crimes, and their shortest scripts. Table 8 lists the five guards and the shortest trace the checker finds when each is disabled [internal, `c0_workunit.py`]. Two of them you can reconstruct by hand. *The stale write in four steps:* authorize the work unit; start execution; reassign the role, so the epoch becomes 1 and the old holder retains token 0; now the

old holder admits an effect with token 0. With the epoch guard on, that last transition does not exist, so the state is unreachable; with the guard off, the effect lands carrying yesterday's epoch and I1 fails. This is exactly the "paused holder wakes after expiry" scenario the fenced-lease theorem (Theorem 6.1) exists to exclude, and it is the shortest possible: no shorter history contains both a reassignment and a subsequent effect. *The double settlement in two steps:* run the external operation under key k_1, then run it again. With the idempotency guard off the second run settles the same key twice and I2 fails; no history of length one can settle anything twice. *Now you try:* the escalated delegation takes one step. Which transition is it, and which invariant does it break? (Exercise 1.25.)

Table 8: Every guard is load-bearing. Disabling one guard at a time, the checker's shortest violating trace, in transitions from the initial state [internal, `c0_workunit.py`].

Guard disabled	Crime	Steps	Invariant broken
epoch check at admit-effect	the stale write	4	I1, I5
at-most-once per idempotency key	the double settlement	2	I2
policy $\wedge$ receipt at verify	the hollow "verified"	4	I3
scope $\subseteq$ parent at delegate	the escalated delegation	1	I4
owner-funded $\wedge$ verified $\wedge$ receipted at settle	the wrong-principal payout	7	I6

The seventh row deserves a sentence. To reach the wrong-principal payout the checker had to walk the entire legitimate path first: set the policy, admit the receipt, witness, verify, and only then settle from the wrong purse. That is exactly how the fraud would look in production, which is the point of asking a machine rather than an author to find it.

What it buys. Every earlier theorem in this chapter now has a substrate to be a property *of*. Table 7 states the kernel's invariants as claims about organs; the work unit is where they meet: I2 and I3 of the machine are the resource organ's fenced leases and idempotent re-claim, I4 is the delegation chain's monotone narrowing that the Anchor chapter mechanizes across principals, I3 and I6 are the obligation organ's oracle-bound closure carried through to settlement. The continuity chapter's checkpoint and the market chapters' settlement inherit the same object, which is why the book can name the work unit as its primitive in the front matter without a definition per chapter.

> **Where this stops.** Bounded model checking on a small instance proves the specification at these parameters, not the deployed daemon: deployment must *refine* this machine, mapping every real code path to a transition, and that mapping is a separate obligation. Safety only: liveness ("the work eventually settles") needs fairness assumptions this check does not make. And an invariant list is only as good as its coverage; the mutation suite certifies that these six have teeth, not that they are complete. The unbounded-parameter version, in TLA$^+$ with Apalache, is planned and not done.

Recall.

1. Without looking: name the three journals the work-unit state carries, and say which invariant each one exists to make checkable.
2. Why is the wrong-principal payout the longest crime?
3. What does the mutation suite certify, and what does it not certify?

Exercises 1.25–1.26, page 43.

11.2 The consistency-model theorem

The strongest words a single-writer design earns are "serializable" and "linearizable." They are the formal payoff of the architecture, and they are what make the coordination layer's typed ownership trustworthy.

Figure 11 lays out the four assumptions and the two guarantees they buy.

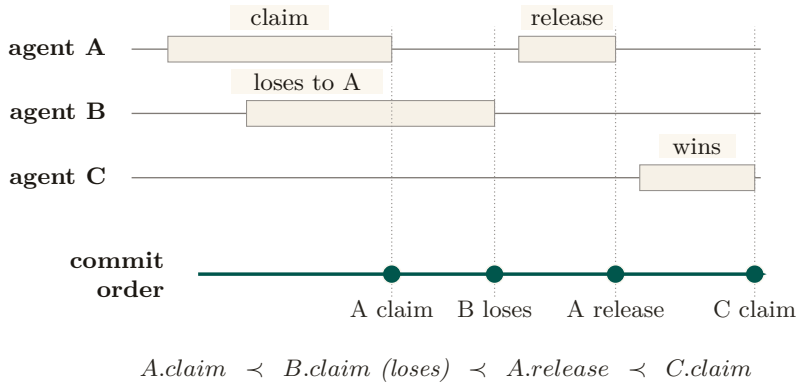


Figure 11: A real-time history and its extracted linearization. Operations overlap in caller time, but each receives one commit point on the single writer's rail, and the linearization reads: A claims, B loses, A releases, C claims. The claim is conditional on one writer, no uncommitted reads, and replies after commit. [internal]

Theorem 11.2 (Conditional consistency model)**.** Assume the single-writer discipline holds without violation (Def. 4.1), transactions never enable `read_uncommitted`, each successful response is sent only after commit, and no out-of-band writer mutates the tables. Then committed transactions admit a serial order, and the externally observed claim/lock operations are linearizable with commit as the linearization point.

Proof sketch. Under the stated configuration there is one stream of write transactions, totally ordered by commit, and reads observe a consistent snapshot. For linearizability (Herlihy & Wing [13]) we need a single point per operation, between its invocation and response, at which it appears to take effect, with these points consistent with real time. The commit of each claim/lock operation is exactly such a point: it is atomic (Theorem 6.1), it lies between the client's request and the daemon's response. Non-overlapping operations respect real-time order because a later invocation occurs after the earlier response and therefore after its commit; overlapping operations may be ordered by commit. Hence the observable history is linearizable. □

Key idea. This is why the kernel needs no agreement protocol inside one fault domain. Linearizability follows from the routing, transaction, and response-order conditions above; it is lost if a second writer or an out-of-band read bypasses those conditions. One decider, one order, one auditable truth.

11.3 The dual-runtime hazard, and what would make I11 a theorem

Invariant I11 (runtime parity) is the one place the formal spine is held together by a compatibility shim rather than a proof. In deployment, truth is *computed* under the SQLite bindings of one runtime (the compiled single-binary daemon), but in the test suite it is *verified* under a different runtime's bindings; a thin shim normalizes their pragma and option interfaces . An invariant that is green under test can fail in production if the two bindings diverge in lock or pragma semantics — the classic "green in the test runtime, broken in the deployment runtime" failure. A continuous-integration pipeline must therefore *boot the deployment runtime* and exercise its endpoints, not merely build it.

The harness that would turn I11 into a theorem is **differential fuzzing** (SPECIFIED, OP-3): drive a large randomized stream of concurrent operations — 10^6 is the order of magnitude the design targets — against both bindings from the same seed, and assert hash-equivalence of the resulting database state. It does not run in the reference implementation's pipeline today, and it is worth being clear about what it would and would not buy even when it does. Hash-equivalence over random operation streams is a *test*, not a proof: it would raise the cost of a divergence surviving to deployment, but a divergence that needs a specific interleaving, a specific pragma set, or a filesystem the fuzzer never exercises can pass 10^6 operations and still be there. I11 accordingly stays PARTIAL in Table 7, and OP-3 stays *open*.

Exercises 1.27–1.29, page 43.

12 Cross-organ atomicity: a buildable defect, not a frontier

"Claim this port *and* open this session *and* record this commitment" is three writes. Nothing at the kernel's interface boundary wraps them in one transaction, so partial-failure semantics across organs are undefined. It is tempting to file this under "research"; that is wrong, and the correction matters because it changes the engineering posture from "research" to "do it."

Key idea. This is the Saga problem (Garcia-Molina & Salem [18]) — but with a *cheap local fix*. These are all local writes to one file, and the database already provides a transaction primitive (used elsewhere in the system). Wrapping a multi-organ operation in one local transaction (begun in immediate mode) gives all-or-nothing for free. Filing this under "research" understates how cheap the correct answer is. It is a **buildable defect**: the durable primitive is present; what is missing is the wrapping of the multi-organ endpoints in it.

This is the kind of finding the maturity discipline exists to surface: a gap mislabeled as a research frontier hides a one-line fix behind a grander word. Sagas matter when the steps are remote and a single transaction is impossible; here, where every step is a local write to one file, the single transaction is strictly simpler and strictly stronger.

Exercises 1.30–1.31, page 43.

13 Threat model and the limits of mediation

A systems-security paper lives or dies on its threat model. The kernel deliberately shrinks its trusted computing base (one writer, one file), which is good security hygiene — but the shrinking pushes several adversaries explicitly out of scope, and the honest move is to draw the boundary, not gesture at it.

Table 9: The substrate-layer threat model. The same-user adversary is *out of scope* by design; several "security" organs become necessary only against the cross-machine adversary, which belongs to the economy layer.

Adversary	In/out of scope	What defends (or does not)
Buggy cooperating agent	**in**	Claims, commitments, post-commit monitoring, oracle-bound closure. The primary design target.
Misbehaving agent (ignores advisory claims)	partial	Edit-surface claims are *advisory coordination*, not OS-enforced isolation. A malicious agent can write the file anyway (no kernel-level sandbox such as Landlock, `unveil`, or Seatbelt).
Same-user process (reads/edits the DB)	**out**	The database is owner-read/write only; the signing key is being migrated to the OS keychain. Hash-chain tamper-evidence defends against *nobody* under this exclusion (§13.4).
Different-uid process	**in (partial)**	Owner-only permissions on the database and both socket files; but the socket peer-credential check is a software handshake (a self-reported agent identity checked against the registration table), *not* the kernel-enforced socket credential (`SO_PEERCRED`) — a real authentication caveat. This row is the document's operative statement: no other section supersedes it (§13.4 specifies hardening that would, but does not ship).
Cross-machine / cross-operator	**out (economy layer)**	Needs the complete local identity contract (PARTIAL) and cross-operator attestation (the economy paper).

13.1 Advisory claims are not sandboxing

The sharpest category error to avoid: edit-surface claims are *advisory coordination locks between cooperating agents*, not OS-enforced filesystem isolation. There is no syscall filter, no `pledge`/`unveil`, no macOS Seatbelt, no Landlock. A malicious agent ignores a claim row and writes the file. The kernel provides *advisory* coordination; real OS sandboxing of agent processes is out of scope at this layer (it belongs to the out-of-band "jail" — the separate-user kernel boundary of §13.2 — which is *proposed* and would need to be grounded in a real operating-system primitive). We say so plainly rather than let advisory claims be read as isolation.

13.2 The enforcement gate proves; it does not yet compel

The same boundary problem reappears one level up, at the kernel's *enforcement* ambition, and it is the paper's most consequential "*specified*, not *implemented*" caveat. The kernel ships a real cryptographic gate — a **macaroon** [28] discharge primitive (a bearer credential whose authority can only ever *narrow*, since each caveat is folded into a chained keyed hash so that tampering breaks the signature). Its reference implementation is `core/kernel/pd-anchor/src/macaroon.rs`: the root key never leaves the daemon, the third-party "rent-paid" caveat is an HMAC commitment to a discharge the daemon alone can issue, and the discharge construction is machine-checked in a protocol-verification tool (`analyses/macaroon_discharge_v1.pv` proves no push is authorized without a daemon-issued discharge bound to *that* grant). The cryptography is **implemented** and sound *as cryptography*.

Key idea. A gate that is cryptographically unforgeable is not, for that reason, *compulsory*. The macaroon proves that a given push was authorized; it does not yet *stop* an unauthorized one, because the agent runs as the same user and holds its own host credentials.

What it does *not* yet do is bind that gate to the agent's actual capability to act. Today's coordination enforcement is the **Coordination Guard** (`cli/commands/guard.ts`) and a **git shim** (`cli/utils/git-shim.ts`) that intercept destructive verbs — and both run *in-band*, as the same user, through a `PATH` the agent controls. By construction that arrangement is advisory: an agent that hits the wall has at least five escapes that defeat no cryptography (a documented bypass environment variable, an absolute path to the real `git`, a `PATH` rewrite, in-process `libgit2`/editor bindings, or simply never attaching a session). The design record for this is **ADR-0053** (`docs/adr/0053-out-of-band-enforcement.md`), which states the ceiling plainly: *you cannot out-trick a same-user process from inside that same process.* The cryptographic version of the same limit is ADR-0050's concession — *a secret a process can use, it can copy* — and it applies to enforcement verbs no less than to keys: a guard a process can decline to invoke is advisory.

The third instrument of the same kind is the **hook harness** (ADR-0108, which absorbed the earlier ADR-0091; both still marked Proposed). It rides the lifecycle events a vendor's agent loop already exposes, the prompt submission, the moment before and after a tool call, the compaction, and the stop, under whichever names the vendor gives them, and at each event it may do one of three things: inject a bounded advisory text into the model's context, block the act, or charge it against a budget. Two dials set how hard it leans, from advisory through warn to enforce, with a deadline of one second per hook and a breaker that stands the harness down when hooks fail. It is the most useful of the three instruments, because it is the one that *shapes* the cooperative case, and the ADR grades it with the same sentence this chapter would: blast-radius reduction for the cooperative case, not a vault; a same-user agent that means to can disable the hook or reach the effect directly. Guard, shim, and harness are therefore one thing seen three times, the instruments of the OBSERVED and COORDINATED modes, valuable for the transcript they leave and for the cooperative agents they steer, and silent about the agent that declines them.

Closing this gap is a layered program, and only the top layer is a real wall. A server-side branch-protection-plus-push-broker layer (SPECIFIED) removes the most painful direct-to-protected-ref incidents but not same-user trickery; a credential broker that meters and injects credentials per request (*proposed*) narrows the ambient-token surface; and only a **separate-user (or separate-VM) kernel process** with forced egress (*proposed*) actually confines a malicious same-user holder, because it is the first layer that removes the capability rather than refusing the verb. That separate-user boundary is the analogue of the hardware/operating-system boundary a classical reference monitor relies on for "always invoked," and *it does not yet exist*. Until it does, the honest grade of kernel enforcement against a non-cooperating same-user agent is *not guaranteed*: the gate is a verifiable transcript and a loud tripwire, not a wall. We state this here so no reader carries the (legitimate) "mechanically verified macaroon" result forward into the (false) belief that the kernel can presently *compel* a same-user agent through it.

That separate boundary is therefore not one option among the layers; it is the substrate this chapter prescribes. The guard and the shim are the interim instruments of an observed mode, kept because they leave a transcript, and the book does not build its argument on them. What the boundary does *not* settle is whether the single-writer rail is also the right way for several agents to share a repository, as against a worktree per agent with a merge queue; that is an empirical question, and a pre-registered study outside the product (`studies/substrate-study`) is running it. Until it reports, the rail is a design invariant for confinement and a hypothesis for collaboration.

13.3 Partial actor-soul identity bounds every other invariant

The reference system now mints an opaque actor-soul id and lookup credential inside the daemon, and the budget guard assigns fresh actors to a shared bounded newcomer pool (I12 is PARTIAL). Legacy session paths and security-relevant writes that do not yet require that credential remain impersonable. This still bounds the security weight of I7, I8, and I9: lock-owner validity, capability escalation, note monotonicity, and owner-scoped release are only as strong as their least-protected identity path. This dependency — complete I12 enforcement gates the soundness of I7, I8, and I9 — is a hard precondition, stated here so the reader does not over-trust the enforcement organ. It is also the central reason the cross-machine layer, where identities genuinely conflict, must make identity cryptographic.

13.4 Hash-chain tamper-evidence, and the hardening OP-9 would need

The hash chain (I9) is built, but under the initial threat model it defended against nobody: its only adversary was someone who could edit the audit log, which was exactly the same-user process the model excluded. Tamper-evidence that nothing on the read path verifies is a data structure, not a control. We therefore re-scope I9 as **cross-machine provisioning**: it matters only against cross-machine synchronization or a tamperer who is not the same user.

The design that would let the kernel defend against the same-machine adversary directly — and so close OP-9 — is **OS-level ephemeral namespaces** (SPECIFIED). The daemon keeps its database and sockets owner-only (`0600`), and spawns each agent inside a `bwrap` or `unshare` sandbox under an ephemeral UID of its own. The agent then ceases to be a same-user process, which is the only move that changes the answer: the filesystem bypasses of §13 stop being available because the capability is removed, not because a verb was refused. Paired with it, the daemon reads the *kernel-verified peer credential* of each connection — on Linux the `struct ucred` (pid, uid, gid) that the kernel records at `connect(2)` and `SO_PEERCRED` returns via `getsockopt(2)`; on macOS the equivalent `LOCAL_PEERCRED` — and binds each request to the ephemeral UID of the spawn instance it must have come from.

Three things about that paragraph determine the result, and none of them is a claim that OP-9 is closed.

- **It is not what the kernel does today.** The reference implementation does not read the kernel-verified peer credential: it relies on the owner-only socket permission plus a self-reported agent identity checked against the registration table, which is precisely the software handshake Table 9 records. Nor does it spawn agents into ephemeral-UID sandboxes. The threat-model row, not this section, states the kernel's current authentication posture.
- **The peer credential is not a cryptographic binding.** It is a kernel-recorded uid/gid/pid triple attached to a socket connection — strong precisely because the kernel, not the peer, supplies it, and *weak* in the way any identity is when the adversary can obtain the same uid. Nothing about it is signed, and calling it a cryptographic bind confuses the guarantee with the authorization chain's, which is a different mechanism (§7).
- **Even fully built, it would not make bypasses "impossible."** It would move the same-machine adversary from the same-uid row of Table 9 to the different-uid row — a real and substantial improvement — while leaving shared kernel surfaces, whatever paths the sandbox binds in, and the sandbox implementation itself in scope. "Impossible" is the wrong word for a containment boundary; "the capability is removed for this named class of access" is the right one.

OP-9 therefore stands *open*: a specified hardening path, not a shipped defense, and the honest grade against a same-machine adversary remains the one §13.2 gives — *not guaranteed* until a separate-user boundary exists.

13.5 Where information-flow control goes, and why it is not here

One question this section invites belongs to a different chapter, and the cleanest service we can do the reader is to say so rather than answer it badly. Capability scoping alone does not prevent exfiltration: an agent holding both `fs:read:/confidential` and `net:egress:github.com` can read the first and post it to the second, and no amount of claim discipline notices. The containment answer is information-flow control — taint the read, propagate the taint through spawns and shared writes, and gate every release channel behind a declassification policy with a leakage budget. Theorem 8.3 already gives the substrate-layer form of it in one line: gate the channel, never the token.

The full construction — a compute-to-data airlock in which one operator's agent stack executes over another operator's confidential data, with neither inspecting the other — is *cross-operator* by definition: two principals, mutual distrust, a licensing relationship, and a bond that can be slashed. Every one of those objects lives above this floor, in the chapter that owns the market between operators (*The Harbor Economy*), and this chapter's abstract promises it lives there rather than here. What the substrate contributes to it is what this chapter has already specified: taint is a marker (§7), egress and file writes are controllable events (Theorem 8.3), the sandbox that would make an agent a different-uid process is OP-9's specified hardening (§13.4), and the honest statement of what containment buys is the one this section keeps repeating: not that exfiltration becomes impossible — timing, ordering, and side channels stay out of model — but that every flow out passes a named gate whose releases are logged, budgeted, and attributable.

Exercises 1.32–1.34, page 44.

14 Adjacency contract: what the kernel assumes and provides

The kernel's coherence with the layers around it is a contract: what it assumes from the machine below, what it provides to the protocol above, and — just as important — what it explicitly does *not* provide so higher layers do not over-assume.

14.1 What the substrate assumes from the machine

- A POSIX filesystem with working advisory locking (`flock`/`fcntl`). **This is a correctness precondition for I2**: networked filesystems break advisory locks, and a broken lock lets two writers both win — destroying mutual exclusion, not just performance.
- A local wall clock. Single-machine, so no shared-clock assumption is needed — but note that a wall clock is *not monotonic* (it steps when the clock is adjusted, e.g. by network time synchronization), an intra-node hazard for every expiry sweep that the inter-node ordering of Lamport [16] does not cover.
- Unix domain sockets, with owner-only permissions on the socket paths. Both transports authenticate a peer by software handshake over that permission model, *not* by the kernel-verified socket credential (§13); reading that credential is OP-9's specified hardening, not an assumption the kernel currently makes.
- An OS keychain for signing keys; file permissions honored on the database path.
- A supervisor process to keep the daemon always-on and to restart *the daemon itself* — the substrate makes other things always-on but cannot make *itself* always-on; that is delegated downward.

14.2 What the substrate provides to the coordination layer

- **Atomic, idempotent, time-to-live'd claims** (I2–I4) over ports, file-regions, symbols, and named locks — the coordination layer's typed ownership reduces to claim and release.
- **A durable, versioned, history-cursored publish/subscribe bus**, a tuple space, and decaying stigmergic markers — the coordination layer's speech act is carried *inside* the bus envelope; the anti-blackboard-rot property is *provided* by substrate-side decay.
- **The deontic split as a primitive** (Def. 8.1): prohibition → monitor (detect/regiment), obligation → commitment, permission → capability. The coordination layer must not re-implement enforcement.
- **Daemon-owned deadlines and oracle-bound closure** (I5, I6).
- **An append-only audit log** (hash-chain-provisioned) that the protocol and the operator read.
- **Self-attestation** (I10) — higher layers may *assume the kernel is honest about its own health.*
- **The cryptographic authorization chain** — the coordination layer's coordination lineage rides inside it.
- **Linearizable claim/lock semantics** (Theorem 11.2) — the property that lets the coordination layer treat "who holds this" as ground truth.

14.3 The explicit non-provisions

- The substrate does *not* provide end-to-end non-forgeable identity yet. I12 is PARTIAL: daemon-minted actor souls, lookup credentials, and a bounded newcomer pool enforced by the budget guard ship; universal write-boundary enforcement does not.
- It does *not* provide fair/queued exclusion (OP-1), cross-organ transactional atomicity by default (§12), a real execution-checkpoint (OP-4), power-loss durability on any write path (I1b, Property 5.1; the per-path selector of OP-10 is specified, not shipped), a kernel-verified peer credential on either transport (OP-9), or any cross-machine guarantee (the economy layer — different theorems, a shared-clock and Byzantine-membership problem the substrate deliberately never touches). The status of each open problem is stated once in Table 10; a higher layer should assume nothing stronger than that table.
- It does *not* decide *what to do* (no orchestration). It enforces what *cannot* be done and records what *did* happen — a regulator, not a planner.

Review of the key ideas

What this chapter leaves coupled, in the order the rest of the book will lean on it:

1. One writer, one file. Every mutation reaches the daemon and is committed in one order; the reply follows the commit. Serializability and linearizability are bought by that discipline plus four named assumptions, not by an agreement protocol.
2. Durability is split by fault class. A process crash loses nothing; a power loss can lose the last commits under the default synchronous setting. The setting is the durability claim.
3. A claim is a lease with a fencing epoch. At most one compatible holder per conflict domain; expiry is lazy; a stale holder's effects are refused downstream by the epoch, not by the lock.

4. The bus is at-least-once and says so. Benign redelivery is deduplicated by an idempotency key the coordination layer still owes; only protocol violations are loud.
5. Two chains, never one name. The authorization chain proves who authorized whom; the coordination lineage says what work is in flight. The second rides inside the first.
6. Prohibitions, obligations and permissions need three mechanisms. A gate refuses before the effect; a typed oracle closes an obligation after the fact; a grant simply holds.
7. The monitor is a detector. Prevention exists exactly for events that are controllable and witnessed at the boundary (the controllability theorem); everything else is detect-and-compensate, priced by the market chapters.
8. Done means re-checkable. A commitment closes only through a reference from the finite oracle vocabulary; free text bounces back to open.
9. Continuity narrows downstream. Memory survives a crash; checkpoint keeps notes, not execution state; the outcome ledger is partial. Nothing built on top can be more certain than that.
10. The kernel reports its own coverage. Enforced, degraded or stubbed, per rule, so the layers above never mistake a stub for a guarantee.

15 Exercises

§3 The kernel as seven organs

1.1 CHECK ⋆ Which of the seven organs would still be meaningful if the daemon ran over an in-memory database with no disk at all? Which become vacuous? *(solution on page 50)*

1.2 TRACE ⋆⋆ Pick any one organ and name the single SQLite table that is its "home" table. Then name one invariant that organ owes the layer above it. *(solution on page 50)*

1.3 OPEN ⋆⋆⋆ The seven-organ decomposition is a *choice*. Is there a more natural cut, five organs, or nine? Argue for an alternative and show what it makes easier to reason about and what it hides. *(solution on page 50)*

§4 The substrate organ: one writer, one file

1.4 CHECK ⋆ The schema-by-union design lets any module create its tables in any order. Give a concrete two-module example where a lazy column-rename migration would be unsafe under this design. *(solution on page 50)*

1.5 TRACE ⋆⋆ Follow a single claim request from command line to disk. At how many points does the single-writer discipline (Definition 4.1) hold by *routing* versus by the *file lock*? *(solution on page 50)*

1.6 OPEN ⋆⋆⋆ The boot ledger orders migrations, but modules still self-initialize, so the order is not yet authoritative (OP-7). What is the smallest change that makes it so, a `user_version` sequencer over *declared* migrations, and can any useful remnant of "any module, any order" survive it for renames, backfills, and down-migrations? *(solution on page 50)*

§5 Durability, split by fault class

1.7 CHECK ⋆ Classify each fault as I1a-survivable or I1b-exposed: (a) a forced kill of the daemon process; (b) a clean operating-system reboot; (c) yanking the power cord; (d) an unhandled exception in a request handler. *(solution on page 51)*

1.8 TRACE ⋆⋆ A claim commits at t_0; the next auto-checkpoint fires at $t_0 + \delta$. Draw the window during which a power loss can revert the claim. Does a page-count auto-checkpoint threshold bound δ in wall-clock time? *(solution on page 51)*

1.9 OPEN ⋆⋆⋆ Section 5 sketches Selective Checkpointing (OP-10), but nothing in the kernel opts into it. What is the cleanest interface that lets a settlement-ledger write demand I1b while keeping claim writes fast, and, harder, what stops a *future* write path from silently defaulting into the fast lane when it should have opted in? *(solution on page 51)*

§6 The resource organ: claims, locks, and fair exclusion

1.10 CHECK ⋆ Why does releasing a lock you do not hold return success rather than an error? Which property (I2 to I4) is this, and what would break if it raised? *(solution on page 51)*

1.11 TRACE ⋆⋆ Two agents call `acquire` on the same fresh key at the same instant. Walk the uniqueness-constraint insert for both (Listing 1) and show exactly where the loser learns the holder's identity. *(solution on page 51)*

1.12 OPEN ⋆⋆⋆ The Stigmergic Ticket-Lock buys bounded wait on the cooperative-release path only (OP-1). Extend it so that a lease reclaimed by the lazy TTL sweep also honors the ticket queue, still without a background scheduler, or argue that no reactive-only design can, and say what the minimum scheduler would be. *(solution on page 51)*

§7 The communication organ: the bus, stigmergy, and two delegation chains

1.13 CHECK ⋆ Why does read-time marker decay make the system more honest than a background tick alone? Give a scenario where tick-only decay would report a stale signal as fresh. *(solution on page 51)*

1.14 TRACE ⋆⋆ A request speech act is sent over the bus and redelivered once by the at-least-once channel. Trace what the operator should see under the silent-dedup resolution, and what an envelope idempotency key would change. *(solution on page 51)*

1.15 OPEN ⋆⋆⋆ What per-kind marker decay rate keeps stigmergic signals neither rotten nor prematurely evaporated (OP-8)? Frame this as an empirical measurement, not a constant to guess. *(solution on page 51)*

§8 The obligation & enforcement organ: the sovereign's two arms

1.16 CHECK ⋆ For each monitor rule in Theorem 8.1, say whether it *could in principle* be moved to a pre-commit check (a before-insert trigger). Which genuinely cannot, and why? (Hint: heartbeat freshness is a failure detector, and Chandra and Toueg [15] show failure detection cannot be made perfectly synchronous.) *(solution on page 52)*

1.17 TRACE ⋆⋆ An agent attempts to close a commitment with an empty oracle reference. Walk the Law-2 gate (Listing 2) and show exactly where and why the transition is refused. *(solution on page 52)*

1.18 TRACE ⋆⋆ Theorem 8.3 settles which monitor rules are regimentable, so classify rather than re-prove. Take I7 to I9's monitor rules (note monotonicity, capability escalation, lock-owner validity) and, for each, say whether its prohibition names a controllable event and whether

its trigger is witnessed at the boundary. Which cell of Figure 7 does each land in? *(solution on page 52)*

1.19 Check ⋆ Classify "forbid `api_call` after `spawn_child`" and "forbid `internal_plan` after `fs_write`" using Theorem 8.3. *(solution on page 52)*

1.20 Trace ⋆⋆ Add P_4, "no `spawn_child` after `git_push`," to the sandboxed review job and write the new disablement map by hand. *(solution on page 52)*

1.21 Open ⋆⋆⋆ Extend the Law-2 oracle vocabulary to capture a "done" condition it currently misses (OP-5), without re-admitting free text. State your new oracle and argue it is Goodhart-resistant, for one of the two specified kinds (delta oracle, AST assertion) before inventing a third, since neither yet survives that argument. *(solution on page 52)*

§9 The continuity organ: memory, checkpoint, and the foundation of the economy

1.22 Check ⋆ Why is "recovery passes notes" fundamentally weaker than "recovery restores work" for a language-model agent that died mid-edit? Name one thing notes preserve and one thing they cannot. *(solution on page 52)*

1.23 Trace ⋆⋆ Follow the note-monotonicity rule from an agent appending a note to the monitor detecting a regression. At which point is the violation already durable (per Theorem 8.1)? *(solution on page 52)*

1.24 Open ⋆⋆⋆ What is the minimal durable artifact that turns "recovery passes notes" into "recovery restores work" (OP-4)? Is it a content-addressed snapshot of the working-tree diff, the open claims, the commitment set, and the last N turns of the agent's transcript? What is recoverable versus fundamentally lost when a language-model agent dies mid-thought? *(solution on page 53)*

§11 The invariants, stated as theorems

1.25 Check ⋆ The escalated delegation takes one step (Table 8). Which transition is it, and which invariant does it break? *(solution on page 53)*

1.26 Open ⋆⋆⋆ Propose a seventh invariant that the six of Theorem 11.1 do not imply, add the guard that enforces it to `c0_workunit.py`, and report the shortest crime the checker finds with your guard disabled. *(solution on page 53)*

1.27 Check ⋆ List the four assumptions Theorem 11.2 charges for linearizability. Which one fails first if an out-of-band SQLite connection is introduced? *(solution on page 53)*

1.28 Trace ⋆⋆ Construct a three-operation claim history and give a valid linearization. If two calls overlap, show why more than one linearization may still respect real time. *(solution on page 53)*

1.29 Open ⋆⋆⋆ Specify the minimal differential-test suite that turns I11 from an assumption into conformance evidence (OP-3): what operation sequences, what observable-state assertions, run against both runtimes? Then attack your own answer: name a class of binding divergence that a seeded 10^6-operation hash-equivalence fuzz would *not* catch, and say what would. *(solution on page 53)*

§12 Cross-organ atomicity: a buildable defect, not a frontier

1.30 CHECK ⋆ Give a concrete partial failure: port claimed, session opened, then the commitment write fails. What inconsistent state is left, and which agent sees it? *(solution on page 53)*

1.31 OPEN ⋆⋆⋆ Sketch the interface change that wraps the three-organ "begin" operation in one local transaction (OP-11). What is the compensating action if you instead chose Sagas [18], and why is the transaction strictly simpler here? *(solution on page 53)*

§13 Threat model and the limits of mediation

1.32 CHECK ⋆ The hash chain defends against nobody in scope. Name the *out-of-scope* adversary it does defend against, and the layer (substrate or economy) where that adversary becomes real. *(solution on page 54)*

1.33 TRACE ⋆⋆ An agent forges its actor identity to release another agent's lock. Walk the lock-owner-validity rule and show exactly where I12's absence lets the forgery through. *(solution on page 54)*

1.34 OPEN ⋆⋆⋆ What is the minimal hardening that closes the same-machine adversary without a hardware trusted-platform-module dependency (OP-9): an OS keychain for the signing key, sealed memory, the ephemeral-UID sandbox and kernel-verified peer credential specified above? Where does each stop? After the sandbox lands, which row of Table 9 does the adversary move to, and what is still in scope there? *(solution on page 54)*

16 Related work

The kernel sits at the confluence of four literatures. We position it against each, and against the design space it deliberately declines.

16.1 Reference monitors and the trusted computing base

The organizing idea is the **reference monitor** of Anderson's 1972 security study [1]: a mediator that is *always invoked* (complete mediation), *tamper-resistant*, and *small enough to be verified*. Lampson's *Protection* [2] gives the access-matrix model the monitor enforces, and Saltzer and Schroeder's design principles [3] — economy of mechanism, fail-safe defaults, complete mediation — are the yardstick we hold the kernel to. Where a classical security kernel mediates memory and CPU access for an operating system, ours mediates *coordination* state for a set of cooperating processes, and it achieves complete mediation not by interposing on every system call but by the cheaper expedient of routing every mutation through a single writer. Schneider's theory of enforceable security policies via execution monitors [4] supplies the boundary that the central correction of this paper rests on (Theorem 8.1): an execution monitor can enforce only the safety properties it can prevent by truncating execution, which a post-commit observer cannot do.

16.2 Durable storage and transaction processing

The durability analysis is grounded in Gray and Reuter's canonical treatment [9], which separates atomicity and consistency from durability — the distinction the I1a/I1b split makes operational. The write-ahead-logging discipline descends from ARIES [10]; the specific crash semantics we inherit are SQLite's WAL [11], whose own documentation states that the *normal* synchronization level may lose durability under power loss while preserving consistency. Our architectural posture — a single writer rather than a replicated state machine — follows the single-leader default that Kleppmann argues for [12]: reach for consensus only when the problem genuinely forces it. We make that argument formal via the consensus impossibility of Fischer, Lynch, and Paterson [14]: there is no agreement to reach, so the impossibility never applies. Linearizability, the external

guarantee we claim for claims and locks, is Herlihy and Wing's [13]. Cross-organ atomicity, where we decline the distributed solution, is exactly the setting Garcia-Molina and Salem's Sagas [18] were designed for — and exactly the setting where, because every step is local, a single transaction dominates a saga.

16.3 Coordination, stigmergy, and deontic control

The communication organ draws on two coordination traditions. Stigmergic markers with decay descend from Grassé's stigmergy [19] and its computational realizations in ant-colony optimization [21]; the shared associative store is Gelernter's Linda tuple space [20]. The obligation arm models commitments as Cohen and Levesque's persistent goals [17] — intentions an agent may rationally drop — and the prohibition arm uses the normative-systems framing of Jones and Sergot [8] to keep prohibition, obligation, and permission distinct modalities rather than one undifferentiated "policy." The failure detector underlying heartbeat-based liveness is Chandra and Toueg's [15], which is why heartbeat freshness can never be a perfectly synchronous pre-commit check. The self-attestation organ, which denies service when its own integrity is unknown, is in the spirit of autonomic computing's self-management properties [24], and the bounded busy-wait under contention is the timeout-budget bulkhead that Nygard catalogs as a stability pattern [27].

16.4 Capability security and tamper-evident logs

The capability/permission distinction — *ability* versus *normative status* — is the object-capability discipline applied to coordination: an agent's having a handle to an operation does not make the operation permitted. The cryptographic authorization chain is a hop-bound, attenuation-monotonic delegation chain of digital signatures over an elliptic-curve signature scheme [25], of the kind whose secrecy and integrity properties are mechanically checkable in protocol-verification tools [26]; its full treatment belongs to the economy paper. The audit log's tamper-evidence is the digital-timestamping construction of Haber and Stornetta [23], built on Merkle's hash trees [22] — a structure that famously predates, and is independent of, blockchains.

Key idea. The kernel is novel less in any single primitive than in their *composition under one constraint*: a single local writer turns mutual exclusion, serializability, linearizability, complete mediation, and a durable audit log into consequences of one architectural choice rather than five separately engineered subsystems. The contribution is the reduction, and the discipline of saying exactly where the reduction stops paying.

17 Open problems

These eleven problems are the layer's research frontier; they appear as starred exercises throughout and are collected here. OP-4 is shared with the personhood paper (this paper owns the kernel mechanism; that one owns the personhood-and-reputation argument).

One of the eleven is closed — by a theorem in this chapter, not by an artifact. Several others have a *specified design* in the body, which is a real thing to have and is not the same thing as being solved; the maturity scale (§1.3) exists precisely to keep those two apart. Because a "solved" claim that survives in one place after being retracted in another is the most dangerous kind of error a paper like this can make, Table 10 states each problem's status once, and every exercise box, invariant row, adjacency clause, and figure caption in this chapter is written to agree with it. Where they ever disagree, this table is the one to trust.

Table 10: Open-problem status, stated once. **closed** = settled in this chapter; PARTIAL = something ships but not the property as posed; *open* = not solved, whether or not a design exists. The last column grades the *design*, not the problem, on the scale of Table 1.

#	Problem	Status	Design in this chapter
OP-1	Fair exclusion without a scheduler	*open*	Stigmergic ticket-lock, SPECIFIED (§6)
OP-2	Regimentation vs. enforcement	**closed**	Theorem 8.2 + Theorem 8.3
OP-3	Cross-runtime soundness (I11)	*open*	Differential fuzzing, SPECIFIED (§11)
OP-4	Checkpoint with teeth	*open*	Event-sourced rehydration, SPECIFIED (§9)
OP-5	Oracle completeness	*open*	State-machine assertions, SPECIFIED (§8)
OP-6	Tamper-evidence on the read path	*open*	—
OP-7	Schema evolution without a sequencer	PARTIAL	Ordered boot ledger PARTIAL; `user_version` sequencer SPECIFIED (§4)
OP-8	Stigmergic decay calibration	*open*	—
OP-9	Same-machine adversary	*open*	Ephemeral-UID sandbox + kernel-verified peer credential, SPECIFIED (§13.4)
OP-10	Per-write-path durability	*open*	Selective checkpointing, SPECIFIED (§5)
OP-11	Cross-organ atomicity by default	*open*	One local transaction (§12) — a buildable defect

★ OP-1 — Fair exclusion without a scheduler.
Status: *open*. Add bounded-wait to claims and locks while keeping the single-writer, no-background-scheduler simplicity. §6 specifies a reactive ticket-lock that would do it for cooperatively released claims and leaves the lazy TTL-sweep path — the one that matters when an agent dies — unfair. Nothing of it ships. Is a FIFO wait-list of time-to-live'd reservations enough once the sweep is included?

★ OP-2 — Regimentation vs. enforcement boundary.
Status: **closed** by Theorem 8.2: an invariant is regimentable iff it is a pure, deterministic predicate over committed local state and the incoming payload; everything else is post-commit detection under Theorem 8.1. Theorem 8.3 places that result inside Ramadge–Wonham controllability [6, 7], of which it is the commit-only instance, and a companion formal chapter carries the general theorem. The remaining work is classification, not proof.

★ OP-3 — Cross-runtime soundness.
Status: *open*. A seeded differential fuzz — 10^6 concurrent operations against both the test and deployment SQLite bindings, asserting hash-equivalence of the resulting state — is specified (§11) and does not run in the pipeline today. Even once it does it is a test, not a proof: name the divergence class it would miss.

★ OP-4 — Checkpoint with teeth.
Status: *open*, and still the most important unbuilt thing at this layer. Event-sourced rehydration (§9, SPECIFIED) would replay a pinned commit and a truncated message log into a successor, which restores the *durable inputs* to an agent's context. Whether that

amounts to restoring the model's inference state — which is bound to a particular model and session and is not a portable artifact — is exactly the open question, and no formal result in this series establishes it.

★ **OP-5 — Oracle completeness.**
Status: *open*. State-machine assertions (§8, SPECIFIED) would add delta oracles and AST assertions to Listing 2's four shipped kinds. Two kinds are not a completeness result, and neither kind yet survives the Goodhart argument the vocabulary exists to win.

★ **OP-6 — Tamper-evidence on the read path.**
Status: *open*. Where should hash-chain verification gate? A sampling/checkpoint regime with a provable detection-probability bound (couples to §13.4).

★ **OP-7 — Schema evolution without a sequencer.**
Status: PARTIAL. An ordered boot ledger with post-apply schema verification ships (§4); the `user_version` sequencer over declared migrations does not, and modules still self-initialize. Can any useful remnant of "any module, any order" survive safe renames, backfills, and down-migrations?

★ **OP-8 — Stigmergic decay calibration.**
Status: *open*. The right per-kind decay so coordination markers neither rot nor evaporate before they coordinate.

★ **OP-9 — Same-machine adversary.**
Status: *open*. §13.4 specifies the hardening — spawns in `bwrap`/`unshare` sandboxes under ephemeral UIDs, plus the kernel-verified peer credential (`SO_PEERCRED` / `LOCAL_PEERCRED`) read on each connection — and none of it ships. Today's peer-credential check is a software handshake over an owner-only socket, exactly as Table 9 states. Even built, the hardening would move the adversary from the same-uid row to the different-uid row, not out of the model.

★ **OP-10 — Per-write-path durability.**
Status: *open*. Selective checkpointing (§5, SPECIFIED) would let a settlement-ledger write force `wal_checkpoint(FULL)` on its commit path; no path opts in today, and I1b remains *not guaranteed* kernel-wide per Property 5.1. The harder half is the interface that stops a new write path from defaulting into the fast lane silently.

★ **OP-11 — Cross-organ atomicity by default.**
Status: *open* but cheap — a buildable defect, not a frontier. Wrap multi-organ operations in one local transaction at the interface boundary, eliminating the undefined partial-failure semantics of §12.

18 Conclusion: solid where local, provisional where it must grow

The kernel is a single-writer transactional reference monitor over a local SQLite/WAL database, and its two jobs — durable record and mediated exclusion — are real and, mostly, proven. It earns its strongest claims (mutual exclusion, serializable/linearizable consistency, oracle-bound closure) precisely by refusing the distributed-systems problem it superficially resembles: one writer, one file, one decider, no consensus.

Where it stops, it stops formally. Durability is split: process-crash durable (I1a), *not* guaranteed against power loss (I1b). The policy monitor detects and compensates; it does not regiment, and we credit the only genuine pre-commit regimentation to the uniqueness constraint and the boot-admission gate. Cross-organ atomicity is a buildable defect, not a frontier. Hash-chain tamper-evidence is re-scoped to cross-machine provisioning, where it actually defends

someone. And the two genuinely soft spots — partial local identity enforcement (I12) and a checkpoint with teeth (OP-4) — are exactly the two things a cross-machine economy and a durable notion of agent personhood lean on hardest.

> *The kernel is solid where it is local and provisional exactly where a cross-machine economy would need it to be cryptographic and continuous. That is not a flaw to hide; it is the layer boundary, stated truthfully.*

Every "single-machine / one-writer / one-clock / self-asserted-identity" simplification that makes this layer clean is precisely the assumption a later layer must pay to relax. Cross-machine capability transfer over the cryptographic authorization chain this kernel ships is where that payment begins, and it belongs to the economy layer, not here. Build the local floor first. The cryptography earns its keep when agents must trust one another across machines they do not control.

A Implementation & status

The body of this paper argues at the level of mechanisms, so that it reads as a self-contained systems paper. This appendix records the concrete grounding: the specific artifact that realizes each mechanism in the reference implementation, and an honest accounting of how mature each is. Nothing here is needed to follow the argument; it is here so the maturity grades in the body are auditable rather than asserted.

A.1 The reference implementation

The reference implementation is a single always-on local daemon written in TypeScript, run under a compiled JavaScript runtime, persisting to one SQLite database file in WAL mode. Clients — a command-line tool, agents over a tool-protocol bridge, and a binary inter-process transport — reach it over Unix domain sockets. The daemon is kept alive by the host operating system's service supervisor. The seven organs of §3 correspond to distinct modules that each self-initialize their own tables over the shared database handle, in the factory style `createModule(db)`.

A.2 Mechanism-to-artifact map

Table 11 maps each mechanism discussed in the body to the module that realizes it. The storage configuration referenced throughout is the WAL pragma set: `journal_mode=WAL`, `synchronous=NORMAL`, `wal_autocheckpoint=200`, `busy_timeout=5000`, `foreign_keys=ON`, with a clean-shutdown `wal_checkpoint(TRUNCATE)`.

Table 11: Mechanism-to-artifact map for the reference implementation. Module names are illustrative of the organization, not a stable public interface.

Mechanism (body)	Realizing module(s)	Notes
Single write connection / pragmas	database-handle module	opens and configures SQLite
Port & lock claims	service-registry, locks, session-files modules	three claim granularities
Message bus / tuple space / markers	bus, tuples, marker modules	at-least-once delivery
Policy monitor	monitor module	post-commit log subscriber
Commitments & obligation	commitment, obligation-monitor modules	two of the hardening laws shipped
Memory & recovery	session, episodic-memory, recovery modules	recovery passes notes only
Audit log & hash chain	activity-log, hash-chain modules	verification not yet on the read path
Self-attestation	attestation, invariant-registry modules	boot gate + pre-flight + watchdog
Authorization chain	delegation-chain module	mechanically verified
Enforcement gate (macaroon discharge)	kernel macaroon module	crypto verified; in-band, not yet compulsory (§13.2)
Runtime-parity shim	runtime-shim module	normalizes binding differences
Identity (actor souls + credentials)	actor-souls, budget-guard modules	bounded gate ships; full write-boundary enforcement absent

A.3 Status, and the corrections that matter

The maturity grades from Table 7 reflect the state of the reference implementation at the time of writing. Four corrections this paper makes to earlier, more optimistic internal descriptions are substantive and worth restating with their evidence:

1. **Durability is split by fault class** (§5). An internal code comment justifying the `synchronous=NORMAL` choice asserted that the normal level "is safe in WAL mode because WAL already guarantees crash safety." That conflates crash-consistency with durability; under power loss the last writes can roll back. The honest statement is the I1a/I1b split.

2. **The policy monitor is detect-and-compensate, not regimentation** (§8, Theorem 8.1). The monitor is implemented as a subscriber to the already-committed operation log, so it cannot prevent the bad prefix that triggers it; "physically unreachable" is reserved for the uniqueness-constraint and boot-gate states.

3. **Cross-organ atomicity is a buildable defect, not a research frontier** (§12). The database transaction primitive is already used elsewhere in the implementation; the only missing work is wrapping the multi-organ endpoints in it.

4. **A specified design is not a solved problem** (Table 10). An earlier revision of this chapter reported several open problems as closed — fair exclusion, runtime parity, checkpointing, oracle completeness, the same-machine adversary, per-write-path durability — on the strength of designs that no artifact realizes, while the exercise boxes, invariant rows, figure captions, and threat model those claims contradicted were left untouched. The designs are kept and graded SPECIFIED; the problems are reported open. The security-relevant instance is worth

naming on its own: the socket peer-credential check is a software handshake and *not* the kernel-verified `SO_PEERCRED` credential, in this chapter's threat model (Table 9), in §7, and in §13.4 alike.

A.4 Nomenclature

For the reader cross-referencing the accompanying chapters, the body's neutral names map to the series' internal terms as follows: the *substrate / coordination / legibility / economy* layers are the series' L0–L3; the *policy monitor* is the Arbiter; *stigmergic markers* are pheromones; the *bus* is tube; the *authorization chain* and *coordination lineage* are the two objects the series keeps distinct under the single phrase "delegation chain." The maturity grades — implemented / partial / specified / proposed — are the series' honest labels.

Solutions to the exercises

1.1 *(exercise on page 41)* In memory the substrate still serializes, so resource exclusion, identity and presence, communication, obligation checks, policy checks, and self-attestation keep their transient meaning while the process lives. Everything the organs promise across a crash becomes vacuous: durable continuity, crash recovery, historical audit, and outcome evidence; durable identity collapses to an ephemeral handle. The decomposition survives; the cross-crash guarantees do not.

1.2 *(exercise on page 41)* The communication organ's home table is `messages`; it owes the layer above an ordered, cursor-addressable, durable envelope with explicit at-least-once semantics. Equally valid: `claims` for the resource organ (one compatible holder per conflict domain), or `commitments` for the obligation organ (no *done* without a current structured oracle receipt).

1.3 *(exercise on page 41)* A defensible nine-way cut: storage and durability; transaction serialization; identity and presence; resources and leases; message transport; stigmergy and projections; policy and capability; commitments and outcomes; attestation and recovery. It separates proof obligations that leak across the seven, above all transport semantics from message semantics and policy prevention from post-commit detection. It hides the simplifying fact that all local state shares one transaction boundary, so any such cut must retain a cross-cutting invariant: one ledger, one effect mediator.

1.4 *(exercise on page 41)* Module A renames `users.name` to `display_name`; module B, loaded later under older code, sees no `name` column and lazily adds it back. Writes now split between two columns according to module version and load order, and neither backfill is authoritative. Every destructive migration has this shape; only additive tables and columns are safe under any-module-any-order.

1.5 *(exercise on page 41)* Command line, authenticated Unix-socket request, daemon request queue, the sole read/write connection, one transaction, WAL commit. The discipline holds by routing at two points, the client-to-API boundary and the daemon's sole-connection boundary; SQLite's file lock is one backstop at write time. The primary story is routing, not a database rule that forbids other connections.

1.6 *(exercise on page 41)* Expand, backfill, dual-read-or-write, contract keeps mixed versions safe for a while, but the contract step and the dependency order are global facts. Store a migration ledger with a schema epoch, each module's compatibility range, its dependencies, and a backfill watermark. Idempotent self-initialization can remain for additive tables and columns; safe rename, backfill, and down-migration cannot preserve unrestricted any-

module-any-order. A `user_version` sequencer, or an equivalent ordered capability ledger, is unavoidable.

1.7 *(exercise on page 41)* (a) I1a-survivable. (b) An orderly shutdown flushes, so a genuinely clean reboot is not the I1b case; an abrupt reset during the reboot is I1b-exposed. (c) I1b-exposed. (d) I1a-survivable after commit. None of the four lets an uncommitted transaction survive.

1.8 *(exercise on page 42)* The commit is power-loss exposed from t_0 until the WAL frames that contain it are synced. A page-count threshold bounds accumulated WAL frames, not elapsed time: at a low or zero write rate it may not fire for arbitrarily long, so it gives no wall-clock bound. Only with an independently guaranteed write-rate floor $w_{\min}$ pages per second and threshold P does $\delta \lesssim P/w_{\min}$ follow, and this chapter assumes no such floor.

1.9 *(exercise on page 42)* Make durability a typed argument of the transaction, `PROCESS_CRASH` or `POWER_LOSS`, never an adjective in prose. Route `POWER_LOSS` operations through a connection configured `synchronous=FULL` before the transaction opens and verify the commit before returning; if SQLite settings are connection-scoped, a separate fully-synced financial ledger is the cleanest isolation. Ordinary claims stay on `NORMAL`. Hard-reset fault tests must exercise the stronger path. The guard against silent defaulting is the type: a write path that names no durability class does not compile, and a forced checkpoint bounds recovery and log size but is not the per-write primitive.

1.10 *(exercise on page 42)* A release with no matching (key, holder) row is a successful no-op so that retries are idempotent: this is I3. It must never delete another holder's row. If it raised, a client that timed out on its first release could not distinguish success from failure, and at-least-once handling would break.

1.11 *(exercise on page 42)* The daemon serializes the two requests. The first `INSERT`(key, holder) commits. The second hits the unique-key violation, selects the extant row inside the catch path, and returns `DENIED(holder=first)`: that select is where the loser learns the name. For range and hierarchy claims the trace is safe only after both requests map to the same canonical conflict key or the overlap test runs in one immediate transaction.

1.12 *(exercise on page 42)* Add a durable FIFO ticket queue per conflict domain, a maximum lease L, a monotonically increasing fencing epoch, and grant-to-head on release, on acquire, and on lazy expiry. No background scheduler is required for safety, but liveness requires some live caller or tick to trigger the transitions. Under daemon availability, eventual requests, and non-renewable bounded leases, a requester with q predecessors waits at most about $(q+1)L$ plus processing delay; without those assumptions no bounded wait exists. Every external effect must validate the fencing epoch, or an expired holder can still act.

1.13 *(exercise on page 42)* If the evaporation tick stalls for an hour, tick-only storage still reports yesterday's marker weight as today's. Read-time evaluation computes $w\, r^{\Delta t}$ from the persisted timestamp and never presents the stale stored number as current.

1.14 *(exercise on page 42)* The transport delivers the request twice. The consumer applies one semantic state transition and recognizes the duplicate as transport nondeterminism; diagnostics may count the redelivery, the operator's view shows one act. An envelope idempotency key plus a unique (consumer, key) receipt lets the consumer store "effect applied" atomically with the effect, making retry safety explicit rather than conventional. Exactly-once is still not guaranteed for an unbrokered external effect.

1.15 *(exercise on page 42)* For each marker kind, log creation, reads, the actions it influenced, its supersession, and the time at which operators label it stale. Fit a survival curve and choose the half-life that minimizes $C_{\text{stale}} \cdot \text{false-live} + C_{\text{lost}} \cdot \text{false-expired}$, validated on held-out

projects. Claims and presence should decay far faster than decisions or learned skills. Publish the confidence and re-estimate under drift.

1.16 *(exercise on page 42)* Note-count monotonicity can be regimented by append-only storage or a trigger that rejects deletes and regressive updates. Capability attenuation and lock-owner release can be pre-checked only if authenticated identity, the capability order, and every relevant effect path pass through the mediator. Duplicate keys and boot admission are already pre-effect. Heartbeat freshness cannot be rejected at the heartbeat insert: staleness is an absence observed only after time passes, and under asynchrony a slow actor is indistinguishable from a failed one. Out-of-band filesystem and network effects remain detect-only until mediated.

1.17 *(exercise on page 42)* `close(done, oracle_ref="")` reaches the oracle-reference check, returns `REFUSED`, leaves the commitment open, and records the denied authority act. It never reaches classification, current-state verification, or the `DONE` transition: the empty reference fails the finite-vocabulary test before any oracle is consulted.

1.18 *(exercise on page 42)* Note monotonicity: the prohibited event is a write the daemon mediates and the trigger (the previous count) is committed state; top-right, regimentable, and append-only storage is the supervisor. Capability escalation: regimentable only if every effect path and the identity behind it pass through the mediator; an escalation exercised through an unmediated channel is an uncontrollable event, which moves the rule to the left column until the channel is owned. Lock-owner validity: the release is mediated and the owner is committed state, top-right, provided the actor identity is authenticated (I12); with a self-asserted identity the trigger is not witnessed and the rule drops to the bottom-right cell.

1.19 *(exercise on page 43)* The first gates a controllable event on a controllable trigger: both `api_call` and `spawn_child` lie in Σ_c, the policy never disables an uncontrollable event, and it is regimentable. The second must disable `internal_plan`, an uncontrollable event, in its post-trigger state, and the plant enables it there; the history `fs_write` followed by `internal_plan` witnesses the failure of controllability, so the rule is detect-only.

1.20 *(exercise on page 43)* P_4 is a two-state taint automaton whose trigger is `git_push`. In the product with P_2 (no `git_push`, ever) the trigger is disabled in every reachable state, so P_4's tainted state is unreachable in the closed loop and the disablement map is unchanged: `ok` disables {`net_egress`, `git_push`} and `tainted` disables {`net_egress`, `git_push`, `fs_write`}. Controllability still holds, since no reachable state disables an uncontrollable event. The lesson is the one synthesis is for: P_4 is vacuous under P_2, and a hand-written engine would have carried the dead rule forever.

1.21 *(exercise on page 43)* A protected-run receipt: tree hash, a suite hash the principal controls, runner or image hash, command, timestamp and nonce, exit code, and a signed result. Closure verifies every binding and the freshness. This resists free-text and stale-result manipulation; it is not universally Goodhart-proof, because the suite may be incomplete or poisoned. The honest theorem is syntactic provenance and non-replay, not semantic correctness.

1.22 *(exercise on page 43)* Notes preserve explicit goals, rationale, observations, and artifact pointers. They cannot preserve unflushed edits, process memory, open handles, provider caches or hidden state, or a latent next action the model never externalized. A successor can understand the story and still be unable to resume the exact computation.

1.23 *(exercise on page 43)* The append commits. A later delete or regressive count also commits and enters the operation log. Only then does the subscriber observe the count drop and

raise or compensate. The violation is durable at the second commit; the monitor proves detection, not prevention.

1.24 *(exercise on page 43)* A content-addressed task capsule sealed at a safe tool or message boundary: base tree and worktree blobs, environment and tool manifests, open claims with epochs, commitments and acceptance criteria, exact tool input and output, pending external operations with their idempotency keys, decisions and hypotheses, a provenance-preserving transcript or compaction, and external references. Flush the ledger and quiesce mediated effects before sealing. This restores observable task state and evidence; it cannot recover hidden activations, unexposed reasoning, provider-internal caches, or unrecorded intent. Arbitrary-provider recovery is semantic continuation, not bit-identical resurrection.

1.25 *(exercise on page 43)* `delegate` from the initial state with a child scope not contained in the root grant's scope $\{a, b\}$, for instance $\{a, b, c\}$: with the attenuation guard off the grant is created and I4 fails immediately. No other guard can be broken in one step, because effects need the executing phase, settlement needs the verified phase, and a double settlement needs two operations.

1.26 *(exercise on page 43)* One candidate: a contested work unit never settles (`settle` is guarded on the verified phase, but the transition from contested back to a settleable phase is unconstrained). Add the guard, rerun the search, and confirm the state count and that the mutation suite finds a trace with the guard off; if the checker finds none, the invariant was already implied and does not earn its row.

1.27 *(exercise on page 43)* One daemon connection handles every claim and lock read and write; no `read_uncommitted`; respond only after commit; no out-of-band writer. A write-capable external connection violates the first and the fourth at once, the out-of-band-writer assumption most literally. A read-only connection need not break write linearizability, but it observes under different timing and API semantics.

1.28 *(exercise on page 43)* A invokes `claim(k)` and commits at c_1. B and C overlap; B's denied read commits at c_2, A releases at c_3, C's acquire commits at c_4. One valid order is A-acquire, B-denied, A-release, C-acquire. If B and C overlap around the release, either may linearize first, as long as each result matches the state at its point and non-overlapping operations keep their real-time order.

1.29 *(exercise on page 43)* Generate identical state-machine traces against both bindings: schema initialization and migrations, acquire, reclaim, release, and expiry, overlapping calls, transactions and rollback, busy and lock behavior, encodings, pragmas, crash and reopen, fault injection. After every step compare returned values, errors, serialized rows, schema, and recovery state, including the deployed binary in CI. A seeded fuzz misses divergences that depend on wall-clock timing, on a pragma the fuzz never sets, or on a crash at a point the fuzz never interrupts; targeted fault injection and pragma sweeps catch those. No finite suite makes I11 a theorem: that needs a formal refinement or equivalence proof, or the same verified binding on both sides. Label the suite conformance evidence.

1.30 *(exercise on page 43)* The port stays claimed and a live or ghost session row exists, but no commitment says what work owns them. Other agents see a busy port and a presence they cannot reconcile; the failed starter may retry into its own leftovers.

1.31 *(exercise on page 44)* Expose one `begin_work` endpoint that opens `BEGIN IMMEDIATE`, validates every input, inserts the claim, session, commitment, and outbox rows, and commits once; on any error it rolls back all of them. A saga would release the claim and mark or delete the session when the commitment insert failed, and each compensation can itself fail or crash between steps. All three rows live in one SQLite file, so the local transaction is

both simpler and stronger.

1.32 *(exercise on page 44)* A later same-user tamperer of the database file, or a remote synchronizer that cannot rewrite an externally anchored root. The same-user process is out of scope at the substrate; the adversary becomes real at the cross-machine economy layer. Without an independent anchor or verifier, the writer can rewrite the log and recompute the chain.

1.33 *(exercise on page 44)* The release path compares the supplied actor identity to the lock owner. If a legacy endpoint accepts a self-asserted identity, the attacker submits the victim's string, the equality check passes, and the owner-scoped delete executes. The lock rule is correct relative to its input; I12 failed to authenticate the input. Complete credential and peer binding must precede every security-relevant write.

1.34 *(exercise on page 44)* The minimum useful wall is a daemon under a separate UID (or a VM), a daemon-owned database and key, a Unix-socket ACL plus kernel peer credentials, and brokered git, filesystem, and network effects. An OS keychain protects a key at rest but not arbitrary signing requests from an equally authorized same-user process; sealed memory does not stop same-UID tracing or injection without an OS isolation policy; external root anchoring detects a later log rewrite but does not prevent it. No trusted platform module is required for strong local separation: separate privilege and complete mediation are. After the sandbox the same-user adversary moves to the mediated-effects row, where side channels the sandbox does not enumerate remain in scope.

References

[1] James P. Anderson. Computer Security Technology Planning Study. ESD-TR-73-51, U.S. Air Force Electronic Systems Division, 1972. (The origin of the *reference monitor* concept.) Cited on pages 5 and 44.

[2] Butler W. Lampson. Protection. *ACM SIGOPS Operating Systems Review*, 8(1):18–24, 1974. Cited on pages 5 and 44.

[3] Jerome H. Saltzer and Michael D. Schroeder. The Protection of Information in Computer Systems. *Proceedings of the IEEE*, 63(9):1278–1308, 1975. Cited on pages 5, 9, and 44.

[4] Fred B. Schneider. Enforceable Security Policies. *ACM Transactions on Information and System Security*, 3(1):30–50, 2000. Cited on pages 19, 23, and 44.

[5] Feng Lin and W. Murray Wonham. On observability of discrete-event systems. *Information Sciences*, 44(3):173–198, 1988. Cited on page 24.

[6] Peter J. Ramadge and W. Murray Wonham. Supervisory Control of a Class of Discrete Event Processes. *SIAM Journal on Control and Optimization*, 25(1):206–230, 1987. (Controllability and the supremal controllable sublanguage — the general boundary of which this chapter's Synchronous State Decidability theorem is the commit-only instance.) Cited on pages 21, 22, and 46.

[7] Peter J. Ramadge and W. Murray Wonham. The Control of Discrete Event Systems. *Proceedings of the IEEE*, 77(1):81–98, 1989. Cited on page 46.

[8] Andrew J. I. Jones and Marek Sergot. On the Characterisation of Law and Computer Systems: The Normative Systems Perspective. In *Deontic Logic in Computer Science*, Wiley, 1993. Cited on pages 17 and 45.

[9] Jim Gray and Andreas Reuter. *Transaction Processing: Concepts and Techniques*. Morgan Kaufmann, 1992. (Atomicity/consistency vs. durability — the ground for I1a/I1b.) Cited on pages 10 and 44.

[10] C. Mohan, Don Haderle, Bruce Lindsay, Hamid Pirahesh, and Peter Schwarz. ARIES: A Transaction Recovery Method Supporting Fine-Granularity Locking and Partial Rollbacks Using Write-Ahead Logging. *ACM Transactions on Database Systems*, 17(1):94–162, 1992. Cited on page 44.

[11] D. Richard Hipp et al. SQLite Write-Ahead Logging. SQLite Documentation, 2010. `https://www.sqlite.org/wal.html` (`synchronous=NORMAL` "commits might lose durability following a power loss.") Cited on pages 10 and 44.

[12] Martin Kleppmann. *Designing Data-Intensive Applications*. O'Reilly, 2017. (Single-leader as the right default; consensus only when forced.) Cited on pages 5 and 44.

[13] Maurice P. Herlihy and Jeannette M. Wing. Linearizability: A Correctness Condition for Concurrent Objects. *ACM Transactions on Programming Languages and Systems*, 12(3):463–492, 1990. Cited on pages 34 and 45.

[14] Michael J. Fischer, Nancy A. Lynch, and Michael S. Paterson. Impossibility of Distributed Consensus with One Faulty Process. *Journal of the ACM*, 32(2):374–382, 1985. Cited on pages 5 and 44.

[15] Tushar Deepak Chandra and Sam Toueg. Unreliable Failure Detectors for Reliable Distributed Systems. *Journal of the ACM*, 43(2):225–267, 1996. Cited on pages 42 and 45.

[16] Leslie Lamport. Time, Clocks, and the Ordering of Events in a Distributed System. *Communications of the ACM*, 21(7):558–565, 1978. Cited on page 39.

[17] Philip R. Cohen and Hector J. Levesque. Intention Is Choice with Commitment. *Artificial Intelligence*, 42(2–3):213–261, 1990. Cited on pages 26 and 45.

[18] Hector Garcia-Molina and Kenneth Salem. Sagas. In *ACM SIGMOD International Conference on Management of Data*, 1987. Cited on pages 35, 44, and 45.

[19] Pierre-Paul Grassé. La reconstruction du nid et les coordinations interindividuelles…: la théorie de la stigmergie. *Insectes Sociaux*, 6(1):41–80, 1959. Cited on pages 16 and 45.

[20] David Gelernter. Generative Communication in Linda. *ACM Transactions on Programming Languages and Systems*, 7(1):80–112, 1985. Cited on pages 16 and 45.

[21] Marco Dorigo and Gianni Di Caro. The Ant Colony Optimization Meta-Heuristic. In *New Ideas in Optimization*, McGraw-Hill, 1999. Cited on pages 16 and 45.

[22] Ralph C. Merkle. A Digital Signature Based on a Conventional Encryption Function. In *Advances in Cryptology — CRYPTO '87*, 1987. Cited on pages 29 and 45.

[23] Stuart Haber and W. Scott Stornetta. How to Time-Stamp a Digital Document. *Journal of Cryptology*, 3(2):99–111, 1991. Cited on pages 29 and 45.

[24] Jeffrey O. Kephart and David M. Chess. The Vision of Autonomic Computing. *IEEE Computer*, 36(1):41–50, 2003. Cited on page 45.

[25] Daniel J. Bernstein, Niels Duif, Tanja Lange, Peter Schwabe, and Bo-Yin Yang. High-Speed High-Security Signatures. *Journal of Cryptographic Engineering*, 2(2):77–89, 2012. (Ed25519 — the signature scheme underlying the authorization chain.) Cited on page 45.

[26] Bruno Blanchet. Modeling and Verifying Security Protocols with the Applied Pi Calculus and ProVerif. *Foundations and Trends in Privacy and Security*, 1(1-2):1–135, 2016. (The class of tool in which the authorization chain's secrecy and integrity properties are mechanically verified.) Cited on page 45.

[27] Michael T. Nygard. *Release It! Design and Deploy Production-Ready Software*. 2nd ed., Pragmatic Bookshelf, 2018. (The bounded busy-wait as a bulkhead / timeout-budget under contention.) Cited on page 45.

[28] Arnar Birgisson, Joe Gibbs Politz, Úlfar Erlingsson, Ankur Taly, Michael Vrable, and Mark Lentczner. Macaroons: Cookies with Contextual Caveats for Decentralized Authorization in the Cloud. In *Network and Distributed System Security Symposium (NDSS)*, 2014. (Attenuation-only bearer credentials with third-party caveats — the enforcement-gate construction.) Cited on page 36.