The Witness Plane

Admissibility, Verification, and Commitment in Event Systems

Flyxion
Independent Researcher

September 2026

Orientation

An event system does not become trustworthy merely because it preserves an append-only log, executes deterministic code, or attaches hashes to its outputs. These mechanisms establish that something was recorded or computed; they do not establish that the input was admissible, that the computation was authorized, that its obligations were satisfied, or that its result was fit to alter the operational world. This book presents Spherepop as an architecture for keeping those propositions separate.

The central pipeline is

$$\begin{aligned}\mathsf{POP} &\longrightarrow \mathsf{REFUSE} \longrightarrow \mathsf{BIND} \\ &\longrightarrow \mathsf{TRANSFORM} \longrightarrow \mathsf{VERIFY} \longrightarrow \mathsf{COLLAPSE},\end{aligned}$$

while LEDGER is reinterpreted as the persistent witness plane beneath all six transitions. The ledger records encounters, refusals, bindings, executions, verdicts, failed commitments, successful commitments, and recovery operations. Operational history advances only when a verified candidate satisfies the policy-relative collapse guard.

Its philosophical thesis is that occurrence, memory, admissibility, execution, verification, and commitment are different achievements. Its technical thesis is that a system becomes more inspectable when each achievement is represented by a typed transition, a declared policy, a versioned witness, and a distinct failure vocabulary. Its practical thesis is that distributed systems should target accountable commitment rather than the fiction of absolute warrant.

Composition and method

The book comprises thirty-five chapters, four interludes, and eight technical appendices. Its argument moves from conceptual separations through a typed transition system, then into concurrency, external effects, recovery, implementation, evaluation, and institutional consequence. Formal chapters pair definitions and positive results with counterexamples, implementation interpretations, and explicit statements of what the machinery does not establish. The appendices consolidate the notation, operational semantics, proof obligations, serialization rules, example records, recovery cases, security boundaries, and open research programme needed to make those claims inspectable as a whole.

Canonical status

The Spherepop pipeline and its event semantics are the principal subject of the book and form one focused elaboration of the author's broader Spherepop research programme [11]. Connections to RSVP, Kinetic-Event Synthesis, Distinction Holonomy Theory, and MEM|8 are presented as bounded integration hypotheses, not as foundations on which the core architecture depends. Detailed KES mathematics not independently established in its source manuscripts remains explicitly provisional. The architecture specified here is substantially complete within its declared scope, while the extensions collected in the research programme remain available for later development without being presupposed by the results already stated.

Contents

Part I

The Separations

Chapter 1

The Event Is Not Yet an Action

1.1 Chapter thesis

Most systems that process events collapse a long sequence of distinct achievements into a single word: "handling" the event. A request arrives, is parsed, is executed, is checked, and its effect becomes part of the system's permanent record—and all of this is described, and often implemented, as one operation. The request "was handled." The claim "was processed." The instruction "went through."

This chapter argues that the compression is a category error, and that the error is not merely stylistic. Receiving an event, recording that it occurred, parsing its structure, executing a computation over it, validating the result against a standard, and committing that result to the system's operational history are six different achievements. Each can succeed while the others fail. A malformed request can be received and recorded without being parsed. A request can be parsed and executed without its output ever being checked. An output can be checked and found wanting without anything having been committed. And, critically, an output can be checked, found acceptable, and still not be the thing that was committed, if the committing step observes

a world that changed between verification and commitment.

Treating these as one operation does not make a system simpler. It makes a system's failures unintelligible, because there is no vocabulary left for describing which of the six achievements actually happened when something goes wrong. This book exists to supply that vocabulary, and this chapter exists to motivate why it is needed before any formal machinery is introduced.

The six achievements are an introductory decomposition rather than a premature one-to-one transcription of the Spherepop operators. Later chapters refine the space between encounter and execution through REFUSE and BIND, and show that LEDGER is not merely another sequential achievement but the witness plane beneath them all.

1.2 Core distinctions

1.2.1 Occurrence versus endorsement

An event can occur—arrive at a system's boundary, be observed, be timestamped—without the system endorsing it in any sense stronger than "this was seen." Occurrence is a fact about the world: something crossed a boundary at a given time, from a given source, with a given content. Endorsement is a fact about the system: it has decided the occurrence is fit to influence what happens next. Confusing the two produces the naive assumption that anything a system has recorded is something the system has approved. It is not. A system can, and generally should, keep a durable record of things it explicitly declines to approve.

1.2.2 Evidence versus warrant

A closely related distinction concerns what an occurrence is good for. An occurrence can serve as evidence—as something that bears on a later decision—long before it constitutes a warrant for that decision.

A single unconfirmed report is evidence that something may be true. It is not, by itself, a warrant for acting as though it is true. Systems that conflate evidence with warrant tend to act on the first plausible signal they receive; systems that keep the distinction can accumulate evidence, let it mature, and act only once a declared threshold of warrant has been met.

1.2.3 Computation versus authorization

A computation can execute correctly—produce exactly the output its code specifies, given its input—without having been authorized to run at all, or without its output being authorized to take effect. This distinction matters most where it is most often ignored: in the assumption that if a piece of code ran successfully, whatever it produced must be legitimate. Correct execution answers the question "did the code do what it says." It does not answer the question "was the code allowed to do this, to this input, on this system's behalf." Those are independent questions, and a system that only asks the first one has no way to refuse a well-executed but unauthorized computation.

1.2.4 History versus audit

Finally, this book distinguishes two kinds of record-keeping that are frequently merged into one log. **Operational history**, written H_t, is the record of what the system has committed to—the sequence of transitions that define its current authoritative state. **Audit history**, written L_t, is the broader record of everything the system has encountered, attempted, refused, verified, failed to verify, and eventually committed or declined to commit. Every element of H_t should correspond to something in L_t; the converse should not hold. Most of what a well-instrumented system remembers is not, and should not be, part of what it has committed to.

These two histories are formalized fully beginning in Chapter 4. For now, it is enough to notice that a system with only one log—

one that does not distinguish “this happened” from “this is now authoritative”—cannot represent the difference between an attempt and a success without inventing ad hoc conventions for doing so after the fact.

1.3 Principal examples

The four distinctions above are abstract until they are seen doing work. The following four cases, drawn from different domains, show what goes wrong when occurrence, evidence, correct execution, and historical record are each mistaken for something stronger than they are.

1.3.1 A malformed network request

A server receives a request with an invalid content-length header. The request occurred—it arrived, it consumed a connection, it can be logged. It was never parsed, because parsing requires a well-formed envelope. A system that logs the request as “received” and stops there has correctly distinguished occurrence from endorsement. A system that treats “received” as equivalent to “in the queue for processing” will eventually try to process something that cannot be processed, and will need a second, informal layer of error handling to recover from a distinction it should have made at intake.

1.3.2 A scientifically unsettled claim

A single study reports an effect. The report is evidence: it should be retained, cited, and available to later inquiry. It is not a warrant for treating the effect as established, prescribing based on it, or retracting prior guidance. The distinction between evidence and warrant is exactly what allows a claim’s status to change over time—as further studies arrive—without requiring anyone to have acted prematurely on the first report, and without requiring the first report to be deleted once it is superseded. Chapter 2 returns to this case in

detail as the paradigm instance of being admissible to memory before being admissible to action.

1.3.3 A stale financial instruction

An instruction to transfer funds is authorized at the moment it is issued, against the account balance and policy in effect at that moment. If the instruction is executed correctly but only after a delay, during which the balance or policy has changed, correct execution of the original instruction is not the same as authorization to execute it now. A system that checks correctness of execution but not freshness of authorization will faithfully carry out instructions that are no longer warranted, precisely because it has conflated two questions that must be asked separately and in order.

1.3.4 An irreversible actuator command

A command to fire a valve, release a mechanism, or otherwise change the physical world cannot be undone by deleting a log entry. Here the history-versus-audit distinction is sharpest: a system can and should preserve full audit history of every candidate command it considered, refused, revised, and eventually issued—but only one of those records corresponds to an actual physical consequence, and the system's own bookkeeping must make it possible to tell, without ambiguity, which one that was. Chapter 17 develops the further complication that even a correctly authorized commit does not guarantee the physical action it describes was completed exactly once.

Across all four cases, the failure mode is the same: something true at one stage—occurrence, evidence, correct execution, or bare record—is treated as though it settled a later, stronger question it was never equipped to settle.

1.4 Proof obligations and status

Foundational but non-theorem-bearing. This chapter's role is to stabilize vocabulary and motivate the separations that later chapters formalize. It supplies typed examples and counterexamples—the four cases above are chosen so that each isolates exactly one collapsed distinction—but it does not, and should not, attempt to state or prove a formal result. No transition system has yet been defined; there is nothing yet for a proposition to be about. The temptation to open a book on formal architecture with an early theorem, for the sake of appearing rigorous from page one, should be resisted here specifically: a theorem proved before its terms are defined is not rigor, it is the appearance of rigor purchased at the cost of actually defining anything. The formal apparatus begins in Chapter 5, once the event envelope itself has a definition to be precise about.

Chapter 2

Admissible to Memory, Forbidden to Action

2.1 Chapter thesis

A claim can deserve preservation before it deserves any influence over what a system does. This is the plainest way to state the Reversibility Gate principle that grounds the present chapter: epistemic admission, letting something into the record, and operational admission, letting something change what the system treats as settled, are separate decisions, and a system is entitled to make one without making the other. The scientific claim of Chapter 1 illustrated this once. This chapter generalizes it and gives it a name.

2.2 Three times

Confusion between memory and action is usually a confusion about time. Three moments need to stay distinct.

Observation time is when a system first encounters a claim, request, or signal. **Decision time** is when the system evaluates that claim against a current policy and history, and settles what disposition to give it. **Action time** is when, if ever, the claim's

consequence is allowed to change something the system or the world depends on.

These three moments can be far apart, and the distance between them is not a defect to be engineered away. A claim observed today may not reach decision time until more corroborating evidence exists. A claim decided favorably at decision time may not reach action time until a further authorization, a scheduled window, or a resource becomes available. The error this chapter is built to prevent is treating the elapsed time between observation and decision, by itself, as sufficient grounds for advancing to action. Maturity of evidence is necessary for many kinds of action; it is not sufficient for any of them, because sufficiency also depends on authorization, capability, and the state of the world at action time, none of which observation time or the passage of time can supply on its own.

2.3 Layered admission

A single Boolean property, admitted or not admitted, cannot represent what a well-run system actually does with a claim. This chapter proposes four separate predicates instead.

Epistemic admission, written Ep, holds when a claim is retained in the system's durable record as something worth knowing about, independent of whether anything further is done with it.

Computational admission, written Comp, holds when a claim is cleared to have some computation, deliberation, or transformation process run over it.

Operational admission, written Op, holds when a claim, or the result of a computation over it, is admitted to become part of the system's authoritative operational state, the state the system treats as current and relies on for further decisions.

Irreversible admission, written Ir, holds when a claim's consequence is admitted to be enacted in a way that has effects the system's own record cannot later undo, typically because those effects

occur outside the system's record entirely.

A system that only tracks one of these, usually the last one it happens to notice, cannot express most of the states a real process passes through. The remainder of this chapter establishes that the four predicates are logically independent of one another: none is implied by any other, in either direction, without an additional architectural guarantee doing the work.

2.4 Proof obligations and status

Load-bearing counterexample result. This section proves pairwise non-implication among Ep, Comp, Op, and Ir. For each of the six unordered pairs, two witnesses are given: one state satisfying the first predicate and not the second, and one state satisfying the second and not the first. Each witness states the assumptions under which it is admissible, since these examples are drawn from general domains rather than from a single governed system, and a different set of institutional or architectural assumptions can change which witnesses apply.

2.4.1 Pair $(\mathrm{Ep}, \mathrm{Comp})$

$\mathrm{Ep} \wedge \neg\mathrm{Comp}$: a claim is retained purely as evidence, with no computation ever cleared to run over it. This is the ordinary case of an evidence-only refusal disposition, assuming the governing policy treats retention and computational clearance as separately decidable.

$\mathrm{Comp} \wedge \neg\mathrm{Ep}$: a computation runs over an input that is never itself durably retained. A streaming system that computes an aggregate statistic from raw sensor readings and discards the raw readings once the statistic is produced admits the input to computation without admitting it to the durable record, assuming the discard is a deliberate storage or privacy policy rather than an unintended loss.

2.4.2 Pair $(\mathrm{Ep}, \mathrm{Op})$

$\mathrm{Ep} \wedge \neg\mathrm{Op}$: the scientific claim from Chapter 1, retained in the literature but never adopted into settled guidance, assuming the domain in question distinguishes publication from adoption.

$\mathrm{Op} \wedge \neg\mathrm{Ep}$: a default judgment rendered because a party failed to appear changes a court's operative record without any evidence about the merits having been admitted, assuming the governing procedure permits judgment on absence rather than requiring an evidentiary record in every case.

2.4.3 Pair $(\mathrm{Ep}, \mathrm{Ir})$

$\mathrm{Ep} \wedge \neg\mathrm{Ir}$: most retained claims, including the scientific claim again, never lead to any irreversible action at all.

$\mathrm{Ir} \wedge \neg\mathrm{Ep}$: a mechanical overcurrent interlock trips a circuit breaker directly, producing a physical change the software record cannot itself undo before any software-level record of the triggering condition exists, assuming the interlock is intentionally designed to act at electromechanical speed ahead of, rather than through, any logging path.

2.4.4 Pair $(\mathrm{Comp}, \mathrm{Op})$

$\mathrm{Comp} \wedge \neg\mathrm{Op}$: a simulation or dry-run mode executes the full computation over a candidate input and never commits the result, by design, assuming the system explicitly separates speculative execution from commitment.

$\mathrm{Op} \wedge \neg\mathrm{Comp}$: an authorized administrator directly edits the operational record to correct a stuck state, without any submitted claim having been computed over, assuming the governing policy grants a privileged manual-override path outside the ordinary pipeline. This witness is exactly the gap that the collapse guard introduced in Chapter 15 is designed to close inside Spherepop itself; outside a system that enforces such a guard, nothing prevents it.

2.4.5 Pair $(\mathrm{Comp}, \mathrm{Ir})$

$\mathrm{Comp} \wedge \neg \mathrm{Ir}$: an internal report is generated by a computation and only ever displayed, never triggering any external effect, assuming display is not itself classified as an irreversible effect in the domain at hand.

$\mathrm{Ir} \wedge \neg \mathrm{Comp}$: a person presses a physical emergency-stop button on their own judgment, with no computational process mediating the decision, assuming the domain permits direct human action outside the computational pipeline, as most safety-critical domains deliberately do.

2.4.6 Pair $(\mathrm{Op}, \mathrm{Ir})$

$\mathrm{Op} \wedge \neg \mathrm{Ir}$: an order is marked approved in a system's authoritative internal state before the corresponding shipment has actually occurred, corresponding to a completed prepare step awaiting actuation, assuming the domain separates internal commitment from external fulfillment.

$\mathrm{Ir} \wedge \neg \mathrm{Op}$: an actuator fires and the physical effect occurs, but the process that would append the corresponding commit to the operational history crashes before completing, leaving an orphaned effect the system's own record does not yet reflect, assuming actuation and record-keeping are not implemented as one atomic step, which Chapter 17 argues they generally cannot be.

2.4.7 Summary and scope

The twelve witnesses above establish that no predicate among Ep, Comp, Op, and Ir implies, or is implied by, any other as a matter of definition alone. This is a claim about the four predicates taken in isolation, prior to any particular system's governance. It is not a claim that a well-designed system should tolerate all twelve gaps equally. Several of the witnesses above, most sharply the manual-override and orphaned-effect cases, name exactly the kind of gap that

later chapters close with an explicit architectural guard: the collapse guard of Chapter 15 forecloses $\mathrm{Op} \wedge \neg\mathrm{Comp}$ within Spherepop by construction, and the prepare/actuate/finalize protocol of Chapter 17 gives $\mathrm{Ir} \wedge \neg\mathrm{Op}$ a named, recoverable status rather than leaving it as silent data loss. The purpose of proving independence here, before any such guard has been defined, is to make clear that these implications are achievements of specific architectural choices, not free consequences of the four predicates existing at all.

Chapter 3

Against Absolute Warrant

3.1 Chapter thesis

It is tempting, once a system distinguishes occurrence from endorsement, evidence from warrant, and epistemic admission from operational admission, to conclude that whatever finally clears every gate must be correct. This chapter argues against that conclusion directly. A transition that has been checked against every obligation the system declared, using a verifier the system trusts, at a history head the system confirms is current, remains conditional on the policy, the specification, the evidence, and the trusted computing base that produced those checks. None of those four can certify itself as an absolute ground of warrant. A verified transition is warranted. It is not, and cannot be, warranted absolutely.

This distinction is not a hedge added for intellectual modesty. It is load-bearing for how the rest of this book treats the word "verified." Later chapters build an elaborate apparatus for discharging obligations, and it would be easy to read that apparatus as a machine for producing certainty. It is not that. It is a machine for producing a specific, inspectable, revisable kind of sufficiency, and this chapter exists to say precisely what that sufficiency does and does not include, before the apparatus is built.

3.2 Policy-relative sufficiency

Call a transition's warrant sufficient, relative to a policy version σ_t, a confirmed history head $h(H_t)$, and a declared verification regime, when the transition has satisfied every obligation that σ_t requires, against the state that $h(H_t)$ identifies, under the specific procedures the verification regime names. Each of the three qualifiers is doing real work, and each is a place where the warrant can be sufficient in exactly the sense defined and still be wrong.

Sufficiency relative to σ_t means sufficiency relative to whatever obligations someone wrote into that policy. It says nothing about whether the policy asked the right questions. Sufficiency relative to $h(H_t)$ means the check was performed against a specific, agreed state of the world as the system understood it at that moment. It says nothing about whether the system's history correctly reflects the world outside it. Sufficiency relative to a declared verification regime means the named procedures were followed and produced the stated verdict. It says nothing about whether those procedures are capable, in principle, of detecting the kind of error that happens to be present.

Later chapters give each of these qualifiers a full formal treatment: σ_t and its versioning in the chapters on binding and obligation generation, $h(H_t)$ in the chapters on the collapse guard and concurrency, and the verification regime in the chapters on verifier independence and correlated failure. This chapter's job is only to fix the shape of the claim before that machinery exists, so that later precision is read as precision about a bounded thing, not as a route to an unbounded one.

3.3 Residual fallibility

Five distinct sources of residual fallibility survive even a maximally careful verification regime. None of them is a flaw in the Spherepop architecture specifically; each is a property of any system that must decide, using finite declared rules, whether to trust something.

3.3.1 Specification error

An obligation set can be internally consistent, fully discharged, and still wrong, because it encoded the wrong requirement. A binding that requires a bridge design to tolerate a given load and never mentions resonance has not failed at verification when a resonant failure occurs; it failed earlier, at the point where the obligation set was written. No amount of rigor at the verification stage recovers a question the binding stage never asked.

3.3.2 Compromised authorities

A verification regime names a verifier, and naming a verifier is an act of trust in whoever built, deployed, and maintains it. If that authority is compromised, through insider action, supply-chain interference, or a captured governance process, the verifier can return exactly the verdict its declared procedure would produce on a correct implementation, while running a different, compromised implementation the declared procedure never actually checked. The check that a verdict was reached honestly is a separate concern from the check that an obligation was satisfied, and this book returns to it directly in the chapter on the threat model.

3.3.3 Incomplete models

Every obligation is stated in terms of some model of the domain it governs: a model of financial risk, a model of structural load, a model of how a piece of software will be used. A model can be internally coherent and still leave out a variable that turns out to matter. Verification against an incomplete model produces a verdict that is entirely correct with respect to the model and entirely silent about the part of reality the model omitted.

3.3.4 Unknown unknowns

A closely related but distinct failure is the absence of any obligation at all covering a class of risk nobody anticipated. An incomplete model at least represents its domain with some variables missing. An unknown unknown is a domain nobody thought to model in the first place. No obligation set can be verified against a condition it was never written to check, and no verification procedure can be faulted for failing to catch what it was never told to look for.

3.3.5 Valid proofs of the wrong proposition

The most subtle failure mode is not an error in reasoning but a mismatch between what was proven and what was needed. A verifier can construct a fully valid derivation that obligation O_i is satisfied, where O_i itself, though correctly proven, is not the proposition that mattered. This differs from specification error in degree rather than kind: specification error is usually a gap noticed only in hindsight, while a valid proof of the wrong proposition can occur even when the specification looks adequate on its face, because the gap lies in the translation from the informally intended requirement to its formal encoding as O_i. Interlude III returns to this exact failure as a full worked case.

3.4 Proof obligations and status

Limitation-setting rather than theorem-bearing. This chapter's purpose is to bound the scope of every sufficiency claim the rest of the book makes, not to prove that absolute warrant is impossible in every formal system whatsoever. That broader claim is not attempted here and would not be established by the five cases above even if it were: each case is a countermodel showing that a particular route to certainty fails in this architecture's setting, not a general impossibility result about verification as such. The chapters that follow are permitted to state exactly what a discharged obligation

guarantees; they are not permitted, and this chapter is the place that forecloses it, to describe collapse, discharge, or a passing verdict using the vocabulary of certainty. Chapter 14 revisits this same territory once the formal verification apparatus exists and states the narrower, precise non-claim illustrated by counterexamples there: a verifier's own successful operation cannot certify the adequacy of the obligation set it was checking against.

Chapter 4

The Witness Plane

4.1 Chapter thesis

The natural first reading of a six-stage pipeline puts the ledger at the end: an event is popped, refused or bound, transformed, verified, and finally, if it survives, collapsed into history, with the ledger appearing afterward as a record of what happened. That reading is wrong, and this chapter replaces it. The ledger is not the last box. It is the substrate that witnesses every boundary the pipeline crosses, including every boundary a candidate fails to cross. A refused event, a failed transformation, an inconclusive verification, and a stale collapse attempt all leave a mark on the ledger without ever touching the system's operational history. The ledger's completeness does not depend on anything reaching the end of the pipeline. It depends only on the pipeline never taking a step it does not record.

This reframing matters for a concrete reason. If the ledger only appeared after collapse, an observer would have no durable trace of anything the system considered and declined. Every refusal, every failed proof, every stale candidate would be as if it never occurred, and the distinctions built up across Chapters 1 through 3—occurrence against endorsement, evidence against warrant, sufficiency against certainty—would have nowhere to be recorded once a decision went

the other way. The witness plane is where those distinctions become durable facts rather than momentary judgments.

4.2 Two histories

Two sequences are maintained side by side. The audit history, L_t, is extended by exactly one record at every step the system takes, regardless of the step's outcome:

$$L_{t+1} = L_t \parallel \ell_t.$$

The operational history, H_t, is extended only when a step succeeds at the one transition authorized to extend it:

$$H_{t+1} \in \{H_t,\, H_t \parallel \kappa_t\}.$$

Which of the two alternatives obtains is not a matter of interpretation after the fact. It is determined by the transition itself: every transition type other than a successful collapse leaves H untouched, and a successful collapse extends it by exactly one commit object κ_t. Later chapters define each transition precisely enough to state this as a proven property rather than an assumption. The Early-Stage Non-Interference result of Chapters 6 through 10 covers the first five transition types; the collapse transition itself, defined in Chapter 15, is the only one that can produce the second alternative above.

Both sequences are append-only. Nothing already written to L_t or H_t is later edited or removed. Redaction, treated fully in Chapter 19, changes the availability of a payload a record refers to; it does not remove the record itself or alter the sequence of hashes chaining prior records together.

4.3 The non-converse

The relationship between the two histories runs in one direction only. Every operational commitment must have a corresponding audit witness: each $\kappa \in H_t$ arose from some step, and that step wrote a record to L_t documenting it. The converse does not hold. Most of what L_t contains is not, and was never meant to become, an operational commitment. A quarantined event, a repair request, a candidate that failed verification, a stale collapse attempt rejected at the head check: each produces a durable audit record, and none produces a κ. An audit ledger that only ever equaled the operational history in content would not be an audit ledger; it would be a second copy of the same thing. The value of the witness plane is precisely that it is larger than what it witnesses.

4.4 Proof obligations and status

Load-bearing seed lemma. This section proves, by induction on t, that every commit object in the operational history has exactly one corresponding collapse-witness record in the audit history:

$$\forall \kappa \in H_t, \quad \exists! \, \ell_\kappa^{\text{collapse}} \in L_t.$$

The proof depends on five premises, each of which is either established formally in a later chapter or is a property this book requires of any implementation. They are stated here explicitly so the induction's scope is visible before its steps are read.

(P1) Append-only histories. Neither L_t nor H_t ever loses or overwrites a previously written entry.

(P2) Sole writer. Only the successful branch of the collapse transition can extend H_t, and it extends it by exactly one commit object per successful collapse. This is proved for the five earlier transition types in Chapters 6 through 10 and holds for collapse itself by the definition given in Chapter 15.

(P3) Atomic bidirectional publication. When a collapse succeeds, the append of κ_t to H_t and the append of its collapse-witness record $\ell_{\kappa_t}^{\text{collapse}}$ to L_t occur as one indivisible step. Neither occurs without the other, and no collapse-witness record is created except through such a successful dual append. This final clause makes explicit the no-orphan-witness fact needed by the uniqueness argument.

(P4) Commit freshness. Distinct successful collapses produce distinct commit objects. This follows from the nonce-uniqueness clause of the collapse guard, $\neg\,\text{Spent}(\eta)$, defined in Chapter 15, which prevents any nonce, and therefore any commit object built from it, from being produced twice.

(P5) Typed records. A ledger record of kind collapse-witness is keyed to a specific commit object and cannot serve as a witness for any other commit object. Records of other kinds—refusal records, failed transformation records, failed verification records, and failed collapse records—are never collapse-witness records for any κ, regardless of their content.

4.4.1 Base case

At $t = 0$, $H_0 = \varnothing$. The claim holds vacuously: there is no $\kappa \in H_0$ for which a witness must be found.

4.4.2 Inductive step

Assume the claim holds at step t: every $\kappa \in H_t$ has exactly one collapse-witness in L_t. Consider step $t + 1$ and case split on the transition performed.

Case 1: the transition is not a successful collapse. By (P2), $H_{t+1} = H_t$. By append-only bookkeeping, $L_{t+1} = L_t \parallel \ell_{t+1}$ for whatever record this step produces. This record documents a pop, a refusal, a binding, a transformation, a verification of any verdict, or a failed collapse attempt, and by (P5) none of these record kinds is a

collapse-witness for any commit object. So no existing $\kappa \in H_t$ gains a second witness, and since $H_{t+1} = H_t$, no new κ requiring a witness has appeared. Every $\kappa \in H_{t+1}$ still has exactly the one witness it had in $L_t \subseteq L_{t+1}$, and the claim holds at $t+1$.

Case 2: the transition is a successful collapse producing κ_{t+1}. By (P2) and (P3), $H_{t+1} = H_t \parallel \kappa_{t+1}$ and, in the same step, $L_{t+1} = L_t \parallel \ell^{\text{collapse}}_{\kappa_{t+1}}$. Two things must be checked: that κ_{t+1} now has a witness and has only one, and that no previously witnessed $\kappa \in H_t$ loses its uniqueness in the process.

Existence for κ_{t+1} is immediate: its witness was just appended. For uniqueness, suppose toward contradiction that κ_{t+1} already had a witness in L_t, before this step. By the no-orphan-witness clause of (P3), the earlier witness could only have been created by an earlier successful collapse that simultaneously appended the same commit object to H_t. But by (P4), κ_{t+1} is distinct from every commit object produced by an earlier successful collapse. The supposed prior witness therefore cannot exist, and exactly one is added now.

For every $\kappa \in H_t$, the inductive hypothesis already supplies exactly one witness in $L_t \subseteq L_{t+1}$. The single record added at this step is, by construction, keyed to κ_{t+1} specifically, and by (P5) it cannot double as a witness for any distinct commit object. So no $\kappa \in H_t$ gains a second witness. Every element of $H_{t+1} = H_t \cup \{\kappa_{t+1}\}$ therefore has exactly one witness in L_{t+1}, and the claim holds at $t+1$.

Both cases close the induction, establishing the result for all t.

4.4.3 Scope and forward reference

This lemma is deliberately narrow. It covers only the successful-collapse path under the five named premises, and it says nothing yet about crash recovery, write-ahead durability, or implementations that persist H and L through separate storage systems rather than one atomic operation. Chapter 18 strengthens exactly this result to cover recovery paths and a write-ahead protocol in place of true atomicity, and should be read as extending this proof's structure

rather than replacing it. The seed lemma proved here is what makes that extension possible: any later argument that ledger completeness survives a crash must still reduce, in the crash-free case, to the induction given above.

Interlude I: A Request That Never Became an Act

This interlude is not a chapter. It proves nothing and defines nothing precisely. Its purpose is to walk one request through every stage this book will eventually formalize, so that when the formal apparatus arrives beginning in Chapter 5, it arrives as a description of something the reader has already seen happen, rather than as machinery with nothing yet to grip.

The request

A customer support agent notices that a customer was billed twice for the same transaction. She submits a correction: reverse a duplicate charge of $412.17. Call this arriving event e_0. It is observed at a definite time, from a definite source, the agent's authenticated session, and it carries a definite content: an account identifier, a signed amount, and a stated reason.

Refusal

The system's intake policy, current version σ_t, requires every balance correction to reference the specific original transaction it corrects, by

its own identifier, not merely by a written description. Event e_0 does not carry that reference. The event is not silently dropped and it is not silently accepted. It is refused, and the refusal is typed: this is a schema deficiency, not a policy objection, a capability problem, or a question about the agent's authority. The refusal decision records a disposition of repair, a reason certificate naming the missing field, a stated requirement for what resubmission needs to include, and a window in which resubmission remains straightforward rather than requiring the request to start over. Event e_0 itself is preserved exactly as submitted. Nothing about the refusal erases it; the refusal is itself a new fact added to the record, sitting alongside e_0 rather than replacing it.

Repair

The agent looks up the original transaction, finds its identifier, and resubmits. The new submission is not treated as though it had arrived out of nowhere. It is a descendant: an event that names e_0 as its parent and supplies exactly the missing reference as its repair. Call this descendant e_1. The system checks e_1 against the same requirement that e_0 failed, and this time the requirement is met. The original refusal of e_0 does not become false. It remains a true statement about e_0, evaluated under the policy in effect when e_0 was submitted. What changes is that a new, related event has now cleared the gate the first one did not.

Rebinding

With e_1 admitted, the system attaches the constraints that govern balance corrections of this kind. Because the amount exceeds a threshold set by policy, three obligations are generated before anything is computed or inspected: the adjustment amount must match the discrepancy recorded against the referenced transaction, a second approver distinct from the submitting agent must sign off before the

change takes effect, and the eventual audit trail must reference the original transaction by its identifier. None of these obligations is invented after the fact to justify a result. They are fixed now, as a contract the eventual output will be judged against, whatever that output turns out to be.

Transformation

A kernel built for ledger adjustments computes the actual entry: the account's new balance, given its old balance and the signed correction. The computation is confined to exactly this task. It cannot reach beyond the account in question, cannot consult unrelated records, and produces, alongside its numeric result, a witness of how it arrived there. Nothing about this step changes anything the system currently treats as authoritative. It produces a candidate, not a commitment.

Failed verification

The candidate is checked against the three obligations fixed at binding. The amount matches the discrepancy: satisfied. The audit trail correctly references the original transaction: satisfied. The second approver's signature is present: it is not. The agent submitted alone. This third obligation was never marked as one a policy officer could waive; it is required outright for corrections of this size, precisely because requiring a second, independent party to look at the change is the control this domain uses to catch a class of error, or a class of abuse, that a single reviewer might miss or might commit. The verification returns a verdict of failure, not because anything was computed incorrectly, but because a specific, named, required condition was not met. The system does not treat this as an error in its own machinery, and it does not treat it as evidence that the correction is wrong. It treats it as exactly what it is: one obligation, out of three, not yet discharged.

Reopening

The agent brings the correction to her manager, who reviews the transaction reference and the computed amount and signs. A new event is submitted, again not from nowhere: it names the failed attempt as its parent and supplies the missing signature as its repair. The system processes this descendant through binding and transformation again. The computation reproduces the same numeric result it produced before, because nothing about the account or the correction amount has changed; only the missing approval has been supplied. Verification runs again, and this time all three obligations are satisfied. Nothing was weakened to make this happen. The same requirement that blocked the first attempt is checked again, in full, against the same standard, and this time it passes because the actual condition it demands has actually been met.

Commitment

A collapse proposal is formed: the verified candidate, its verification result, a reference to the current state of the account ledger, the policy under which this collapse is authorized, and a fresh, unused marker preventing this exact proposal from ever being committed twice. The system checks that nothing else has changed the account's ledger state since verification began, checks that the proposal is authorized, checks that the candidate is indeed collapsible, and checks that the posting mechanism is ready to accept the entry. Every check passes. The correction is posted. The account balance changes, and this change now belongs to what the system treats as settled and current, not merely to what it has considered.

What remains afterward

Two records exist afterward, and they are not the same record. The system's operational history has grown by exactly one entry: the

posted correction. The system's audit history has grown by a great deal more: the original request, its refusal, the repaired resubmission, the binding and its three obligations, the first computed candidate, the failed verification naming exactly which obligation was unmet, the second resubmission carrying the manager's approval, the second computation, the successful verification, and finally the commitment itself. An observer who only ever looks at the account's current balance sees one clean change. An observer with access to the full record sees that the change was neither immediate nor uncontested, that it was refused once, repaired once, verified twice, and failed once in between, and that every one of those steps is still there to be inspected, long after the balance itself no longer shows any sign that the first attempt ever happened.

A preview of the graph

Stated informally, ahead of the definitions to come, the identifiers this story touched form a small lineage graph: e_0, refused; e_1, a repair descendant of e_0, admitted, bound, transformed, and found wanting at verification; e_2, a reopening descendant of e_1, bound again, transformed again to the same candidate, verified again and this time passed; and finally a single commit object descending from e_2's verified candidate, the only object in this lineage that the operational history ever came to contain. Chapter 5 begins supplying the formal definitions this graph anticipates, starting with the event envelope itself.

Part II

The Event Calculus

Chapter 5

Events, Envelopes, and Identities

5.1 The event envelope

Every distinction drawn informally in Part I needs a fixed object to attach to once formal machinery begins. That object is the event envelope. Begin with an identity-bearing event core

$$c(e) = \langle u, s, h, p \rangle,$$

where u is a source-scoped occurrence token, s identifies the source, h is a commitment to the content, and p is a stable reference to the payload that content describes. The complete event envelope is

$$e = \langle id, u, t, s, h, p, m \rangle,$$

where id is derived from the core, t is the observation time at which the system first encountered the occurrence, and m is auxiliary metadata that does not itself bear on identity or admissibility.

The occurrence token u distinguishes two intentionally distinct requests that happen to have identical payloads, while remaining stable when a client retries the same occurrence after losing an acknowl-

edgement. It must be unique within the source's identity domain and authenticated as part of the source's submission. Observation time t cannot play this role because a retry is observed at a different time from its first arrival.

Two further design choices are worth making explicit. First, the envelope separates content commitment, h, from payload reference, p. The commitment is what the rest of the architecture reasons about and incorporates into later structures; the reference is how an authorized party locates the actual bytes when it needs them. This separation is what later makes redaction coherent in Chapter 19: a payload can become unavailable at its stable reference without touching the commitment already incorporated into every record built on top of it.

Second, t and m are deliberately excluded from the identity core. Layout hints, display preferences, receipt timestamps, or free-text notes belong in m precisely because they must never be capable of changing whether two submissions denote the same occurrence. Placing something in m is a declaration that it does not matter to the identity or admissibility questions this machinery answers. If a field later proves relevant to either, it was misclassified and must migrate into a versioned identity-bearing schema.

5.2 Domain-separated identity

An event's identifier is computed from its identity-bearing core, not assigned and not computed from a structure already containing that identifier. Given a canonical encoding function encode and a fixed domain tag domain_e reserved for event cores specifically,

$$id(e) = \mathsf{Hash}\big(\mathsf{domain}_e \parallel \mathrm{encode}(c(e))\big).$$

The domain tag exists to prevent a coincidence that has nothing to do with cryptographic weakness: two structurally different kinds of object, an event core and a ledger record, for example, could encode

to the same bytes if nothing distinguished which kind of object was being hashed. Prefixing every identifier construction in this book with a fixed, reserved tag specific to the kind of object being identified prevents cross-domain equality by shared encoding, independently of the collision resistance of Hash.

The encoding must be canonical, meaning that a given well-typed core has exactly one encoding, not several equally valid ones. Raw concatenation without canonical or length-prefixed encoding can make two different logical structures produce the same byte string, the same failure mode as writing $ab \parallel c$ and $a \parallel bc$ and expecting them to be distinguishable without a delimiter. Section 5.4 makes precise what canonical encoding is required to guarantee, and what it is not asked to guarantee.

5.3 Equivalence and duplication

Several notions of sameness apply to events, and failure to keep them apart produces either false alarms or missed ones.

Core equality holds when two event cores encode to the identical byte string. Given canonical encoding, this is the strongest notion tracked by $id(e) = id(e')$, subject to the collision-resistance assumption.

Full-envelope equality additionally requires equality of observation time and auxiliary metadata. Two retry arrivals can therefore share an event identifier without being the same observed envelope instance: the later arrival has its own receipt time and may carry different non-identity metadata.

Semantic equivalence is broader and policy-relative. Two different identity cores may express claims considered equivalent for a particular downstream purpose. Such equivalence must not be silently built into the identity hash. It belongs in an explicit normalization or equivalence relation, because collapsing it into encode would contradict the injectivity required by the collision reduction

below.

Causal descent is a relation between envelopes with different identifiers, not a form of sameness. When an event is reopened, its descendant carries new content, a fresh occurrence token, a new identifier, and an explicit parent reference to the event it repairs. Chapter 8 treats this relation formally. A descendant is related to its parent, but it is not the same event.

Retries are arrivals that carry the same authenticated source token u, source s, content commitment h, and stable payload reference p. They therefore reproduce the same core and identifier even though their receipt times differ. This is intended behavior: it allows intake to recognize a resubmission as the same occurrence and deduplicate it instead of processing the request twice. Reuse of u with a different h or p is not a retry; it is a source-equivocation or malformed-submission event that must be recorded and refused.

Malicious replay looks identical to a retry at the event-core layer: an adversary resubmitting an authenticated core produces the same identifier a legitimate retry would. The two cannot be distinguished by identity comparison alone, and this chapter does not pretend otherwise. The danger concerns the reuse of downstream authorization. Chapter 13 addresses it through freshness, signature context, nonce uniqueness, and the current-head check at collapse.

5.4 Proof obligations and status

Load-bearing reduction. This section proves that a collision between distinct identity-bearing event cores reduces to a collision in the underlying hash function, under two named hypotheses, and states plainly what it has not proven.

Hypothesis (Inj). encode is injective on the space of well-typed identity cores:

$$c \neq c' \quad \Longrightarrow \quad \text{encode}(c) \neq \text{encode}(c').$$

Hypothesis (Dom). domain_e is a fixed bit string of known length, applied identically to every event-core encoding, and reserved exclusively for this purpose elsewhere in the architecture.

Proposition 5.1 (Event-identifier collision reduction)**.** *Suppose* $c(e) \neq c(e')$ *and* $id(e) = id(e')$*. Then* Hash *has a collision.*

Proof. By (Inj), $c(e) \neq c(e')$ gives $\mathrm{encode}(c(e)) \neq \mathrm{encode}(c(e'))$. Let

$$x = \mathsf{domain}_e \parallel \mathrm{encode}(c(e)), \qquad x' = \mathsf{domain}_e \parallel \mathrm{encode}(c(e')).$$

Because domain_e has fixed known length by (Dom), it can be stripped from either input uniquely. Thus $x = x'$ would force equality of the two core encodings, a contradiction. Hence $x \neq x'$. By definition, $id(e) = \mathsf{Hash}(x)$ and $id(e') = \mathsf{Hash}(x')$. The assumed identifier equality therefore gives $\mathsf{Hash}(x) = \mathsf{Hash}(x')$ with $x \neq x'$, which is a collision for Hash. □

5.4.1 What this does and does not establish

This is a reduction, not a proof of collision resistance. It does not show that Hash has no collisions; it shows that an identifier collision between distinct cores, given (Inj) and (Dom), cannot occur without one. Collision resistance remains an imported cryptographic assumption, established nowhere in this book and not attempted here [6].

The two hypotheses are equally load-bearing and neither is a cryptographic claim. Hypothesis (Inj) is a property of a serialization format, violated by ambiguous field boundaries, non-canonical numeric formatting, or unordered-map iteration. It applies to the identity core, not the full envelope: excluding t and m from identity is an explicit projection, not a failure of injectivity. Chapter 25's reference data model exists partly to specify a format where (Inj) is checkable and enforced. Hypothesis (Dom) is an engineering discipline maintained across the architecture: every identifier construction introduced later, for ledger records, commit objects, and collapse proposals alike, must

use its own distinct, reserved tag. Appendix D collects the complete registry of tags this requires.

A system satisfying (Inj) and (Dom) inherits exactly as much identifier collision resistance as its hash function provides, no more and no less. A system violating either hypothesis can produce identifier collisions with no hash weakness at all, and no stronger hash function will repair a non-canonical encoding or missing domain tag. Cryptographic strength and specification discipline are independent sources of assurance, and a reduction proof shows where their boundary falls.

Chapter 6

POP: Encounter Without Endorsement

6.1 Chapter thesis

Before an event can be refused, bound, transformed, verified, or collapsed, it must first be registered as something the system has encountered at all. POP is the transition that does this registration, and nothing more. It does not evaluate whether an encounter is welcome. It does not check the encounter against policy. It converts raw, arriving material into the well-typed envelope Chapter 5 defined, or determines that no such envelope can be constructed, and in either case it writes a durable record that the encounter happened. Everything this book calls admission—epistemic, computational, operational, or irreversible—happens strictly after POP, never during it.

6.2 Outputs and failure modes

The transition POP takes raw arriving material, call it e_{raw}, such as bytes on a wire, a message on a queue, or a submitted form, and

attempts to produce a well-typed envelope

$$e = \langle id, u, t, s, h, p, m \rangle$$

as defined in Chapter 5. Five distinct tasks make up this attempt.

6.2.1 Normalization

Normalization converts e_{raw} into the shape an envelope requires: extracting a claimed source and occurrence token, separating payload from descriptive metadata, and producing something canonically encodable. Normalization can fail outright, before any of the remaining tasks begin, if e_{raw} cannot be parsed into that shape at all.

6.2.2 Timestamping

Timestamping assigns t as the time of observation by the system itself, not a time claimed by whatever submitted e_{raw}. This detail is not incidental. A client-supplied timestamp is a claim the client makes about itself, and later chapters that reason about freshness and staleness need t to be a fact the system asserts about its own observation, not a fact it merely repeats on someone else's authority. For a recognized retry, the event retains the first-observation time in its registered envelope while the new arrival witness records the retry's own receipt time.

6.2.3 Source authentication

Source authentication verifies that s and its occurrence token u correspond to a principal or session the system can authenticate, rather than strings e_{raw} merely claims. An unauthenticated or unverifiable source produces intake failure prior to any question of whether that source is authorized to perform a requested operation. Authorization is a separate question addressed by later policy and binding stages.

6.2.4 Payload commitment

Payload commitment computes h from the payload's actual bytes and establishes p as a stable reference through which an authorized party can later retrieve them. This is where the separation argued for in Chapter 5, between content commitment and payload reference, is first put into practice: h is computed once, at intake, and everything built on top of the event refers to the commitment rather than incorporating the payload bytes directly.

6.2.5 Duplicate detection

Duplicate detection checks whether the authenticated identity core $c(e) = \langle u, s, h, p \rangle$ has already been registered. A genuine retry reproduces the same core and therefore the same $id(e)$, allowing the event registry to avoid creating a second independent event. The arrival itself is not erased or ignored: each retry still produces a new POP witness in the audit ledger, recording when it arrived and that it was deduplicated against the previously registered event. Reuse of u with a different payload commitment or reference is source equivocation rather than duplication and is recorded as such.

6.2.6 Intake failure

Intake failure is the outcome when normalization, timestamping, source authentication, or payload commitment cannot be completed at all, not because the content is unwelcome, but because no well-typed envelope can be constructed from what arrived. This differs in kind from the typed refusals Chapter 7 defines. Refusal presumes an envelope already exists and is being evaluated against policy. Intake failure means the system never got that far. The malformed network request of Chapter 1, missing a coherent content-length field, fails here, at POP, not at REFUSE, precisely because there is no well-formed e yet for a policy to evaluate.

6.3 The first ledger witness

Every one of these outcomes, successful registration or outright intake failure, must leave a durable trace, and the two cases require different minimum witnesses.

When normalization succeeds, the ledger records

$$\ell_t^{\mathrm{pop}} = \langle id(e), t, s, h, \mathit{dedup\_status} \rangle :$$

enough to establish that this identified event was encountered at this time, from this source, with this content commitment, and whether it was recognized as a retry of something already seen. An implementation may distinguish the event's first-observation time from the particular arrival time in an expanded record.

When intake fails, there is no $id(e)$ to record, because there is no well-typed e. The minimum sufficient witness commits to whatever raw bytes can safely be represented, records the arrival time and whatever source information could be determined even if it could not be authenticated, and states which preceding task failed and why. Sensitive or adversarial raw input need not be retained in plaintext; a commitment, bounded diagnostic sample, and protected payload reference may be the appropriate witness.

This is the floor beneath the entire architecture: even when a system cannot construct a single well-typed event from what it received, it must still be able to say later that something arrived, roughly when, and roughly what defeated the attempt to make sense of it. Without this floor, malformed input, garbled transmission, and deliberately unparseable payloads would leave no trace at all, precisely the kind of gap Chapter 22's threat model returns to when considering adversaries who benefit from a system's inability to say what it encountered.

6.4 Proof obligations and status

Load-bearing non-interference lemma. This section states and proves, for the first time in this book, the Early-Stage Non-Interference Lemma that Chapters 7, 9, and 10 instantiate for their transitions without repeating the proof. The canonical statement is registered in Appendix C; it is proved here because POP is the first transition defined formally, and the proof technique is easiest to see in the simplest case.

Lemma 6.1 (Early-Stage Non-Interference)**.** *Let τ be a transition whose complete defining semantics is given by a finite set of equations specifying its outputs and effects on machine state. Suppose H is either absent from those equations or appears only as a value read by τ, and never as a state component assigned by τ. Then for every t, applying τ at step t yields*

$$H_{t+1} = H_t.$$

Proof. The abstract machine's operational semantics, given fully in Appendix B, adopts the standard frame convention that every state component not assigned by a transition persists unchanged across that transition. By hypothesis, H is never assigned by τ. Applying the frame convention to H therefore gives $H_{t+1} = H_t$ directly. □

The proof is intentionally almost immediate. Its content is not in the derivation, which directly applies a convention fixed once in Appendix B, but in the discipline the lemma imposes on every transition definition that follows. Establishing the hypothesis requires an explicit check that H never appears on the written side of the transition's equations. This check is worth making precisely because it is easy to make silently false in an implementation: a shortcut that caches a computed value into the same store backing H, for example, violates the hypothesis even if the local code appears innocuous.

6.4.1 Instantiation for POP

The defining semantics of POP produce two kinds of result: a well-typed envelope together with a POP witness, or an intake failure together with a minimal salvage witness. In neither case does a transition equation assign a value to H. The lemma's hypothesis is satisfied by inspection, and it follows that

$$H_{t+1} = H_t$$

for every application of POP, successful or failed. The transition registers an encounter. It cannot, by the shape of its own definition, do anything more than that.

Chapter 7

REFUSE: Typed Non-Admission

7.1 The refusal transition

Once POP has produced a well-typed envelope, that envelope must still clear a second gate before anything is bound, computed, or committed on its behalf. The REFUSE gate evaluates the envelope against the policy version current at decision time and returns one of two typed branches:

$$\mathsf{REFUSE}_{\sigma_t}(e, H_t) \longrightarrow \begin{cases} \mathsf{ADMIT}(e, \alpha), \\ \mathsf{REFUSAL}(d), \end{cases}$$

where α is a screening certificate authorizing the envelope to proceed to BIND and

$$d = \langle r, \rho, \Delta, \tau, \pi \rangle$$

is the typed refusal decision. Within d, r is the disposition given to the event, ρ is a typed reason certificate, Δ states what a resubmission would need to supply, τ gives the conditions under which the matter can be reconsidered, and π is the evidence object supporting ρ.

This is a decision about admissibility, not a judgment about the

underlying claim's truth or worth. A refusal states that this envelope, evaluated against this policy and history at this decision time, does not clear the gate. It does not state that the claim it carries is false, harmful, or without merit. The admitted branch is equally bounded: it says that the event cleared screening, not that its later transformation or commitment is warranted.

For a refusal, the corresponding ledger record is

$$\ell_t^{\mathrm{ref}} = \langle id(e), h_{\mathrm{env}}(e), t_{\mathrm{obs}}, t_{\mathrm{dec}}, \sigma_t, r, \rho, \Delta, \tau, \\ h(\pi), \mathrm{parent}(e), \mathrm{sig}_G \rangle,$$

where $h_{\mathrm{env}}(e)$ commits to the complete registered envelope and is distinct from its payload commitment h. The record distinguishes observation time from decision time, commits to the evidence object rather than embedding it, tracks repair lineage through $\mathrm{parent}(e)$, and authenticates the deciding gate through sig_G. The admitted branch analogously writes ℓ_t^{admit}, committing to $id(e)$, $h_{\mathrm{env}}(e)$, σ_t, α, and the gate signature. Thus clearing the gate is witnessed no less than failing it.

7.2 Disposition algebra

The refusal disposition r is drawn from a closed set, and its six values are not interchangeable labels for the same act.

Quarantine holds an event pending resolution that is the system's responsibility, not the submitter's, most commonly while an internal investigation is underway. Nothing is asked of the submitter, and no repair path is offered, because the obstacle is not a defect in what they submitted.

Repair applies when the obstacle is a specific, correctable defect in the event itself, exactly the case Interlude I traced. The repair description Δ names what a descendant submission must supply.

Evidence-only retention preserves an event without pursuing a path toward admission and without rendering a judgment on its

eventual merit. This is the disposition suited to the scientifically unsettled claim of Chapter 1: retained because it may matter later, not admitted because nothing yet warrants that.

Deferral differs from evidence-only retention by attaching an explicit reconsideration condition in τ: a date, policy version, or external event. Evidence-only retention makes no such temporal commitment.

Redaction, as a refusal disposition specifically, applies when an event must be refused and its payload must additionally become unavailable, usually for privacy or legal reasons attached to the content. This is narrower than Chapter 19's general redaction mechanism, which may apply anywhere in a record's lifecycle. Here, payload availability is removed at refusal, before the event becomes a candidate for further processing.

Denial applies when evaluation is complete and final under current policy and evidence: not merely unready, as in deferral, but considered and rejected. A denied event remains fully present in the audit history. Denial changes what happens next; it does not erase that the claim was made and evaluated.

7.2.1 Why purge is not an ordinary disposition

No value of r authorizes deleting the fact that an event occurred. This follows from the append-only architecture of Chapter 4. Removing an encounter record would violate the completeness the witness plane exists to guarantee and erase the trail that lets a refusal be checked later. Even redaction, the closest disposition to what an ordinary system might call deletion, removes only payload availability. The event identifier, content commitment, refusal decision, and authorization for payload removal remain. What common usage calls purging is represented here as authorized payload redaction under a retained record, never removal of the authorization record itself.

7.3 Reason certificates

Six grounds populate ρ, and distinguishing them matters because each implies a different remedy, or the absence of one.

Identity grounds concern whether the authenticated principal is bound to the identity or role claimed for this particular class of event, a finer question than the bare session authentication POP performed.

Schema grounds concern whether content carried in a well-typed event envelope satisfies the more specific structural requirements of current policy, the missing transaction reference of Interlude I being the paradigm.

Provenance grounds concern whether the content's origin and chain of custody meet the required standard, independently of whether the envelope is well formed and its immediate source authenticated.

Policy grounds concern substantive prohibition: content that is well formed, properly sourced, and adequately provenanced but falls outside what current policy permits, such as an amount exceeding a hard cap.

Capability grounds concern possession of a particular delegated and potentially revocable authority. A principal can be exactly who it claims to be while lacking the capability token this event class requires.

Temporal grounds concern timing: arrival after a deadline, before a waiting period has elapsed, or outside another declared validity window.

7.3.1 Why $h(\pi)$ is a commitment, not a proof

The evidence object π can be large, sensitive, or both: a fraud report, excerpted logs, or an internal review record. The ledger commits only to $h(\pi)$ rather than embedding π. Within an authenticated, timestamped witness record, this secures a specific property: the evidence object used at decision time is fixed against silent later substitution. It does not establish that π actually supports ρ, and

hashing alone does not supply authorship or non-repudiation. Those depend on the gate signature, provenance, and declared evidence-evaluation procedure.

A committed evidence object and a sound reason certificate are separate achievements. The first is a cryptographic fixation claim; the second requires inspection of π against a declared standard for adequate support. Treating $h(\pi)$ as though it already certified ρ would repeat the error Chapter 3 warned against: mistaking a mechanism securing one property for a mechanism securing a stronger property it was never built to provide.

7.4 Refusal as information

A structured refusal certificate is more useful to a legitimate submitter than a bare rejection. Naming the missing field, violated cap, or expired window is what makes Δ actionable rather than a guess. Yet the same specificity can help an adversary map the system's defenses, learning which control caught an attempt and adjusting the next one to route around it.

This tension has no uniform resolution across all six grounds. Schema and temporal refusals are usually safe to disclose in detail. Identity, provenance, and capability refusals sit closer to active defense, and repeated attempts against these grounds are themselves evidence. Submitter-facing disclosure may therefore be coarsened to report only a security-relevant refusal while the full internal certificate remains available to authorized auditors.

What must not vary is the evidence commitment and full internal witness. The submitter-facing message and internally retained record may differ in granularity; the internal record must not be less complete than the actual basis of decision. Chapter 18's ledger projections generalize this principle: different audiences receive different views of one witness graph without requiring the underlying graph to be impoverished.

7.5 Proof obligations and status

Load-bearing non-interference lemma. This section instantiates the Early-Stage Non-Interference Lemma proved in Chapter 6. The generic lemma is not reproved; only its hypothesis is checked.

The REFUSE gate produces either an admitted result and ℓ_t^{admit}, or a refusal decision d and ℓ_t^{ref}. No admitted certificate and no disposition in the closed refusal set—quarantine, repair, evidence-only retention, deferral, redaction, or denial—is defined by an equation assigning a value to H. Every branch writes to L alone. The hypothesis of Early-Stage Non-Interference is therefore satisfied by inspection, and

$$H_{t+1} = H_t$$

holds for every application of the gate. Whether the event clears screening or receives a typed refusal, the decision extends audit history and leaves operational history exactly as it found it.

Chapter 8

Reopening Without Revisionism

8.1 Descendants rather than mutation

A refused event is never repaired by editing it. It is repaired by submitting a new event that names the refused one as its parent and supplies what the refusal's repair requirement requested:

$$e' = \mathrm{REOPEN}_{\nu_R}\big(id(e), \Delta', w, u'\big),$$

where Δ' is the claimed repair, w witnesses satisfaction of the reconsideration condition τ, u' is a fresh source-scoped occurrence token, and ν_R versions the reopening specification. That specification declares how Δ' combines with the original payload to construct the descendant's payload. The new event carries

$$\mathrm{parent}(e') = id(e).$$

Once registered, e' is an ordinary event under Chapter 5, subject to POP, REFUSE, and every downstream transition. Its only special structural feature is its authenticated parent reference.

One degenerate case deserves separation. If no content has changed

and the submitter seeks only a new decision under a later policy or history, creating a repair descendant would misdescribe what happened. The appropriate operation is a reevaluation request over the same event identifier, producing a new decision record under a new decision context rather than a new event. This book does not develop REEVALUATE in detail, but names it so that REOPEN is not mistaken for the only way a policy change can affect a prior refusal.

8.2 Historical truth

The refusal decision d made about e is a statement that the gate produced a particular disposition when evaluating e under policy σ_t, history H_t, and the evidence available at decision time. Nothing that happens to a descendant changes that historical fact. If e' is eventually admitted, verified, and collapsed, this does not by itself imply that the original refusal was mistaken. It may instead show that e' supplied something e lacked.

Append-only preservation must not be confused with correctness, however. A later audit may show that the original gate misapplied σ_t, relied on bad evidence, or generated an erroneous reason certificate. Such a finding is recorded as a later judgment about the earlier decision; it does not rewrite the earlier record. The architecture preserves what was decided, by whom, on what evidence, and under which policy, not a presumption that every preserved decision was justified.

This distinction has two consequences. First, accountability evaluates the decision against the policy, history, and evidence actually applicable at its decision time, rather than silently substituting later conditions. Second, it forecloses editing an old decision to make the record appear more consistent in hindsight. Chapter 20 generalizes this as the invariant against in-place revision of historical decisions.

8.3 Repair graphs

Treat every registered event identifier as a node and every authenticated parent reference as a directed edge from parent to child. The resulting structure, rooted at the original event, is a repair graph. It is often a tree in practice but need not be a chain: a submitter may attempt several repairs of the same parent, producing multiple children.

8.3.1 Bounded retry

Bounded retry is a policy layered over the graph rather than a structural property. Policy may cap how many descendants of a root can be reopened before mandatory escalation. This is the bounded-repair hypothesis used by Chapter 21's conditional liveness theorem; without it, the graph can grow indefinitely and no termination argument follows.

8.3.2 Acyclic repair

Each successful REOPEN operation must name an already registered parent and must introduce a fresh occurrence token u'. Registration assigns every new event a monotonically increasing creation rank $\mathrm{rank}(e)$. Therefore every parent edge satisfies

$$\mathrm{rank}(e) < \mathrm{rank}(e').$$

This temporal-order invariant, rather than an appeal to payload equality, forecloses cycles. Reuse of an existing occurrence token is handled as a retry or equivocation by POP and cannot create a new lineage node or edge.

8.3.3 Lineage queries

Given an event identifier, its ancestor chain is recovered by following parent references backward. Recovering all descendants requires an

index over audit records whose parent field names the identifier, because references point backward only. Both directions matter: an auditor asks how many attempts preceded success, while a reviewer asks what refusal a particular repair answered.

8.4 Proof obligations and status

Supporting lineage results. These propositions are not central safety theorems, but later audit and liveness arguments depend upon them.

Proposition 8.1 (Fresh Descendant)**.** *For* $e' = \mathrm{REOPEN}_{\nu_R}(id(e), \Delta', w, u')$*, where* u' *is fresh in the authenticated source domain,* $id(e') \neq id(e)$*, except with the collision probability inherited from* Hash*.*

Proof. The identity cores differ in at least the occurrence-token field, since u' is fresh. Under (Inj) and (Dom) from Chapter 5, equality of the resulting event identifiers would therefore induce a collision in Hash by the event-identifier collision reduction. □

Proposition 8.2 (Non-Mutation)**.** *No application of REOPEN alters or removes the refusal witness originally written for its parent.*

Proof. By premise (P1) of Chapter 4, existing ledger records are never overwritten. REOPEN constructs a new event with a fresh identifier and writes only records for that descendant. No defining equation reassigns a record keyed to the parent. The original refusal witness therefore persists unchanged, although a later audit may append a new record criticizing or superseding its judgment. □

Proposition 8.3 (Acyclicity of Repair Graphs)**.** *No repair graph whose parent references satisfy the registration-order invariant contains a directed cycle.*

Proof. Assume a directed cycle $e_0 \to e_1 \to \cdots \to e_k = e_0$. Every parent-to-child edge strictly increases creation rank, so

$$\mathrm{rank}(e_0) < \mathrm{rank}(e_1) < \cdots < \mathrm{rank}(e_k) = \mathrm{rank}(e_0),$$

which is impossible by irreflexivity of $<$. Hence no directed cycle exists. □

Chapter 9

BIND: Constraint as Contract

9.1 The binding transition

An admitted event does not proceed directly to computation. It first passes through BIND, which consumes the admission certificate α issued by the REFUSE gate and attaches the requirements against which a candidate output will eventually be judged. For fixed binding specification ν_B,

$$\mathsf{BIND}_{\nu_B}(e, \alpha, C, \Gamma_t) \longrightarrow \begin{cases} \mathsf{BOUND}(b), \\ \mathsf{DEFERRED}(d_B), \\ \mathsf{CONFLICT}(f_B), \end{cases}$$

where C is the finite constraint set drawn from governing policies, Γ_t is the binding environment describing which constraints apply and how they relate, and

$$b = \langle e_C, \alpha, \beta, \mathcal{O}, \nu_B \rangle.$$

Here e_C is the event paired with the constraints actually applied, β certifies which C, Γ_t, precedence rules, and policy versions produced

the binding, and $\mathcal{O}$ is the resulting obligation set. The outcomes d_B and f_B witness incomplete binding context and a detected constraint conflict respectively. They are outcomes of the total binding function, not obligation sets disguised as successful bindings.

The architectural point is timing. The obligations are fixed before TRANSFORM produces anything and before VERIFY inspects anything. Interlude I's three obligations were not invented after a candidate correction existed. They were fixed when the admitted request was bound, and the candidate was judged against those terms. BIND commits the system, in writing, to what will count as sufficient before it knows what computation will produce.

9.2 Obligation generation

Each member of $\mathcal{O}$ is a four-part record:

$$O_i = \langle P_i, m_i, w_i, \delta_i \rangle,$$

where P_i is the condition a candidate must satisfy, m_i is the declared verification method, w_i is the class of witness required to support a verdict, and δ_i is the discharge policy stating whether and under what authority the obligation may be waived rather than strictly satisfied.

All four components are fixed from $(e, \alpha, C, \Gamma_t, \nu_B)$ alone. None may be adjusted after a candidate exists. An obligation whose terms can shift after output inspection is not a contract; it is post-hoc rationalization written in contractual vocabulary.

9.3 Constraint conflicts

Four distinct problems can arise while turning constraints into obligations, and each requires defined handling rather than ad hoc judgment at binding time.

9.3.1 Unsatisfiable obligation sets

Policies written in isolation can generate requirements no candidate could jointly satisfy. BIND should catch syntactically obvious contradictions, such as disjoint numeric intervals or a predicate paired with its direct negation, and return $\mathsf{CONFLICT}(f_B)$ rather than manufacture a contract known to be hopeless. It cannot detect every unsatisfiable combination expressible in a sufficiently rich constraint language; Section 9.4 records that limit.

9.3.2 Policy precedence

Where policies impose differing but compatible requirements on one topic, Γ_t and ν_B declare a fixed order of precedence. The rule may select the more restrictive requirement or follow a jurisdictional hierarchy, but β must record which rule was applied so an auditor need not infer an unrecorded judgment.

9.3.3 Local versus inherited constraints

An event may carry locally declared constraints alongside constraints inherited from policy. The default is asymmetric: a local constraint may tighten the policy floor but may not loosen it. Otherwise a submitter could negotiate down terms already fixed by governing policy. Domains permitting a different merge must encode it explicitly in ν_B.

9.3.4 Binding under incomplete information

The environment Γ_t may lack context necessary for binding. BIND must not guess and proceed as if it were present. It either applies a conservative template that has been proven monotone under every permitted completion of the missing information, or returns $\mathsf{DEFERRED}(d_B)$. Merely calling a template "most restrictive" is insufficient when candidate constraints are incomparable; monotonicity must be a property of the binding specification.

9.4 Proof obligations and status

Load-bearing well-definedness result. The total result concerns the binding outcome; obligation determinism concerns its successful branch.

Proposition 9.1 (Binding-Outcome Well-Definedness)**.** *Fix ν_B. For every finite, well-typed (e, α, C, Γ_t), the mapping computed by BIND_{ν_B} returns exactly one element of*

$$\mathsf{BindingOutcome} = \mathsf{BOUND}(b) + \mathsf{DEFERRED}(d_B) + \mathsf{CONFLICT}(f_B),$$

and is deterministic. Conditional on the BOUND *branch, $\mathcal{O}$ is uniquely determined.*

Proof. Four properties of ν_B establish the result.

Canonical ordering. The specification fixes a total order on finite constraint contributions, independent of collection or transmission order.

Fixed resolution. It fixes a deterministic resolution function and a canonical fold order for every pair of contribution forms admitted by its domain. A detected syntactic contradiction maps to the conflict branch rather than an undefined state.

Deterministic obligation identifiers. Each resolved obligation is assigned

$$id(O_i) = \mathsf{Hash}\big(\mathsf{domain}_O \parallel \mathrm{encode}(O_i)\big),$$

under the (Inj) and (Dom) hypotheses of Chapter 5. Identical resolved obligations therefore receive identical identifiers.

Total handling of missing context. When necessary information is absent, the specification either selects a declared monotone conservative template or returns the deferral branch. Thus no well-typed input leaves the function stuck.

Canonical ordering and fixed resolution yield one resolved sequence. The identifier rule yields one identifier per resolved obligation. Conflict and missing-context cases map to distinct declared branches. Exactly

one binding outcome therefore results, and any successful obligation set is unique. □

9.4.1 Explicit non-claim

The proposition does not establish that a produced $\mathcal{O}$ is jointly satisfiable. General satisfiability for sufficiently expressive predicate languages, including languages capable of encoding unbounded arithmetic or arbitrary computation, is undecidable. BIND may and should decide declared fragments and catch syntactically evident contradictions. Beyond them, it has no general procedure for ruling out an impossible contract before TRANSFORM and VERIFY attempt to discharge it. Deterministic generation is not adequacy, correctness, or satisfiability.

9.4.2 Non-interference

The three BIND outcomes write binding, deferral, or conflict witnesses to L. None assigns H. By the Early-Stage Non-Interference Lemma of Chapter 6,

$$H_{t+1} = H_t$$

for every BIND application. Producing a contract, discovering a conflict, or deferring for missing context does not commit an operational transition.

Interlude II: Refusal Is Not Erasure

Chapters 7 and 8 gave refusal a formal shape: a typed disposition, a reason certificate, a repair path, and a lineage that survives whatever happens to a repaired descendant. None of that machinery is original to computing. Every institution that has had to decide what to accept and what to decline has faced the same problem, and the oldest of them solved it in the same general way this book does: by keeping a record of what was declined that can be every bit as durable as the record of what was accepted.

What archives already knew

An archive does not define its holdings by what it destroyed. A museum's list of declined acquisitions, a library's record of what it chose not to shelve, and a publisher's file of manuscripts it passed on are themselves historical documents, sometimes more revealing of an era's priorities and blind spots than the collection that resulted from them. Archival practice long ago settled the distinction this book calls epistemic admission against operational admission, without using those terms: something can be worth remembering precisely because it was not selected, since the act of declining it is itself a fact about the institution that declined it.

An archive that kept records only of what it accepted would not

merely be a smaller archive. It would be a different kind of thing, one unable to answer what it once considered, which standards it applied, which alternatives it excluded, or why it said no. The computational witness plane inherits this institutional insight while making its mechanics explicit. Refusal changes an event's admissibility; it does not retroactively prevent the encounter from having occurred.

Part III

Execution and Warrant

Chapter 10

TRANSFORM: Computation Without Commitment

10.1 The transformation boundary

Once an event has been successfully bound, a kernel is finally permitted to compute something from it. The transition is total over two branches:

$$\mathsf{TRANSFORM}_{K,\theta}(b, H_t) \longrightarrow \begin{cases} \mathsf{CANDIDATE}(x), \\ \mathsf{FAILURE}(f_K), \end{cases}$$

where f_K is a typed timeout, resource, sandbox, dependency, or execution failure. A successful candidate is

$$x = \langle y, \chi, T, \mathcal{C}, \nu_K, h(b), h(H_t) \rangle,$$

where K is the kernel, θ its configuration, y the candidate result, χ an execution witness, T the trace or a commitment to it, $\mathcal{C}$ the capabilities and resources consumed, and ν_K the kernel identity. The final two fields bind the candidate to the exact contract and read-only

operational context from which it was computed. Any reference to H_t is read-only; neither branch may write operational history.

The whole of Chapters 1 and 3 converges on this transition's most important property: it produces a candidate and nothing more. However correctly K executes, x has no standing beyond being one computation's output until VERIFY examines it against the obligations BIND fixed and COLLAPSE admits it. Correct execution and authorization remain separate questions.

10.2 Kernel identity

The kernel identity is

$$\nu_K = \langle h(K), h(\theta), h(E), h(D) \rangle$$

and commits separately to code, configuration, execution environment, and dependencies. The separate fields prevent the phrase "the same kernel" from hiding changes in parameters, runtime, libraries, hardware assumptions, data files, or model weights. Each hash commits to a canonical manifest or artifact; where a claim about the environment that actually executed is required, the manifest must be joined to an attestation rather than treated as self-proving.

These commitments are also precisely what replay requires. A version label or deployment tag is insufficient: replay must identify the exact code and the declared context under which it ran.

10.3 Capability confinement

A kernel must be bounded along several dimensions before its output is worth verifying.

Authorization determines which kernels and configurations may run against which classes of bound event. Availability of a general-purpose kernel does not license its use for every obligation class.

Resource bounds constrain execution time, memory, and other

consumed resources, producing a typed failure rather than indefinite execution or unbounded consumption.

Network access is denied by default. When necessary, it must be declared, and every response affecting computation must be captured or committed as an input. Even nominally read-only network calls may produce remote logging effects, so deployments requiring strict absence of external effects must use recorded substitutes rather than live services.

Clocks are a common hidden source of nondeterminism. Every clock read must be declared and its returned value captured. A client timestamp and a system observation time remain distinct inputs.

Randomness must be deterministically seeded from a recorded seed or treated as a declared nondeterministic channel whose actual outcome is captured for replay.

Side effects that mutate shared or external state are prohibited during TRANSFORM. Permitting them would let candidate production change the world before verification or collapse.

Other nondeterminism, including scheduler interleavings, unordered iteration, and floating-point reordering, must be eliminated or captured. An undeclared channel removes the execution from the scope of replay equivalence.

10.4 Execution witnesses

Different obligation classes warrant different witness strengths. The chosen style must be declared by ν_K so VERIFY knows what evidence to expect.

A **full trace** records every relevant operation, offering high fidelity at high storage and inspection cost. A **trace commitment** stores the trace elsewhere and retains a content commitment and protected reference. A **deterministic-replay witness** retains at least the complete input, context, version, and nondeterminism manifest required to reproduce execution, while omitting an operation-by-operation

trace. **Proof-carrying execution** supplies a machine-checkable proof about a declared computational property, practical only for kernels and claims amenable to formalization. An **attested environment** signs a statement that identified code ran within an identified protected platform, shifting trust to the attestation infrastructure and its trusted computing base.

No witness style is self-interpreting. Its verification relation, trust base, and non-claims must be declared alongside it.

10.5 Proof obligations and status

Load-bearing non-interference. Both CANDIDATE and FAILURE append an execution witness to L; neither assigns H. Early-Stage Non-Interference therefore gives

$$H_{t+1} = H_t$$

for every execution outcome.

Theorem 10.1 (Conditional Transformation Replay Equivalence)**.** *Assume: (a) K under θ is deterministic in its bound input, read-only history snapshot, initial state, and declared nondeterministic values; (b) those inputs are canonically encoded; (c) ν_K is identical across original and replay executions; (d) $h(b)$, $h(H_t)$, and the readable initial state are identical; (e) every clock value, random draw, network response, scheduling choice, and other nondeterministic channel is eliminated or replayed from its captured value; and (f) witness generation is itself deterministic modulo a declared trace-equivalence relation $\sim_T$. Then replaying TRANSFORM produces the same y and an execution witness equivalent under $\sim_T$.*

Proof. Under (a), the kernel is a mathematical function of the listed arguments. By (b)–(e), each argument supplied to the replay is identical to its original value. Function extensionality therefore yields the same y. Under (f), the witness mechanism receives the same

semantic execution inputs and may differ only in fields explicitly ignored by $\sim_T$, so the replayed witness is equivalent to the original under that relation. □

The hypotheses are substantive. If, for example, the kernel reads an undeclared clock whose value influences rounding, two executions with the same declared ν_K may produce different y. The theorem does not call such an execution deterministic merely because it contains no explicit random-number call. It excludes it because a semantic input was neither pinned nor replayed.

Chapter 11

VERIFY: The Discharge of Obligations

11.1 The verification transition

A candidate produced by TRANSFORM is checked against the obligations BIND fixed before it existed:

$$\mathsf{VERIFY}_{V,\sigma_v}(x, b, H_t) \longrightarrow v = \langle q, \mathbf{d}, \zeta, (\mathcal{R}_{\text{block}}, \mathcal{R}_{\text{waived}}), \nu_V \rangle,$$

where V is the verifier, σ_v its policy version, q the overall verdict, $\mathbf{d}$ the per-obligation discharge vector, ζ the verification witness, and ν_V the verifier identity. Like ν_K, ν_V commits separately to code, configuration, execution environment, and dependencies. A verifier is another program, not an epistemically privileged observer, and Chapter 3's cautions apply to it in full.

11.2 Four verdicts

A verification attempt terminates in exactly four reported ways.

PASS holds when every obligation whose policy status is REQUIRED has been demonstrated SAT under its declared method and witness

class.

FAIL holds when at least one required obligation is demonstrably UNSAT, rather than merely unproven.

INDETERMINATE holds when the verification procedure completed reliably but the available evidence leaves at least one required obligation unresolved, with no required obligation demonstrably violated.

ERROR holds when the verifying procedure did not complete as a valid run: a trusted wrapper detects a crash, timeout, resource exhaustion, dependency failure, malformed result, or comparable execution fault. This is a statement about the attempt, not about the true status of any obligation.

A binary pass/fail interface loses distinctions later stages need. Mapping indeterminacy to failure conceals the difference between refutation and missing evidence; mapping it to success admits the unproven. Mapping execution error to either can make an infrastructure failure look like a substantive verdict.

11.3 Discharge vectors

Each obligation is recorded on two independent axes:

$$
\begin{aligned}
&d_i = \langle s_i, a_i \rangle, \\
&s_i \in \{\mathsf{SAT}, \mathsf{UNSAT}, \mathsf{UNKNOWN}, \mathsf{NOT\_CHECKED}\}, \\
&a_i \in \{\mathsf{REQUIRED}, \mathsf{WAIVED}\}.
\end{aligned}
$$

The status s_i is evidential: it records what the declared method shows about P_i. The status a_i is normative and procedural: it records whether a shortfall is permitted, under the discharge policy δ_i fixed at binding, not to block commitment. A waiver changes operational eligibility; it cannot change UNSAT, UNKNOWN, or NOT_CHECKED into SAT.

Every waiver over a non-satisfied obligation cites the pre-encoded δ_i, the current policy version granting the invoking authority, and a wit-

ness that an authorized principal made this waiver for this obligation instance. VERIFY may validate and report such an authorization; it may not invent one after seeing the candidate.

NOT_CHECKED means the obligation was not established by a valid completed verification run. In an otherwise reliable run, this may mean its method was never executed. When the run ends in ERROR, normalization sets every s_i to NOT_CHECKED, even if partial internal artifacts suggest that some checks completed before the fault. Those artifacts may survive as quarantined diagnostics but are not reusable discharge results. By contrast, UNKNOWN means the declared method completed within a valid run and produced genuinely inconclusive evidence.

The discharge vector is total over $\mathcal{O}$: every obligation receives one entry. The residual sets are

$$\mathcal{R}_{\text{block}} = \{O_i \in \mathcal{O} \mid a_i = \mathsf{REQUIRED} \wedge s_i \neq \mathsf{SAT}\},$$

$$\mathcal{R}_{\text{waived}} = \{O_i \in \mathcal{O} \mid a_i = \mathsf{WAIVED} \wedge s_i \neq \mathsf{SAT}\}.$$

The first blocks collapse. The second does not block it, but remains a required disclosure containing the evidential deficit and its waiver authority.

11.4 What a verifier verifies

A verifier's report is recognized only if it satisfies a declared, independently checkable relation:

$$\mathsf{Check}_{\nu_V}\big(h(b), h(x), \mathcal{O}, \mathbf{d}, \zeta\big) = 1.$$

Given commitments to the binding and candidate, the obligation set, discharge vector, and witness, an authorized independent checker can determine whether the report conforms to ν_V's declared relation. This does not prove that ν_V embodies an adequate standard, only that the supplied witness supports the report under that standard.

A vector lacking evidence sufficient to evaluate this relation is an unsupported claim, not a verification result.

11.5 Proof obligations and status

11.5.1 Verdict computation

A trusted execution wrapper first determines whether a well-formed verifier run completed within its declared resource and dependency conditions. If not, $q = \mathsf{ERROR}$. Otherwise, if any required obligation is UNSAT, $q = \mathsf{FAIL}$. Otherwise, if every required obligation is SAT, $q = \mathsf{PASS}$. Otherwise, $q = \mathsf{INDETERMINATE}$.

Proposition 11.1 (Verdict Partition)**.** *Conditional on the wrapper returning exactly one completion classification, the four verdicts are jointly exhaustive and pairwise exclusive over every verification attempt.*

Proof. The decision procedure is a sequence of three ordered dichotomies: valid run or execution error; given a valid run, some required demonstrated violation or none; given none, complete required satisfaction or not. These cases exhaust the possible classified outcomes. Because evaluation stops at the first matching branch in a fixed order, no attempt reaches two branches. □

The proposition does not claim that every semantic fault inside V is detectable. A compromised verifier may return a well-formed but wrong result, and a compromised wrapper may misclassify the run. Those are trust and correlated-failure questions for Chapters 12 and 22, not contradictions of the partition over classified outcomes.

Proposition 11.2 (Fault Containment)**.**

$$q = \mathsf{ERROR} \quad \Longrightarrow \quad \forall i,\ s_i = \mathsf{NOT\_CHECKED}.$$

Proof. This follows from the output-normalization rule enforced by the trusted wrapper: on the error branch it constructs a fresh total

vector indexed by $\mathcal{O}$ and assigns NOT_CHECKED to every evidential component. It does not forward V's partial vector. Thus no errored run can export a reusable SAT status. □

This is a reporting and containment property, not the claim that an earlier partial check was logically false. It prevents later processes from salvaging apparently good fragments of a run whose overall integrity failed. Full re-verification from a clean attempt is required.

11.5.2 Non-interference

VERIFY writes its verdict, vectors, residual sets, and witness to L. It does not assign H. By Early-Stage Non-Interference,

$$H_{t+1} = H_t$$

for every verdict, including PASS. Passing verification makes a candidate eligible for the collapse guard; it does not itself commit the candidate.

Chapter 12

Verifier Independence and Correlated Failure

12.1 Chapter thesis

Placing verification in a separate stage from transformation is not, by itself, enough to make it independent. Independence is a property of what the two stages share, not of how many boxes separate them. If VERIFY parses output with the same faulty library TRANSFORM used to produce it, both may agree because they share the blind spot that made the result look correct. If the obligation set encodes a common misreading of an ambiguous requirement, separate implementations do not catch the gap. If one authority approves, deploys, or maintains both kernel and verifier, one compromise may place a matching flaw in both. This chapter names these shared failure channels because a stage that shares them is not doing the independent work its position in the pipeline appears to promise.

12.2 Diversity and quorum

Several practices strengthen verification only against particular failures.

Heterogeneous verifiers, built by separate teams with different languages and libraries, reduce common implementation faults. They do nothing against a specification error every team encoded identically.

Threshold acceptance is only as strong as the independence of the verifiers counted toward its quorum. A quorum sharing one critical dependency creates apparent redundancy without protection against that dependency.

Disagreement records matter independently of resolution. Divergent verdicts may reveal ambiguous obligations, model differences, or unequal evidence. Majority resolution should not erase that diagnostic fact.

Majority voting over correlated implementations can create positive miscalibration. Three matching reports are stronger than one only to the extent that their relevant failure modes are genuinely separate. Counting one shared interpretation or dependency three times does not make it independent.

12.3 Recursive verification

Requiring every verifier to be verified invites a regress: its verifier would itself require checking. This architecture truncates the regress explicitly at a named bounded trust root, such as attested hardware, a minimal reference implementation, a formally verified kernel, or a designated institution.

The truncation must be narrow and accompanied by an assurance statement. A hardware root may be trusted for a vendor-attested execution identity while providing no independent assurance of internal logic. A reference verifier may have a machine-checked soundness proof whose force still depends on the proof assistant and the specification formalized. The goal is not to eliminate the regress but to expose where internal verification ends and imported trust begins.

12.4 Proof obligations and status

Load-bearing negative result. Let dependency D, whether a library, specification, model, or authority, be shared by verifiers $V_1, \ldots, V_k$. Suppose D contains a fault for which there is a class of candidates X such that every verifier routing obligation O_i through D misreports s_i on every candidate in X. Let $p_X = \Pr[x \in X \mid D \text{ has the stated fault}]$.

Proposition 12.1 (Correlated Verifier Failure Lower Bound)**.** *Conditional on D having the stated fault,*

$$\Pr[V_1, \ldots, V_k \textit{ all misreport } s_i \mid D \textit{ faulty}] \geq p_X,$$

independently of k. If the fault is present with probability p_F and the candidate distribution conditional on its presence has mass p_X, then the unconditional joint-failure probability is at least $p_F p_X$, absent additional dependence information.

Proof. Conditional on the stated fault, the event $x \in X$ implies that all k verifiers misreport. It is therefore a subset of the joint-failure event, so the latter has probability at least p_X. The expression contains no factor depending on k. For the unconditional form, the intersection of fault presence and $x \in X$ implies joint failure; multiplying its conditional probability by p_F yields $p_F p_X$ under the stated conditional model. □

12.4.1 Why naive multiplication fails

If each verifier has observed marginal error rate ε, an independence model estimates unanimous error as ε^k. That calculation is valid only when the relevant error events are independent. If a substantial component of ε arises from D, the joint error is bounded below by the shared-fault term and need not shrink as verifiers are added. The arithmetic is not wrong; the independence model is.

12.4.2 Diversity that addresses the dependency

Independent reimplementation with genuinely separate libraries weakens library-level correlation. Independently derived formalizations of the informal requirement address specification-level correlation. Separate organizational authority and supply chains weaken common compromise channels. Diverse hardware and runtimes expose environment-specific faults. None of these substitutes automatically for another: hardware diversity does not repair identical source logic, and separate codebases do not repair a shared misreading of one specification.

Surface diversity does not suffice. Teams using different languages may still bind to the same load-bearing dependency, consult the same ambiguous source, or answer to the same authority. Assurance claims must therefore inventory shared dependencies and justify independence at the level of the failure being mitigated rather than infer it from organizational or syntactic appearances.

Chapter 13

Freshness, Substitution, and Replay

13.1 Candidate identity

A verification result is meaningful only if tied unambiguously to the exact candidate and binding it verified. Chapter 10 placed $h(b)$ inside the candidate as $x.h_b$, and the signed verification result commits to both $h(x)$ and that binding commitment. Define

$$\text{SameCandidate}(x, v) \iff h(x) = v.h_x \ \wedge \ x.h_b = v.h_b.$$

An adversary cannot pair a legitimate passing verification for x with a different x' unless it finds a distinct object with the committed hash or alters the authenticated verification result.

13.2 Freshness

Five freshness predicates apply at collapse, and each protects a different time-of-check to time-of-use boundary.

Elapsed-time freshness bounds the interval between verification and collapse. A verified market quote may become useless without

any change to the candidate's internal content.

State-head freshness requires the history head embedded in x and the collapse proposal to equal the actual current head at the atomic collapse operation.

Policy freshness requires the policy version used for binding, verification, and waiver authority to remain current or explicitly grandfathered.

Capability freshness requires every authority used by verification or collapse to remain unrevoked at collapse time.

Evidence freshness checks the separate validity windows of external evidence, which may expire before the verification result's general time limit.

The predicate $\mathrm{Fresh}(v)$ is the conjunction of the freshness requirements declared by the binding and collapse policy. Their reference versions, validity intervals, and registry identifiers must be committed into the authenticated verification and collapse records so the predicate can be re-evaluated rather than trusted as an opaque Boolean.

13.3 Attack catalogue

Certificate replay resubmits an already successful proposal. **Candidate substitution** pairs a passing result with an unverified candidate. **Stale authorization** relies on a revoked capability or superseded policy. **Rollback** presents an old history head as current. **Equivocation** presents different current heads to different parties, defeating the premise of freshness rather than its local equality check. **Time-of-check to time-of-use gaps** name the general interval in which state, policy, evidence, or authority can change after verification and before collapse.

13.4 Proof obligations and status

Load-bearing conditional security theorem. Let $\mathcal{A}$ be a probabilistic polynomial-time adversary able to observe L, delay, reorder, and replay messages, substitute candidates, and corrupt every principal except the trusted collapse arbiter, verifier-signing authority, freshness registries, and clock named below.

Assume: (i) Hash is collision-resistant; (ii) signatures authenticating v and c are existentially unforgeable under chosen-message attack; (iii) spent-nonce tracking is linearizable and rejects every reuse of η; (iv) the current-head comparison is an atomic compare-and-append performed by an uncorrupted arbiter against the true head; and (v) elapsed-time, policy, capability, and evidence freshness are evaluated fail-closed against an authenticated consistency snapshot of the required registries and a trusted clock, with all checked versions and validity windows committed into the signed proposal.

Theorem 13.1 (Freshness and Substitution Resistance)**.** *Under (i)–(v), the probability that $\mathcal{A}$ causes the collapse guard to accept a replayed, substituted, or stale candidate is negligible in the security parameter, except for the explicitly assumed failure probabilities of the trusted arbiter, registries, clock, and nonce store.*

Proof. Suppose $\mathcal{A}$ succeeds with non-negligible probability. Since the guard is a conjunction, success must evade at least one relevant check.

Candidate substitution. Presenting $x' \neq x$ while satisfying $h(x') = v.h_x = h(x)$ yields a hash collision, unless the adversary changes the committed value in v, which requires a signature forgery. The same argument applies to substituting a different binding commitment.

Certificate replay. Reusing a successful proposal reuses its nonce. By (iii), the atomic $\neg\,\mathrm{Spent}(\eta)$ check rejects it. A random nonce collision is also rejected; nonce-space size concerns availability and accidental rejection, not acceptance of a spent proposal. Altering η inside the authenticated proposal requires a signature forgery.

State-head staleness or rollback. By (iv), a proposal bound to an

old head fails the atomic current-head comparison. Presenting two distinct histories with one head commitment requires a collision in Hash.

Other freshness failures. By (v), expired evidence, elapsed time, revoked capability, or obsolete ungrandfathered policy causes Fresh(v) to fail. Changing a committed version or validity window requires a signature forgery or hash collision. Acceptance despite an authentic stale registry value requires failure of the trusted freshness infrastructure, which the theorem records as an external assumption rather than bounding cryptographically.

A union bound over collision, forgery, and the explicitly imported trusted component failures yields negligible adversarial success under the stated hypotheses. □

13.4.1 What the theorem does not cover

The atomic, non-equivocating arbiter and authenticated freshness infrastructure are indispensable. A dishonest arbiter may present different heads to different parties without finding a collision, forging a signature, or reusing a nonce. Likewise, a compromised capability registry can report a revoked authority as current while every local cryptographic check succeeds. Chapter 16 names the linearizability assumption required to prevent forks; Chapter 22 places these trusted components inside the threat model. Stronger hashing cannot repair a false source of current state.

Chapter 14

Verification Is Not Truth

14.1 Chapter thesis

A verifier can execute correctly, suffer no dependency failure, avoid every correlated-implementation problem Chapter 12 names, satisfy the hypotheses of Chapter 13's security theorem, and still certify an obligation set that fails to capture what mattered. This is not the claim that verification is unreliable. Even a maximally reliable verifier cannot exceed what it was told to check. Chapter 3 promised to return to this point once the formal apparatus could state it narrowly. With that apparatus now available, the limitation is almost definitional.

14.2 Specification horizons

Three questions sit at increasing distance from what VERIFY answers.

Internal validity asks whether the procedure correctly assessed the declared predicates using their methods and witness classes. Chapters 11–13 specify the conditional assurance available here, including verdict semantics, correlated-failure limits, candidate identity, and freshness.

Operational adequacy asks whether $\mathcal{O}$, taken as a whole, captures what the domain requires for commitment. This concerns

the content of the obligation set supplied by BIND. VERIFY takes that set as an input and has no semantic rule allowing it to infer completeness relative to an unstated domain requirement.

World-relative success asks whether acting on a candidate satisfying an adequate obligation set produces the desired real-world outcome. Even a well-specified and correctly verified contract can fail here because the model omitted a relevant variable or the world contains an unknown condition.

Consider an autonomous braking system whose obligations require stopping distance to remain within a bound for a declared friction coefficient and speed. The verifier executes flawlessly and the trial evidence supports $q = \mathsf{PASS}$. Internal validity holds. Suppose, however, that the binding has no predicate for a sharp localized change in friction during the stop. VERIFY did not fail; it answered the question it received. The obligation set omitted the question that later mattered, and execution reliability cannot manufacture a predicate that BIND never supplied.

14.3 Revisable assurance

An incident, near miss, or new research can revise confidence in the adequacy of the earlier obligation set. The domain community may add the missing predicate so later bindings do not repeat the gap. This does not rewrite the historical record that the earlier verifier produced PASS for the narrower set under the policy then in effect. Under this chapter's premise of a correct verification run, the report was accurate about what it checked. What hindsight changes is the judgment of whether that was the right thing to check.

This is the historical-preservation principle of Chapter 8, not a rule that old judgments must forever be considered correct. If later evidence shows the earlier run itself was faulty, that finding is appended as a new judgment about the preserved result rather than used to edit the original record.

14.4 Proof obligations and status

Documented non-claim with a definitional proposition.

Proposition 14.1 (What PASS Certifies)**.** *By the semantics of* PASS *alone,* $q = \mathsf{PASS}$ *for* (b, x, v) *certifies that every obligation whose policy status is* REQUIRED *received* $s_i = \mathsf{SAT}$ *under its declared method* m_i*, supported by evidence of the declared witness class* w_i*, as reported in* $\mathbf{d}$ *and checkable through* Check_{ν_V}*. It assigns no additional architectural meaning concerning the completeness or adequacy of* $\mathcal{O}$*, or the real-world consequences of collapse.*

Proof. Chapter 11 defines $q = \mathsf{PASS}$ as the reliable-run branch in which every required obligation is SAT. The definition of Check_{ν_V} ranges over the committed binding, candidate, obligations, discharge vector, and witness. Neither definition quantifies over unstated domain requirements, the fidelity of an informal-to-formal translation, or downstream consequences. Therefore no such claim belongs to the semantics of PASS itself. Stronger conclusions may follow when separate domain premises are supplied, but their additional force comes from those premises rather than from VERIFY alone. □

14.4.1 Why this is not a Gödelian result

Gödelian incompleteness concerns what sufficiently expressive formal systems can prove in settings involving internalized syntax and self-reference. The present limitation requires none of that machinery. It is a scope fact about an interface: VERIFY evaluates the obligations supplied to it. The braking counterexample shows what lies outside that declared input. Invoking incompleteness would dress an ordinary semantic boundary in borrowed authority while obscuring the more useful question of who constructed $\mathcal{O}$, from which model, and under what process of revision.

Interlude III: The Proof That Proved Too Little

Chapter 3 mentioned a binding that requires a bridge to tolerate a given load and never mentions resonance. Chapter 14 gave that sentence its precise content: a verifier can execute flawlessly and still certify an obligation set that never asked the right question. This interlude gives the sentence a full life, tracing one certification through the pipeline without a refusal or failed verification, and showing what that cleanliness did and did not guarantee.

The submission

A structural engineering firm submits a certification request for a new pedestrian bridge: lightweight, materially efficient, and intended for a river crossing that has needed one for years. The request is well formed, arrives from an authenticated and appropriately credentialed source, and POP registers it without incident.

Admission

REFUSE finds nothing to object to. Schema and provenance are complete, the firm holds the required capability, and timing raises no temporal concern. The event is admitted on its first attempt, with no repair descendant.

The obligations

BIND draws from the certification code then in force. Static load capacity must meet the design specification for foot traffic, material fatigue ratings must satisfy the intended service life, and the overall safety factor must clear the regulatory minimum. These are serious obligations. Their ordering is canonical, precedence is unambiguous, and a second application of the same binding specification produces the same set.

Computation and verification

The kernel performs load-distribution calculations, material-stress modelling, and fatigue projection. The candidate exceeds every declared threshold. VERIFY checks each obligation. Static load, fatigue rating, and safety factor are all SAT; $q = \mathsf{PASS}$, with neither blocking nor waived residuals. A reviewer inspecting exactly these obligations has nothing in the discharge vector to flag.

Commitment

The collapse guard finds the actor authorized, the candidate collapsible, the head current, the nonce unspent, and the certification mechanism ready. The certification becomes operationally authoritative. Construction proceeds and the bridge opens.

What no obligation asked

Months later, during a crowded crossing, the bridge begins to sway. The cause is not static overload but pedestrian footsteps coupling with a low-damping lateral mode. Pedestrians adjust their gait for balance, strengthening the synchronization. The bridge closes for inspection; no one is seriously hurt, but retrofit is lengthy and costly.

The audit that found no pipeline violation

Review recovers the intake witness, admission certificate, binding and its three obligations, execution witness, clean discharge vector, and collapse record. Obligation generation was deterministic. Computation and verification were correct under their declared models. Replay reproduces the same result. Nothing was skipped or falsified.

The obligation set did not contain a predicate for pedestrian-synchronized lateral response in a low-damping design. The phenomenon was known in the broader literature but absent from the jurisdiction's adopted code. Whether that absence should be attributed to negligence, institutional lag, or a reasonable limitation of knowledge is a separate governance inquiry. At the pipeline level, the defect was a specification gap rather than a failure to discharge the specification supplied.

What changes and what does not

The code is revised to require explicit pedestrian-synchronized lateral analysis below a damping threshold. The revision does not edit the earlier record. That record continues to show a design satisfying every obligation its binding generated, verified without procedural error and committed under a guard that found no declared defect. The bridge still swayed. Both facts remain visible, allowing later reviewers to distinguish a failure of execution from a failure in what the institution had learned to ask.

Part IV

Commitment and Effect

Chapter 15

COLLAPSE: The Commitment Boundary

15.1 The collapse proposal

Everything preceding this chapter exists to be checked at one final boundary. A collapse proposal gathers what must be true for a verified candidate to enter operational history:

$$c = \langle x, v, h(H_t), \sigma_c, \eta \rangle.$$

Its identifier is derived without self-reference:

$$id_c = \mathsf{Hash}\big(\mathsf{domain}_c \parallel \mathrm{encode}(c)\big).$$

The commit object is then

$$\kappa = \langle id_c, h(H_t), h(x), h(v), \sigma_c, \eta, t_c, a_c \rangle.$$

It records the proposal, prior head, candidate and verification commitments, policy, nonce, commit time, and committing authority. The new head is derived only afterward:

$$h(H_{t+1}) = \mathsf{Hash}(\mathsf{domain}_H \parallel h(H_t) \parallel \mathrm{encode}(\kappa)),$$

and belongs to the history header or append result, not to κ itself. This avoids a circular pre-image in which the commit object contains the hash whose computation already depends upon that object.

The compact operational object κ commits to v through $h(v)$. Its corresponding audit witness carries the fuller disclosure record required for inspection, including any waived deficits. Operational history need not duplicate every audit field in order to commit cryptographically to it.

15.2 The collapse guard

A proposal succeeds only if five independent checks hold:

$$\begin{aligned} G(c, H_t) = \ & \mathrm{Authorized}(c) \wedge \mathrm{Collapsible}(x, v) \\ & \wedge (h(H_t) = c.h_H) \wedge \neg\, \mathrm{Spent}(\eta) \\ & \wedge \mathrm{EffectsReady}(x). \end{aligned}$$

Authorization concerns the committing actor, not merely the original submitter. Head agreement enforces state freshness atomically with append. Nonce uniqueness rejects replay. Effect readiness determines whether the internal posting mechanism or Chapter 17's prepared external-effect mechanism can proceed. Collapsibility is defined against the blocking residual set:

$$\begin{aligned} \mathrm{Collapsible}(x, v) \iff & \ q = \mathsf{PASS} \wedge \mathcal{R}_{\mathrm{block}} = \varnothing \\ & \wedge \mathrm{Fresh}(v) \wedge \mathrm{SameCandidate}(x, v). \end{aligned}$$

15.3 Principal proposition

Proposition 15.1 (Collapse Guard Respects Waivers)**.** *Two properties hold:*

(a) *collapse eligibility depends on* $\mathcal{R}_{\mathrm{block}}$*, not on whether* $\mathcal{R}_{\mathrm{waived}}$ *is empty; and*

(b) *every waived deficit is explicitly disclosed in the collapse audit*

witness and cryptographically committed by κ through $h(v)$.

Proof. For (a), the definition

$$\text{Collapsible}(x, v) \iff q = \mathsf{PASS} \wedge \mathcal{R}_{\text{block}} = \varnothing \wedge \text{Fresh}(v) \wedge \text{SameCandidate}(x, v),$$

contains no emptiness test for $\mathcal{R}_{\text{waived}}$. Holding the displayed conjuncts fixed therefore fixes collapsibility regardless of the waived set.

For (b), the successful-collapse rule constructs $\ell_\kappa^{\text{collapse}}$ by copying, for every $O_i \in \mathcal{R}_{\text{waived}}$, the obligation identifier, unchanged s_i, authorizing δ_i, waiver-policy version, authority, and witness from v. No transition rule rewrites these fields. The operational object stores $h(v)$, so alteration of the verification result or disclosed waiver data breaks that commitment, subject to hash collision resistance. Thus waiver changes eligibility without laundering a deficit into SAT, and the deficit remains inspectable in the witness plane. □

15.4 Typed collapse failure

A false guard produces a typed failure rather than a generic rejection.

Stale head is detected during preflight when the proposal's base head already differs from the current head. Remedy may require reverification or rebinding, depending on what the intervening change affected.

Replay means η is already spent. Both malicious replay and an innocent duplicate proposal are rejected. A new attempt requires a newly authorized proposal with a fresh nonce.

Unauthorized means the committing actor lacks current standing under σ_c.

Verification invalid covers failure of a collapsibility conjunct: non-PASS verdict, blocking residual, stale verification, or candidate mismatch. The audit record retains the failed sub-conjunct rather than flattening these reasons into one opaque code.

Effect unavailable means the declared realization mechanism is not ready. This failure may be transient, but retry remains subject to freshness and nonce rules.

Commit conflict means preflight observed a matching head, but another proposal won the subsequent atomic compare-and-append. This is a genuine race lost at the serialization point, distinct diagnostically from a proposal known to be stale before attempting the atomic operation.

Every failure appends $\ell^{\text{collapse\_failure}}$ to L and leaves H unchanged:

$$H_{t+1} = H_t, \qquad L_{t+1} = L_t \parallel \ell^{\text{collapse\_failure}}.$$

Only the successful atomic branch appends κ to H and its matching collapse witness to L.

Chapter 16

Concurrency and the Moving Head

16.1 Compare and append

The collapse guard's head-agreement check, $h(H_t) = c.h_H$, is not a convenience check performed before the real work of appending happens. It is the same primitive that underlies every lock-free concurrent data structure: compare a shared value against an expected value, and perform the update only if they match, as a single, indivisible operation. If the comparison and the append were instead two separate steps—check the head, then append—a window would open between them during which another collapse could change the head. The original attempt could then append on top of a head that was no longer current by the time it acted. This is the time-of-check to time-of-use gap from Chapter 13 applied to the moment operational history advances. Atomicity is not a performance optimization. It is what makes the head-agreement check mean what it claims to mean.

16.2 Competing valid candidates

Two candidates can each be fully and correctly verified while being jointly inconsistent if both are committed. Consider two withdrawal requests against the same account, submitted close together. Each is individually verified against the balance as it stood when its verification ran, and each satisfies every obligation in its own obligation set. If both could be committed, their combined value might exceed the balance even though neither verification, taken alone, could see the other candidate racing toward commitment.

The head-based atomic compare-and-append solves this problem not by making verification smarter, but by serializing commitment. Whichever withdrawal reaches the atomic append first changes the head. The second attempt, built against the old head, then fails head agreement and must be reconsidered against the new state. Under that state, its balance obligation may now fail, correctly and for exactly the reason joint consistency requires: the first commitment spent value on which the second candidate depended.

16.3 Rebinding after conflict

A head conflict establishes that some operational fact changed; it does not by itself establish how far backward through the pipeline a candidate must go. The remedy depends on which earlier artifact the intervening commit may have invalidated.

If the new head is proven irrelevant to both the derivation of y and the evaluation of every obligation, the existing candidate may be reverified against the new head. This is the narrowest case: the history moved, but no state read by TRANSFORM and no fact used by VERIFY changed.

If the intervening commit changed state read by TRANSFORM, while the binding and obligation set remain current, reverification of the old candidate is not enough. The candidate itself may be stale. The system must rerun TRANSFORM against the new state and

then VERIFY the newly produced candidate.

If the intervening change altered the governing policy, the binding specification ν_B, or any constraint context from which $\mathcal{O}$ was derived, full rebinding is required. The system returns to BIND, generates a new obligation set under the current specification, and carries the resulting binding through TRANSFORM and VERIFY again. Reverifying against a stale contract would silently hold a candidate to requirements the system no longer considers authoritative.

16.4 Distributed ordering

A single sequencer, one designated arbiter processing collapse attempts in a strict order, provides linearizability directly. It is the simplest design to reason about and the clearest single point of failure.

Consensus protocols let multiple replicas agree on one commit order while tolerating some replica failures. They can provide the same abstract linearizability guarantee as a sequencer, at the cost of latency and protocol complexity. Paxos and Raft are representative constructions [3, 4].

Causal partial orders relax the demand for one global total order. Commits without causal dependence may proceed without being ordered against one another. This changes what "history never forks" means: history becomes a directed graph rather than a single chain. The theorem below assumes a total order and does not transfer automatically to this setting.

Sharded histories give separate state partitions independent local heads and arbiters. Commits confined to different shards need not coordinate. A collapse that must update several shards atomically, however, requires machinery beyond the theorem in this chapter.

Domain-local collapse occupies a middle position. Each domain maintains its own history and arbiter, while explicitly declared cross-domain obligations trigger synchronization. The common case remains local; cross-domain consistency becomes a visible and deliber-

ately more expensive exception.

16.5 Principal theorem

Theorem (History Never Forks). Assume:

(H1) H_0 is a unique, well-defined chain, whether the empty history or a single agreed genesis state.

(H2) Every application of COLLAPSE performs its head comparison and append as one linearizable atomic compare-and-append operation. This operation is enforced either by a single arbiter processing attempts serially or by a consensus mechanism providing the equivalent guarantee that every observer sees operations as if they occurred in one agreed total sequence.

Then, for every t, H_t remains a unique chain, and at most one candidate referencing a given $h(H_t)$ as its expected head succeeds in extending it.

Proof. Proceed by induction on t. The base case holds by (H1). For the inductive step, assume H_t is a unique chain, with one agreed value of $h(H_t)$. Consider all collapse attempts submitted with that expected head. By (H2), these attempts take effect as if processed in one agreed serial order. Failed attempts leave the head unchanged. The first attempt in that order that satisfies every conjunct of the guard succeeds and atomically replaces the shared head with $h(H_t \,\|\, \kappa)$. No later attempt expecting the old head can then succeed, because its head-agreement conjunct is false at its own linearization point. Thus at most one candidate referencing $h(H_t)$ extends the history.

If no attempt succeeds, $H_{t+1} = H_t$. If one succeeds, $H_{t+1} = H_t \,\|\, \kappa$ for the unique successful commit object. Either result is one well-defined chain agreed by every observer under (H2), so the induction closes. ■

What the theorem does not get for free. Neither Chapter 5's identity-core construction nor Chapter 13's freshness and substitution guards supplies (H2). Identity protects object distinction; freshness

protects a correctly functioning head comparison. Neither prevents two non-linearizable replicas from independently accepting different commits against what each believes is the current head. In that case, two internally valid chains can diverge without any hash collision, signature forgery, or nonce reuse. This is the equivocation attack Chapter 13 deferred. Preventing it requires an actual single sequencer or a correctly implemented consensus or compare-and-swap service. It is an engineered infrastructure premise, not a consequence of cryptographic hygiene elsewhere in the architecture.

Chapter 17

Irreversible Effects

17.1 The external-world problem

A ledger append is entirely within a system's own control. It touches storage the system manages, and Chapter 4 established the conditions under which the append and its audit witness can be published atomically. A physical actuation, a wire transfer through an external network, or a shipment dispatched through a third party is a different kind of operation. It reaches infrastructure this architecture does not control, through a network or physical link that can fail, delay, or duplicate messages. There is generally no way to make "record this locally" and "cause this to happen out there" one indivisible operation because they do not occur within one transactional boundary.

Suppose a system calls an external wire-transfer API and the connection times out before a response arrives. The request may never have reached the bank, or the bank may have completed the transfer while only its response was lost. Those possibilities are indistinguishable to the caller from the timeout alone. This ambiguity is not cured by better exception handling. The protocol below is designed to survive it without pretending it away.

17.2 Prepare, actuate, finalize

Three steps replace the impossible atomic operation:

$$\text{PREPARE}(c) \to \iota, \qquad \text{ACTUATE}(\iota) \to \epsilon, \qquad \text{FINALIZE}(\iota, \epsilon) \to \kappa.$$

PREPARE records a specific intent ι durably before any request is sent outside the system. The intent includes the collapse proposal, the external operation to be requested, and an idempotency key derived from its committed fields. A process resuming after failure can therefore discover that an actuation may be pending without reconstructing intent from volatile state.

ACTUATE first writes a durable attempt witness keyed to $id(\iota)$, then uses ι's stored idempotency key in the external request. If a response arrives, it returns a receipt ϵ, which records either the external system's attestation of success or a typed failure. The attempt witness is written before transmission because a crash after transmission but before local bookkeeping would otherwise make a genuine retry look like a first attempt. The receipt may still be lost even when the external effect succeeds.

FINALIZE combines ι and a conclusive ϵ, obtained either from the original response or from later reconciliation, into the commit object κ appended to operational history. An inconclusive transport failure is not a receipt of non-occurrence and cannot authorize finalization.

Between successful PREPARE and FINALIZE lies an inherent interval in which the effect may already have occurred while H reflects neither success nor failure. The prepared intent belongs to the machine's durable pending-intent state P_t, introduced in Appendix B, rather than to H_t. If a deployment treats P_t as authoritative operational state, it instantiates Chapter 2's $\text{Op} \wedge \neg\text{Ir}$ case. If it reserves Op strictly for H, PREPARE is instead a durable procedural commitment below operational admission. Conversely, successful ACTUATE followed by a crash before FINALIZE produces the orphaned-effect case $\text{Ir} \wedge \neg\text{Op}$.

17.3 Recovery semantics

Exactly-once external execution cannot be guaranteed from this side of the boundary. The defensible target is durable intent, at-least-once delivery where appropriate, idempotent processing where the external system supports it, and explicit reconciliation wherever ambiguity remains.

Durable intent is ι, retained before actuation. Every actuation attempt uses the same stored idempotency key and produces a local attempt witness. Receipts are retained and referenced by FINALIZE whether they arrive promptly or through a later status query. Reconciliation examines a stalled intent, queries the external system using that key, and either finalizes an effect shown to have occurred, records a conclusive failure, or retries under the external system's declared duplicate-handling contract.

When an effect occurred but later must be opposed, the remedy is a compensating action: a refund, reversal, corrective shipment, or other new event. The original consequence cannot be deleted from the world. Compensation therefore resembles Chapter 8's descendant repair: it adds a causally related act rather than rewriting the act that preceded it.

17.4 Proof obligations and status

External assumption, not an internal theorem.

Proposition (Intent Determinism and Local Duplicate Detectability). Let $k(\iota)$ be the idempotency key stored by PREPARE, and let $A(\iota)$ be the durable sequence of actuation-attempt witnesses for the intent. Then: (a) every ACTUATE invocation for the same ι presents the same key; and (b), assuming the pending-intent and attempt stores are durable, an invocation is locally identifiable as a retry exactly when $A(\iota) \neq \varnothing$ before its new attempt witness is appended.

Proof. PREPARE computes $k(\iota)$ once as a pure function of the

intent's committed fields and stores it inside the immutable intent. ACTUATE reads this value rather than generating a new one, proving (a). By the transition's required order, the first invocation finds no prior attempt witness and durably appends one before transmission. Every later invocation finds at least that witness, even if the first transmission or its response was interrupted by a crash. Under the durability premise, the emptiness test therefore distinguishes a first local attempt from a retry, proving (b). ■

The attempt journal is necessary. The mere presence of ι proves that PREPARE occurred, not that ACTUATE was previously attempted; an intent can remain prepared for an arbitrary period before its first transmission.

Neither part proves what an external system does when it receives the same key twice. ACTUATE-idempotency is a capability contract of that integration, not a theorem of Spherepop. Where the contract holds, the external system absorbs duplicate requests. Where it does not, Spherepop can prove only that its own duplicate attempts are visible and available to reconciliation. Even this weaker guarantee matters: an unpreventable duplicate need not also be a silent one, and a detected duplicate can trigger a separately recorded compensating action.

Chapter 18

The Ledger as a Witness Structure

18.1 Ledger records

Every audit record accumulated across the preceding chapters shares one common shape:

$$\ell_i = \langle seq_i, kind_i, subject_i, h(payload_i), policy_i, \\ parents_i, t_i, actor_i, sig_i, h(\ell_{i-1}) \rangle.$$

The sequence number seq_i is assigned by the ledger and fixes append order independently of any claimed wall-clock time. The typed field $kind_i$ distinguishes pop, refusal, reopening, binding, transformation, verification, collapse, collapse failure, preparation, actuation attempt, reconciliation, and related witnesses. It makes Chapter 4's typed-record premise mechanically checkable. The subject identifies the event, binding, candidate, intent, or commit object concerned. The payload hash commits to detailed content while permitting that content to be stored under a separate access regime.

The remaining fields name the governing policy, causal parents, timestamp, actor, and authenticating signature. Finally, $h(\ell_{i-1})$ chains the record to its predecessor. Altering a committed field

changes the record hash, which invalidates the successor's predecessor commitment and propagates toward the anchored head under the assumptions stated below.

18.2 Tamper evidence and its limits

Tamper evidence does not establish correctness. A dishonest actor can write a false record initially and leave it untouched thereafter; the chain preserves that falsehood perfectly. It protects integrity after recording, not truth at recording time.

Nor does it establish availability. A well-chained record may be lost, destroyed, inaccessible, or deliberately withheld. Replication, backup, and access governance remain separate requirements.

Hash chaining also does not establish non-equivocation. A ledger operator can present different observers with different chains, each internally consistent. Chapter 16's linearizability premise governs H; it does not automatically govern L. If the two histories use different publication mechanisms, the audit ledger can equivocate even while operational history remains unique. Closing that gap requires an authoritative, consistently published ledger head or an equivalent consensus or transparency mechanism.

Finally, the chain does not make t_i honest. A corrupted or skewed clock can supply a false timestamp that the chain then preserves faithfully. Independent timestamping, external anchoring, or attested clocks are needed when temporal truth is itself load-bearing.

18.3 Ledger projections

Different audiences require different views of the same witness graph. An audit view exposes structural fields and the signature chain while respecting payload access restrictions. An operational view selects collapse witnesses. A provenance view follows parent relations for one subject. A policy view selects records governed by a particular

policy version. A security view selects relevant refusal, authentication, capability, and failure records. A privacy view withholds unauthorized payload references while retaining the structural fact that a record exists.

These are derived projections, not independently maintained histories. A faithful projection must satisfy two conditions relative to its declared filter: it fabricates no source record, and it omits no source record that the filter requires. Selective disclosure may hide fields or payloads, but any claim of completeness for a projection requires either access to the source graph or a cryptographic inclusion-and-completeness proof supplied by the implementation. The hash chain alone supplies neither.

18.4 Principal theorems

Load-bearing ledger completeness. Chapter 4 proved that every $\kappa \in H_t$ has exactly one collapse witness in L_t under atomic dual publication. Real deployments may store H and L separately, so this chapter admits a weaker implementation premise.

(P3') Recoverable write-ahead publication. Before either visible history is extended, a single durable transaction record w_κ commits to both κ and $\ell_\kappa^{\text{collapse}}$. Each application to H or L is keyed by $id(w_\kappa)$ and is idempotent. Recovery replays every transaction not marked complete on both sides, applying only the missing append, before ordinary processing resumes.

Theorem (Strengthened Ledger Completeness). Under (P1), (P2), (P4), and (P5) from Chapter 4, together with either atomic dual publication or (P3'),

$$\forall \kappa \in H_t, \qquad \exists!\, \ell_\kappa^{\text{collapse}} \in L_t$$

holds at every quiescent machine state, including every state exposed after crash recovery has completed.

Proof. The crash-free cases are Chapter 4's induction. Under

(P3'), a crash can occur before either append, after the H append only, after the L append only, or after both. Recovery consults the durable transaction record and the idempotent application markers. In the first case it applies both sides; in either middle case it applies only the missing side; in the last case it applies neither. Thus every transaction reaches one append on each side and never two. Premises (P4) and (P5) preserve uniqueness across distinct commits. Recovery completes before the resulting state is exposed, so every visible $\kappa \in H_t$ has exactly one matching witness in L_t. ■

The qualification "quiescent" is essential. During recovery, or while a write-ahead transaction is durably prepared but only one side is visible, the pointwise invariant can temporarily be false. The protocol guarantees restoration before normal readers or writers are admitted, not metaphysical simultaneity across separate stores.

Load-bearing tamper evidence.

Theorem (Anchored Hash-Chain Tamper Evidence). Assume canonical, injective encodings for ledger records, domain separation, collision resistance of Hash, and an authentic reference to a later head $r_n = h(\ell_n)$. If a committed field of ℓ_i, $i \leq n$, is altered while the anchored head remains fixed, validation of the chain segment from i through n rejects except with negligible collision probability.

Proof. Altering a committed field changes the canonical encoding of ℓ_i. Equality of its old and new hashes would therefore yield a collision by Chapter 5's reduction. If the successor remains unchanged, its stored predecessor commitment no longer equals the recomputed hash. If an adversary rewrites the successor to repair that link, the successor's own encoding and hash change; the same dilemma recurs at every later record. To arrive at the unchanged authentic head r_n, the rewritten suffix must eventually produce the old hash from a distinct input, again yielding a collision. Hence validation against the authentic anchor rejects except with negligible probability. ■

For the special counterfactual in which the suffix is recomputed honestly from the altered record without adversarially compensating

fields, every descendant hash differs except with negligible probability. The operational detection claim is narrower: a verifier must obtain and validate the intervening chain segment, or receive an authenticated proof structure designed for abbreviated verification. Possession of a head hash alone does not locate an edit and does not eliminate the need to authenticate a path to it.

The theorem establishes none of correctness at initial entry, availability, global non-equivocation, or honest timestamping. It establishes only that an alteration cannot be reconciled with an already authentic later anchor without breaking the hash assumption.

Chapter 19

Redaction, Secrecy, and Cryptographic Erasure

19.1 Persistent occurrence, removable content

Chapter 5 separated an event's content commitment from the retrievability of its payload. The event identifier, commitment, and ledger position persist; what redaction can remove is the ability to resolve the immutable payload handle p to plaintext. Redaction is therefore structurally closer to sealing than expungement. It cannot make the encounter disappear without abandoning append-only history.

The distinction is implemented through an external resolver $\mathsf{Resolve}_t(p)$, not by editing the event envelope. Before redaction, the resolver may return ciphertext, a storage location, or authorized plaintext. After redaction it returns a typed tombstone or access denial. The handle p itself remains fixed inside the event's identity core.

19.2 Mechanisms

A bare hash of low-entropy content permits dictionary testing. A randomized commitment

$$h = \mathsf{Hash}(\mathsf{domain}_{payload} \parallel \mathrm{encode}(payload, salt))$$

resists this only while a high-entropy salt remains secret; if the salt was published, deleting it later adds no hiding. Encryption with key destruction leaves ciphertext in place while rendering it unreadable, conditional on strong encryption and the absence of surviving key copies. A resolver tombstone distinguishes deliberate redaction from a payload that never existed. Access revocation is weaker but reversible because privileged recovery remains possible.

Selective disclosure, commonly through committed substructures such as Merkle trees, can reveal authorized fields with inclusion proofs. Zero-knowledge proofs can establish a declared property of hidden content without revealing the content itself. Neither mechanism proves that the hidden payload was true; each proves only consistency with a commitment and satisfaction of the specified relation.

19.3 Governance

Every redaction produces a typed authorization record naming its subject, authorizer, policy provision, mechanism, time, and any review or reversal conditions. The redaction record is itself append-only. A resolver may cease returning the payload, but the fact that access was removed and the authority under which it was removed cannot vanish with it.

19.4 Proof obligations and status

Proposition (Redaction Preservation). Let Chapter 5's immutable identity core be

$$c(e) = \langle u, s, h, p \rangle, \qquad id(e) = \mathsf{Hash}(\mathsf{domain}_e \parallel \mathrm{encode}(c(e))).$$

Suppose redaction changes only the resolver state associated with p, or destroys a key needed to open the referenced ciphertext, while leaving the canonical envelope and $c(e)$ unchanged. Then $id(e)$, h, and the canonical envelope hash $h(e)$ remain unchanged.

Proof. None of the arguments to $id(e)$ or $h(e)$ changes. Redaction changes the time-indexed resolution relation, not p itself. Both functions therefore receive byte-identical canonical encodings before and after redaction, while h is retained directly. ■

This formulation preserves Chapter 5 rather than revising it retroactively. Replacing p inside the envelope with a tombstone would change an identity-bearing field and therefore change both the event encoding and, absent a collision, its identifier. The tombstone belongs in the resolver and the redaction ledger record.

The result establishes commitment continuity, not later inspectability or non-repudiation of erased plaintext. Once every opening, key, salt, and payload copy is destroyed, the system may retain a commitment it can no longer open. It can still truthfully say that a specific value was committed at a specific time; it may no longer be able to demonstrate what that value was.

Interlude IV: The World Does Not Roll Back

Chapter 17 built a protocol around a fact that deserves plain expression: some effects cannot be undone, only answered. A database rollback operates on the same internal state as the original transaction. Once an effect enters the world, restoration of a later state is not erasure of the path that produced it.

Physical actuation

A valve that fired has fired; material that was cut remains cut; a vehicle that moved occupied the intervening trajectory. Closing the valve, replacing the material, or returning the vehicle is a new act with new costs and risks. The later action may restore a selected variable while leaving every collateral consequence of the interval intact.

Public communication

A published statement may be corrected, retracted, or removed from its original host, but it cannot be unread, unremembered, or withdrawn from every copy made before revocation. The correction often reaches fewer people than the claim. Chapter 19's redaction controls future retrieval within governed stores; it cannot recall information

already transported beyond them.

Human decision-making

Reliance makes institutional decisions path-dependent. An applicant resigns from another position, a patient undergoes a procedure, or a student allows a deadline to pass because an assurance was given. Rescission does not restore the prior life trajectory. It adds a new decision atop consequences already incurred.

This is why compensation must never be described as undo. Durable intent, receipts, reconciliation, and corrective descendants are honest precisely because they preserve the difference between opposing an effect and making it never have happened. Part V now asks what safety and liveness can mean once that difference is admitted.

Part V

System Properties and Failure

Chapter 20

Safety Invariants

20.1 Chapter thesis

This chapter proves nothing new. It collects the preceding results, identifies their dependencies, and distinguishes abstract-machine invariants from conditional infrastructure claims.

20.2 Six invariants

No unbound transformation and no verification without a candidate hold by the typed transition signatures of Chapters 10 and 11. These are unconditional inside the abstract machine, though an implementation still owes a refinement argument showing that its interfaces enforce those types.

No collapse without a passing certificate follows from Chapter 15's guard: $q = \mathsf{PASS}$, candidate identity and freshness, and an empty blocking residual set are necessary conjuncts. No operational commitment without exactly one audit witness is Chapter 4's seed result strengthened by Chapter 18, and holds at exposed post-recovery states under atomic or recoverable dual publication.

No silent policy substitution is a composite obligation rather than one standalone theorem. The binding must commit to ν_B and its

policy inputs, verification must refer to that same binding, collapse authorization must check the current or grandfathered policy, and Chapter 16's conflict rule must force rebinding when the contract changed. No in-place revision of historical decisions follows abstractly from append-only transition rules; in a physical implementation, collision-resistant chaining makes unauthorized revision detectable against an authentic anchor, while actual prevention requires append-only storage or equivalent replication controls.

20.3 The residual sets, recalled

$$\mathcal{R}_{\text{block}} = \{O_i \in \mathcal{O} \mid a_i = \mathsf{REQUIRED} \wedge s_i \neq \mathsf{SAT}\}.$$

$$\mathcal{R}_{\text{waived}} = \{O_i \in \mathcal{O} \mid a_i = \mathsf{WAIVED} \wedge s_i \neq \mathsf{SAT}\},$$

Collapse requires $\mathcal{R}_{\text{block}} = \varnothing$ and places no eligibility condition on $\mathcal{R}_{\text{waived}}$. Every waived deficit nevertheless remains disclosed with its evidential status and authorization.

20.4 Proof obligations and status

Synthesis only. Typing and guard consequences are unconditional only within the specified transition system. Ledger completeness, policy continuity, history uniqueness, and tamper detection require their named publication, versioning, linearizability, encoding, hash, and authentic-anchor hypotheses. Canonical encoding, domain separation, fail-closed verifier reporting, append-only storage, protocol-appropriate honest consensus quorums, and external actuation contracts remain implementation assumptions. Appendix C records each claim in the appropriate class.

Chapter 21

Liveness and Refusal Deadlock

21.1 Chapter thesis

A system that collapses nothing violates no collapse invariant and remains useless. Permanent quarantine, impossible repairs, cyclic evidential dependencies, unavailable verifiers, and policy churn can all preserve safety while preventing resolution. Liveness therefore requires progress hypotheses that safety alone cannot supply.

21.2 A minimal counterexecution

Let a required obligation demand evidence that does not and will never exist. Strict discharge gating correctly prevents collapse forever. The architecture therefore cannot promise both strict discharge and eventual admission of every request. The defensible liveness target is eventual terminal resolution: collapse or final refusal.

21.3 Proof obligations and status

Theorem (Conditional Liveness). Assume: (L1) every binding has a finite obligation set; (L2) each root permits at most a fixed total number of ordinary reopening attempts, not merely bounded depth along one branch; (L3) bounded fairness, so every pending item is scheduled within a known bound; (L4) every invoked verifier returns a typed verdict within a known bound; (L5) each policy and binding specification remains stable for a known interval long enough for one end-to-end attempt; (L6) exhausted ordinary repair triggers an escalation procedure that terminates within a known bound in admission or final refusal; and (L7) every non-verifier internal stage completes or returns a typed failure within a known bound. Then every submitted event reaches successful collapse or final refusal within a finite computable bound.

Proof sketch. By (L2), only finitely many ordinary attempts can occur. By (L1), (L3), (L4), (L5), and (L7), each attempt is scheduled and completed within a bounded interval rather than being starved, stuck, or perpetually invalidated by policy churn. It either collapses or consumes one of the finite ordinary attempts. Exhaustion invokes escalation, which terminates by (L6). Summing the per-attempt, scheduling, and escalation bounds yields a finite global bound. ■

Ordinary eventual fairness would justify eventual resolution but not the stronger phrase "within bounded time"; bounded fairness and bounded stage execution are what license that conclusion. The result resembles a CAP-shaped tradeoff only informally. It is not an application of the CAP theorem [1].

Chapter 22

Threat Model

22.1 Trust boundary

Sources may submit adaptive malicious inputs. A compromised kernel may diverge from the artifact named by ν_K if deployment attestation fails. A malicious verifier may fabricate discharge vectors rather than merely share a correlated bug. Authorities may issue illegitimate capabilities or policies; clocks may be skewed; storage replicas may suppress, rewrite, or equivocate; and the network may delay, reorder, duplicate, or drop messages.

22.2 Categories of failure

Byzantine behavior is arbitrary and adaptive [2]. Ordinary faults are crashes, timeouts, partitions, and hardware failures without malicious intent. Specification error occurs when honest components faithfully implement an inadequate requirement. Insider abuse uses legitimately held authority for an improper purpose, and therefore calls for separation of duties and audit rather than authentication alone. Institutional capture occurs when the policy-making body itself comes to serve interests contrary to the governed domain. The architecture cannot cryptographically rescue a captured standard; it

will enforce that standard rigorously.

22.3 Proof obligations and status

Load-bearing specification chapter. Chapter 13 permits Byzantine sources and network manipulation but excludes compromise of the signing keys and atomic-head arbiter. Chapter 16 abstracts its trust boundary as linearizability; a consensus realization must retain whatever honest quorum its particular protocol requires, not a generic unspecified majority. Chapter 18 assumes collision resistance and at least one authentic later anchor outside the adversary's rewrite power. These are hypotheses, not a new theorem. Their centralization here prevents one chapter's trust boundary from being silently imported into another.

Chapter 23

Recovery and Replay

23.1 Chapter thesis

Chapter 18 proved crash-safe dual publication at collapse. Recovery across the whole pipeline requires a further distinction: reproducing a deterministic computation is not the same as re-deciding a historical transition. Authentication, refusal, authorization, clock reads, and concurrency outcomes are historically contingent. Recovery must normally reapply their durable records, not ask the current world to decide them again.

23.2 Write-ahead persistence at every boundary

POP must durably witness successful intake or salvage failure before releasing an envelope downstream. REFUSE must persist its decision before communicating or acting upon the disposition. BIND may recompute obligations only when every historical input, including Γ_t, policy, and ν_B, is durably available; otherwise it replays the recorded binding. TRANSFORM may rerun under Chapter 10's replay-equivalence hypotheses, but applying a previously recorded candidate is sufficient when its witness is already durable.

VERIFY may be recomputed only under an independently stated deterministic verifier discipline. Otherwise recovery replays the signed verification record as historical output and never upgrades it by consulting a newer verifier. COLLAPSE uses Chapter 18's write-ahead transaction. PREPARE, ACTUATE, and FINALIZE use Chapter 17's pending-intent and attempt journals: a prepared intent with no attempt witness is pending first actuation, while an attempt without a conclusive receipt is ambiguous and must enter reconciliation. Multiple receipts sharing one intent key are records concerning one requested effect; whether the external system produced one effect or several remains a reconciliation question, not something key equality alone proves.

23.3 Derived indexes

Forward-lineage indexes and ledger projections are caches over canonical records. They may be rebuilt by scanning the ledger and need not block recovery unless an operation depends on them for a safety check. An index used only for query acceleration can return later; an index used to enforce nonce spending or uniqueness is safety-critical state and cannot be dismissed as merely derived without a validated reconstruction first.

23.4 Proof obligations and status

Theorem (Crash-Recovery Equivalence). Let a checkpoint commit to a canonical machine state and a ledger sequence number. Assume: (R1) every post-checkpoint transition that became externally visible has a complete, authenticated durable transition record or a recoverable write-ahead record; (R2) the durable prefix and record order remain intact; (R3) the state-application function for each record kind is deterministic; (R4) application is idempotent under the record's stable identifier; (R5) every historical artifact needed to validate a record remains available; and (R6) recovery completes

before normal readers and writers observe the reconstructed state. Then replaying the ordered durable suffix reconstructs the same canonical internal state that those recorded transitions produced before the crash, excluding explicitly unreconciled external-world outcomes.

Proof sketch. The checkpoint supplies the base state. Induct over the ordered durable suffix. At each step, (R3) applies the same authenticated record to the same preceding canonical state and therefore yields the same successor. If the transition was already applied before the crash, (R4) makes replay a no-op; if it was only prepared, its write-ahead status determines which application remains. Hypotheses (R1), (R2), and (R5) prevent missing, reordered, or unverifiable inputs, while (R6) hides intermediate states in which only part of a multi-store publication has been restored. Induction therefore reconstructs the same canonical internal state. An external effect with no conclusive receipt is not reconstructed by assertion and remains in reconciliation. ■

This theorem intentionally requires deterministic record application, not deterministic historical decision-making. Rerunning POP would generate a new observation time; rerunning REFUSE under current policy could revise the past; rerunning COLLAPSE would enter a new concurrency race. Recovery preserves what the system durably decided, while selective recomputation is permitted only for transitions, such as a suitably confined TRANSFORM, whose replay theorem has separately been established.

Chapter 24

Observability and Restorability

24.1 Chapter thesis

Logs, metrics, and traces are not sufficient merely because they exist. Operational observability is the degree to which the witness plane supports restoration, explanation, and re-evaluation. Restoration reconstructs canonical state under Chapter 23's model. Explanation reconstructs why a decision was made under the policy and evidence then available. Re-evaluation applies a declared current procedure without rewriting the historical result.

24.2 Dependency-closed witness sets

For a scope U, define $W(U)$ as the transitive closure of every canonical record concerning U, every parent and input commitment those records name, and every versioned policy, binding specification, kernel, verifier, witness, and checkpoint required by the requested operation. Sufficiency is purpose-relative. A hash commitment may suffice to check continuity but not to explain content; a redacted payload may make restoration possible while making substantive re-evaluation

impossible.

24.3 Local and global restoration

A local set closes over one event's repair descendants and their referenced artifacts. Under Chapter 21's bound on the total number of ordinary attempts and a bounded escalation protocol, the number of transition records in this closure is bounded by the protocol constants. Its byte size is not thereby bounded: payloads, traces, proofs, and artifacts can vary arbitrarily unless separate size limits are imposed.

Global crash restoration does not require the entire ledger from genesis when a trusted canonical checkpoint exists. Its sufficient set is the latest valid checkpoint, the ordered durable suffix after it, and the historical artifacts needed to validate and apply that suffix. Full historical explanation or audit may still require the earlier ledger. Recovery sufficiency and archival completeness are therefore different properties.

24.4 Proof obligations and status

Supporting characterization. Dependency closure is sufficient under the machine model because every transition reads only its declared inputs and Chapter 23 recovers by applying authenticated records. Necessity is relative to the requested purpose: removing a component is necessary exactly when no remaining record or valid derivation reconstructs the information it supplies. Prediction of future failures and detection of institutional capture are research conjectures, not consequences of this characterization.

Part VI

Implementation and Evaluation

Chapter 25

A Reference Data Model

25.1 Chapter role

The abstract tuples now receive concrete fields and representation invariants. The following condensed schema preserves the identity and replay corrections made in Chapters 5, 10, 11, 15, and 19.

```
EventCore { u, source, content_commit, payload_handle }
Event     { id, u, observed_at, source, content_commit,
            payload_handle, metadata, parent, creation_rank }
Binding   { event_id, event_commit, certificate, obligations, nu_b }
Obligation{ id, predicate, method, witness_class, discharge_policy }
Candidate { output_commit, binding_commit, history_head,
            execution_witness, trace_ref, capabilities, nu_k }
VerificationCore { verdict, candidate_commit, binding_commit,
                   discharge_vector, r_block, r_waived, witness, nu_v }
CollapseProposal { candidate_commit, verification_commit,
                   expected_head, sigma_c, nonce }
CommitCore { id_c, prior_head, candidate_commit, verification_commit,
             sigma_c, nonce, committed_at, authority }
LedgerCore { seq, kind, subject, payload_commit, policy,
             parents, recorded_at, actor, prior_record_hash }
ResolverRecord { payload_handle, disposition, authorization, recorded_at }
Signed<T> { object: T, signer, algorithm, signature }
```

The identifier is

$$id(e) = \mathsf{Hash}(\mathsf{domain}_e \parallel \mathrm{encode}(\langle u, s, h, p \rangle)).$$

Observation time and metadata remain outside the identity core, while the payload handle p is immutable. Redaction changes a *ResolverRecord*, not the event. Candidate records retain both $h(b)$ and the history head they read; verification records retain $h(x)$ and $h(b)$, making SameCandidate representable.

25.2 Canonical encoding

Fields are encoded in fixed schema order. Every variable-length component is length-prefixed, every sum type carries an explicit variant tag, maps and sets are sorted by declared canonical keys, scalar encodings are unique, and Unicode and numeric normalization rules are versioned. These rules together, not field order alone, make encoding injective over well-typed normalized values.

Each object kind has a fixed reserved domain tag. Appendix D is the registry. An identifier is computed over an unsigned core; authentication then wraps that core in Signed$\langle T \rangle$. A signature is never included in the bytes whose identifier it authenticates, avoiding circular self-reference. Ledger signatures likewise authenticate *LedgerCore*, and commit signatures authenticate *CommitCore*.

25.3 Migration and payload storage

Schema migration produces a versioned derived view and never rewrites original bytes. Payload handles may be ordinary opaque locators, randomized content-addressed capabilities, or keyed addresses. A public locator of the form $h(payload)$ is unsuitable for low-entropy secret payloads because it enables dictionary testing; integrity is checked against the event's commitment after authorized retrieval instead.

25.4 Proof role

Implementation-bearing. The schema exhibits a realizable witness for Chapter 5's encoding and domain-separation hypotheses, conditional on the leaf encoders and normalization rules being injective. Cross-language golden vectors must confirm that independent implementations produce identical bytes and identifiers.

Chapter 26

A Minimal Spherepop Service

26.1 Pure decisions and effectful adapters

The service separates transition decisions from persistence and external effects. A pure transition receives explicit state, authenticated facts, clock values, and inputs, and returns a typed result plus records to publish. It does not read a live clock, database, network, or random source. POP's authentication result and observation time are therefore explicit inputs; COLLAPSE can construct and validate a proposed update purely, while only the persistence adapter can perform the linearizable compare-and-append.

```
bind(event, admission, constraints,
     environment, binding_version)
  -> Bound(binding)
   | Deferred(reason)
   | Conflict(reason)
```

The adapter implements durable witnesses, write-ahead publication, nonce and attempt journals, checkpointing, and the external PREPARE–ACTUATE–FINALIZE protocol. Keeping decision logic separate makes failures of storage semantics distinguishable from failures of transition logic.

26.2 Service boundaries and idempotency

Submission should be asynchronous by default: intake returns an identifier, while later status or notification reports refusal, repair, escalation, or commitment. A synchronous facade may wait for fast terminal cases but cannot make instant resolution part of the protocol contract.

A client idempotency key is mapped durably to the source-scoped occurrence token u before envelope construction. It complements event identity without changing it. Because observation time and metadata are already excluded from the identity core, client retries do not need to manufacture new identities merely because transport metadata changed.

Transaction scopes follow safety boundaries. Early transitions atomically publish their outputs and ledger witnesses within their relevant stores. COLLAPSE requires the linearizable head operation of Chapter 16 and the recoverable dual publication of Chapter 18. No claim that an operation is "local" excuses it from the witness-plane append.

26.3 Deterministic fixtures

Golden fixtures check byte-identical BIND outputs for fixed $(e, \alpha, C, \Gamma_t, \nu_B)$, and replay fixtures check TRANSFORM under all of Chapter 10's hypotheses, including pinned state and captured nondeterminism. Fixtures test the implementation of those hypotheses; they do not replace the proofs.

26.4 Proof role

Refinement obligation. A mapping α from implementation states to abstract states must make every visible service operation simulate a valid abstract transition, with stuttering steps permitted for internal persistence work. Full refinement requires models of the ac-

tual database isolation level, message delivery, compare-and-append primitive, recovery protocol, clocks, and external effect boundary. Documentation and tests provide conditional evidence; they are not a completed simulation proof.

Chapter 27

Testing the Boundaries

27.1 Seven test families

Property-based tests exercise BIND determinism, verdict partition and error normalization, waiver independence, and encoding round trips over generated well-typed inputs. State-machine tests generate transition sequences and check ledger correspondence, sole-writer behavior, lineage acyclicity, and nonce spending after every step.

Bounded model checking exhaustively explores small arbiter and recovery models, especially competing collapses against one head. Its completeness applies only to the stated finite model and bound. Fuzzing attacks POP parsing, canonical decoding, signature envelopes, and typed failure paths with malformed bytes.

Fault injection crashes the service before and after each durable boundary, testing Chapter 23's complete-record premise and idempotent application requirement. Concurrency tests exercise the deployed sequencer or consensus mechanism rather than its simplified model. Adversarial fixtures instantiate certificate replay, substitution, stale authorization, rollback, resolver tampering, forged waiver, and equivocation attempts.

27.2 Proof role

Empirical support, not proof. Passing samples does not establish a universal theorem, and bounded exploration does not establish unbounded correctness. The value of the mapping is diagnostic: a failure names the specific invariant, assumption, transition boundary, and durable record whose implementation diverged from the abstract machine.

Chapter 28

Evaluation Programme

28.1 Metrics

Measure throughput and latency separately for each stage; audit amplification as records and bytes in L relative to commits and bytes in H, treating empty denominators explicitly; recovery time against checkpoint age and unapplied suffix length; verification cost by verifier and quorum; stale-head and commit-conflict rates; repair success; indeterminate and error verdict rates; escalation frequency; and storage, replay, and validation cost for each execution-witness style.

The proposed "false refusal" metric is rejected because it contradicts Chapter 8: a descendant's later success does not make the parent's earlier refusal false. The measurable quantity is *first-pass repair burden*: the fraction of ultimately successful lineages that required at least one refusal and repair, accompanied by an audit of whether each original refusal was correct under its contemporaneous policy. Burden measures friction without rewriting historical truth.

High audit amplification may reflect either pathological churn or appropriately rich evidence; high stale-head frequency may motivate sharding; low repair success may reveal decorative repair paths; frequent indeterminacy may reveal unobtainable witness classes. Metrics

diagnose questions rather than answer them by themselves.

28.2 Methodological discipline

Confirmatory comparisons preregister workloads, metrics, baselines, exclusion rules, and fault injections before results are inspected. Continuous operational monitoring need not be preregistered, but any later inferential claim must distinguish exploratory discovery from confirmatory evaluation.

28.3 Proof role

Empirical chapter. Performance and assurance are orthogonal. A fast non-linearizable arbiter remains unsafe, and a low repair burden does not prove that refusal policy is just. Benchmarks describe observed costs and frequencies; they establish none of the formal properties in Chapters 4 through 24.

Chapter 29

Three Applied Case Studies

29.1 Scientific claims under unsettled evidence

A manuscript and its supporting materials form an event. Journal intake is POP; editorial disposition is REFUSE's total gate, returning either admission or a typed refusal. Revision is REOPEN: a new submission with a new occurrence token and explicit lineage. BIND fixes methodological, disclosure, and statistical obligations before the reported candidate is assessed. Registered protocols, versioned datasets, executable analysis, and environments supply a domain analogue of ν_K. Peer and statistical review instantiate VERIFY, with INDETERMINATE expressing that the available evidence does not settle a required question. Publication is COLLAPSE only relative to the journal's own authoritative publication history, not to the field's globally accepted beliefs.

Later replication failure may justify correction, retraction, or a new study. It does not erase what reviewers saw or which standards governed the earlier decision, but neither does historical preservation certify that the earlier review was correct relative to its own evidence.

Fraud, oversight, or misapplication may have made it defective even then.

The mapping imports fallible human review, domain standards, and a journal-local arbiter. Science as a whole has no linearizable global head, so History Never Forks does not apply to the literature. Human reviewers also do not reliably self-report their own failure modes in the manner Chapter 11's trusted wrapper requires.

29.2 A distributed software transformation

A pull request is an event; continuous-integration intake, policy gates, required tests, reproducible builds, test verdicts, and merge correspond to POP, REFUSE, BIND, TRANSFORM, VERIFY, and COLLAPSE. Build identity pins the container, compiler, configuration, dependency lockfile, and relevant environment. A flaky test belongs under INDETERMINATE or ERROR, depending on whether the test completed inconclusively or the checking machinery failed. A logged emergency override is a waiver, not a silent conversion of UNSAT into SAT. An outdated pull request is the familiar stale-head case.

Version-control hashes are close analogues of domain-separated content commitments, but they are not literally Chapter 5's event identifiers: commit objects commonly include parents, authorship metadata, and messages, whereas Spherepop identity includes a source-scoped occurrence token and immutable payload handle. Deployment then invokes PREPARE, ACTUATE, reconciliation, and FINALIZE rather than treating merge as proof that production changed.

The mapping imports uncompromised build infrastructure and collision-resistant hashing. The practical SHA-1 collision demonstrated in 2017 shows why hash agility is an operational requirement rather than a decorative caveat [5]. Supply-chain compromise violates the kernel or verifier identity boundary; the architecture names that failure without solving it.

29.3 An irreversible actuator

A command to position an industrial valve is admitted and authenticated at POP, checked against interlocks and operator capability at REFUSE, and bound to pressure, conflict, authorization, and sensor obligations. TRANSFORM simulates a pressure and flow trajectory without touching the valve. VERIFY checks that candidate, returning INDETERMINATE when uncertainty straddles a safety threshold.

A successful collapse proposal enters Chapter 17's external-effect protocol: PREPARE durably records intent, ACTUATE sends the physical command, and FINALIZE appends the operational commit only after a conclusive receipt or reconciliation. Flow that already occurred cannot be rolled back; closing the valve is a new compensating event.

The mapping imports trustworthy sensors, adequate physical models, actuator identity, and an external duplicate-handling contract. A correct simulation can omit a decisive physical variable, exactly as Chapter 14 and Interlude III warn.

29.4 Proof role

Model validation, not theorem. Each case demonstrates a typed correspondence and exposes its imported assumptions. The formal guarantees hold only after a domain has genuinely supplied those assumptions; an elegant mapping cannot establish honest reviewers, uncompromised infrastructure, adequate models, or trustworthy sensors.

Part VII

Integration and Consequence

Chapter 30

From RSVP to Event Histories

30.1 Standing caution

RSVP is treated here as a possible continuous scalar-vector substrate and KES as a provisional account of event-mediated historical fixation. No detailed KES state vector, admissibility field, or monotonic-contraction law is treated as canonical [10]. In particular, global monotonic contraction is too strong if it forbids repair and reopening.

30.2 Bridge principle

Spherepop is a candidate worked instance of the abstract pattern KES gestures toward. RSVP would supply continuous dynamics; KES would name the emergence of historically fixing events; Spherepop supplies typed computational boundaries for encounter, admission or refusal, binding, candidate production, verification, and commitment, with the ledger witnessing each boundary. This is a correspondence of roles, not a derivation of one theory from another.

The apparent conflict with monotonic contraction can be localized. A refused event never regains admissibility in place. REOPEN creates

a descendant event with a fresh identity and its own admissibility judgment. Contraction may therefore be coherent per event-attempt while the global possibility structure branches through new descendants. The growing audit history L and selective operational history H provide distinct indices that the earlier sketch lacked.

Any genuine RSVP bridge would still need an observation or discretization map from continuous field state to an event envelope, and an admissibility map from field-relative conditions to REFUSE and BIND outputs. Their domains, codomains, covariance properties, and dimensional consistency remain unspecified.

30.3 Proof role

Speculative integration. The chapter asserts only that the types admit a plausible correspondence. RSVP equations do not entail Spherepop transitions, and Spherepop's theorems do not validate KES mathematics.

Chapter 31

Distinction Holonomy

31.1 Paths and terminal equivalence

The terminology and general framing are inherited from the author's separate, unpublished treatment of distinction holonomy [7]; the reduced definitions here do not validate that larger formalism. Because a repair graph can branch, a path π is one causally continuous, ordered root-to-terminal branch together with the transition records generated along that branch. The entire descendant graph is not one path. Two paths are terminally equivalent under a declared projection T, written $\pi_1 \sim_T \pi_2$, when $T(\pi_1) = T(\pi_2)$. The projection may compare only success, committed candidate content, or a finer terminal structure.

31.2 Transported residue

Rather than an undefined set-theoretic "complement," define a declared residue map R_T that extracts the path information intentionally ignored by T: refusals, repairs, elapsed time, verifier identities, policy versions, waivers, authorities, and other provenance. Terminally equivalent paths may therefore carry different R_T values.

An actual holonomy theory would additionally require a transported object, a transport operator along composable path segments,

a notion of closed loop or alternative-path comparison, and an invariant showing how transport around that loop fails to return the object unchanged. A distance between residues is not by itself holonomy, and dependence on an arbitrary projection must be analyzed rather than hidden.

31.3 Proof role

Prospective formalization. The chapter supplies disciplined definitions and a list of missing structures. It claims no holonomy theorem.

Chapter 32

MEM|8 and Resonant Recall

32.1 Archive, salience, and admissibility

The resonant-recall proposal derives from separate MEM|8 manuscripts [8, 9]. It is treated here as a candidate integration model rather than as a result of Spherepop. Archive is the canonical L and H. Salience is a mechanism for ranking what may be useful to recall. Admissibility determines what a requester and the retrieval computation itself may access. These are three distinct relations. A highly resonant record may still be forbidden, and a complete archive does not imply complete approximate recall.

32.2 Explicit retrieval map

Any derived memory system must declare

$$\rho(q, c) = \mathrm{Rank}_{q,c}(\mathrm{Eligible}_c(L)).$$

Eligibility must constrain the index or retrieval computation before unauthorized records are scored. Filtering only the displayed results

after a global ranker has processed secret material is insufficient because scores, embeddings, caches, and associations can themselves leak information. Deployments therefore need policy-partitioned indexes, admissibility-preserving representations, or cryptographic computation that enforces the same boundary.

Ledger completeness and tamper evidence remain properties of the upstream archive; they do not "survive" as completeness properties of a lossy latent memory. A resonant layer may omit or distort retrieval and must never be the only route to canonical records. Exact addressed lookup and verification remain available through the archive.

32.3 Proof role

Speculative integration. MEM|8 is one candidate salience model. Spherepop neither entails coupled-oscillator recall nor validates a latent trajectory formalism. The established contribution is the integration constraint separating archive, admissibility, and salience.

Chapter 33

Institutions as Event Systems

33.1 Legal procedure

Filing resembles POP; threshold dismissal can resemble REFUSE; amendment is a new descendant rather than mutation; legal elements resemble bound obligations; and judgment resembles commitment. The mapping must not be made too exact. Standing is principally an authority or capability question, jurisdiction concerns the forum's authority and governing policy, and neither is simply provenance.

A hung jury is not automatically Chapter 11's INDETERMINATE. It may reflect evidential uncertainty, disagreement over interpretation, or a non-unanimous aggregation rule. An acquittal likewise means the prosecution did not satisfy its burden, not necessarily that innocence was positively proved. Legal verdicts need their own typed aggregation semantics before the four verifier verdicts can be imported.

Vacatur is also subtler than in-place deletion. A well-kept docket preserves the original judgment and appends the appellate act that removes its authoritative legal effect. Doctrinally the judgment is vacated; historically the issuance and vacatur both remain. This resembles descendant correction and status supersession more closely

than erasure, although legal consequences and precedential treatment exceed Spherepop's present model.

33.2 Governance and interpretation

Bill introduction, committee disposition, procedural constraints, voting, and enactment offer a rough mapping. Real legislatures, however, renegotiate both the candidate and the standards for accepting it throughout deliberation. Consensus bodies may revise the target and proposal together. This is continuous rebinding, or something beyond the pipeline, rather than verification against a contract fixed in advance.

The deepest disanalogy is interpretation. Courts and institutions often decide what "reasonable," "material," or "in the public interest" means through the act of applying it. Spherepop assumes that enough interpretive labor has already occurred for BIND to emit checkable obligations. It has no theorem reducing institutional judgment to predicate evaluation.

33.3 Proof role

Analogical and diagnostic. The vocabulary can locate conflated boundaries, but it neither proves institutional adequacy nor supplies a theory of interpretation.

Chapter 34

The Ethics of Preserved Refusal

34.1 The ethical tension

Permanent preservation supports accountability while creating durable risks for subjects of unsubstantiated allegations, third parties named incidentally, and people whose individually correct denials are later aggregated into an improper adverse profile. Structural metadata and commitments can themselves be sensitive even after payload erasure, especially when values are low-entropy or linkable.

34.2 Four principles

Proportional retention assigns different availability and disclosure periods to unsubstantiated attempts, denied claims, and committed findings without rewriting the structural ledger. Contestability requires notice, comprehensible reasons, a usable reopening process, and often practical support; an API endpoint alone is not meaningful recourse.

Redaction may protect a vulnerable third party or conceal misconduct by a powerful actor. An authorization record proves that

a decision was recorded, not that it was ethically justified. Access control should therefore default to directly involved parties and independently accountable audit functions, with wider disclosure requiring a specific recorded basis. Chapter 18's projections make these policies implementable but cannot choose among them.

34.3 Proof role

Normative chapter. Completeness, tamper evidence, redaction continuity, and authorization records do not determine retention duration, default access, or legitimate purpose. Formal consistency cannot choose the ethical premise.

Chapter 35

Conclusion: Commitment Without Conflation

35.1 Four promises

A trustworthy event architecture does not promise that every committed state is true or good. It makes four narrower promises under named conditions. Encounter does not masquerade as admission: POP records occurrence and REFUSE's total gate separately returns admission or a reason-bearing non-admission. Execution does not masquerade as verification: TRANSFORM produces a candidate, while VERIFY alone evaluates the obligations fixed by BIND.

Verification does not masquerade as truth: PASS means that the declared required obligations were reported SAT under their declared methods and valid checking relation. It says nothing by itself about the adequacy of the obligation set or the goodness of the governing policy. Commitment does not occur without reconstructible warrant: the collapse guard, sole-writer rule, and witness plane connect every visible post-recovery commit to exactly one collapse witness under the publication hypotheses.

35.2 Open boundary

Institutional capture, interpretive judgment, ethical retention, RSVP–KES integration, distinction holonomy, and MEM|8's resonant dynamics remain open. They are not omissions disguised as future certainty. They mark the boundary between the architecture's proved properties and the surrounding questions that require other theories, empirical evidence, or normative argument.

35.3 Closing

The witness plane cannot make policy legitimate or preserved allegations harmless. It can prevent an attempt from being silently rewritten as success, a computation from being mistaken for authorization, a passing check from being inflated into truth, and a commitment from being detached from the historical warrant that produced it. This is not certainty. It is accountable commitment, falsifiable at named boundaries and offered as nothing more.

35.4 Proof role

Synthesis only. No new theorem is introduced.

Appendix A

Notation and Type Catalogue

A.1 State and event types

$$\mathcal{S}_t = \langle Q_t, L_t, H_t, P_t, A_t, W_t \rangle,$$

where Q_t is pending work, L_t the audit ledger, H_t operational history, P_t the policy and version registry, A_t authority state, and W_t durable protocol state containing write-ahead transactions, pending external intents, actuation attempts, receipts, checkpoints, and spent nonces.

An event has immutable identity core and full envelope

$$c(e) = \langle u, s, h, p \rangle, qquade = \langle id, u, t, s, h, p, m, parent, rank \rangle,$$

$$id(e) = \mathsf{Hash}(\mathsf{domain}_e \parallel \mathrm{encode}(c(e))).$$

Here u is the source-scoped occurrence token, t system observation time, s authenticated source, h randomized or otherwise policy-appropriate content commitment, p immutable payload handle, and m non-causal metadata. Redaction changes $\mathsf{Resolve}_t(p)$, never p or the envelope.

A.2 Transitions

$$
\begin{aligned}
\mathsf{POP}(e_{\mathrm{raw}}) &\to \mathsf{POP\_RESULT},\\
\mathsf{REFUSE}_{\sigma_t}(e, H_t) &\to \mathsf{ADMIT}(\alpha) \mid \mathsf{REFUSAL}(d),\\
\mathrm{REOPEN}(id(e), \Delta', w, u') &\to e',\\
\mathsf{BIND}(e, \alpha, C, \Gamma_t, \nu_B) &\to \mathsf{BOUND}(b) \mid \mathsf{DEFERRED}(d_B) \mid \mathsf{CONFLICT}(f_B),\\
\mathsf{TRANSFORM}_{K,\theta}(b, H_t) &\to \mathsf{CANDIDATE}(x) \mid \mathsf{FAILURE}(f_K),\\
\mathsf{VERIFY}_{V,\sigma_v}(x, b, H_t) &\to v,\\
\mathsf{COLLAPSE}(c, H_t, L_t, W_t) &\to \mathsf{COLLAPSE\_RESULT},\\
\mathrm{PREPARE}(c) &\to \iota,\\
\mathrm{ACTUATE}(\iota) &\to \epsilon \mid \mathsf{AMBIGUOUS},\\
\mathrm{FINALIZE}(\iota, \epsilon) &\to \kappa.
\end{aligned}
$$

Every branch writes a typed audit witness. Early-stage transitions do not write H. External-effect finalization invokes the same successful commit publication primitive as direct collapse; it is not a second independent writer.

A.3 Refusal, obligations, and verdicts

$$d = \langle r, \rho, \Delta, \tau, \pi, terminal \rangle,$$

with r in quarantine, repair, evidence-only, defer, redact, or deny. Terminality is an orthogonal Boolean set only by the bounded escalation procedure; it is not a seventh disposition. Reason grounds are identity, schema, provenance, policy, capability, and temporal.

$$O_i = \langle P_i, m_i, w_i, \delta_i \rangle, \quad d_i = \langle s_i, a_i \rangle,$$

where s_i is SAT, UNSAT, UNKNOWN, or NOT_CHECKED, and a_i is Required or Waived. The blocking and disclosure residuals are

$$\mathcal{R}_{block} = \{O_i : a_i = \mathsf{REQUIRED} \wedge s_i \neq \mathsf{SAT}\}, \quad \mathcal{R}_{waived} = \{O_i : a_i = \mathsf{WAIVED} \wedge s$$

The overall verdict q is PASS, FAIL, INDETERMINATE, or ERROR. ERROR normalizes every s_i to NOT_CHECKED.

A.4 Candidate, verification, and collapse

A candidate commits to output, binding, and read head:

$$x = \langle y, h(b), h(H_t), \chi, T, \mathcal{C}, \nu_K \rangle.$$

A verification core contains $q, h(x), h(b), \mathbf{d}, \zeta, \mathcal{R}_{block}, \mathcal{R}_{waived}, \nu_V$ and is authenticated by a separate signature envelope. Thus

$$\text{SameCandidate}(x, v) \iff h(x) = v.h_x \wedge h(b) = v.h_b.$$

$$c = \langle h(x), h(v), h(H_t), \sigma_c, \eta \rangle,$$

and the unsigned commit core is

$$\kappa = \langle id_c, h(H_t), h(x), h(v), \sigma_c, \eta, t_c, a_c, \mathcal{R}_{waived} \rangle.$$

Its signature envelope is external to the core, and the resulting head is

$$h(H_{t+1}) = \mathsf{Hash}(\mathsf{domain}_H \parallel h(H_t) \parallel \text{encode}(\kappa)).$$

Collapsibility requires PASS, empty $\mathcal{R}_{block}$, freshness, and candidate identity. Freshness has elapsed-time, state-head, policy, capability, and evidence components.

A.5 Hypothesis catalogue

Encoding hypotheses are injective canonical encoding and exclusive domain separation. History hypotheses are unique genesis and linearizable compare-and-append. Liveness hypotheses $(L1)$–$(L7)$ are finite obligations, a bound on total ordinary attempts per root, bounded fairness, bounded verifier termination, a sufficiently stable policy

interval, terminating escalation, and bounded completion or typed failure for every other internal stage.

Recovery hypotheses $(R1)$–$(R6)$ are complete durable records for every visible transition, durable ordered-prefix integrity, deterministic record application, idempotent application by stable record identifier, availability of historical validation artifacts, and recovery isolation before state is exposed. These replace the earlier and stronger assumption that historical decisions themselves can always be recomputed deterministically.

A.6 Speculative notation

A distinction path π is a single causally continuous root-to-terminal branch. Terminal equivalence is $\pi_1 \sim_T \pi_2$ iff $T(\pi_1) = T(\pi_2)$; R_T is the declared residue projection. No criterion-independent transported complement or holonomy operator is assumed.

Appendix B

Operational Semantics

B.1 Frame and publication conventions

Any state component absent from a rule's conclusion is carried unchanged. Every transition first creates an authenticated durable transition record, then applies that record idempotently to canonical state. The notation $\mathsf{emit}(r)$ appends the audit witness for record r. The operation $\mathsf{commit}(w_\kappa)$ is the sole abstract writer of H: direct COLLAPSE and external FINALIZE both call this one primitive.

B.2 Early stages

Representative success rules are

$$\frac{\langle \mathsf{POP}, e_{raw}\rangle \in Q_t \quad \mathrm{normalize}(e_{raw}) = e}{Q_{t+1} = Q_t - \{\langle \mathsf{POP}, e_{raw}\rangle\} \cup \{\langle \mathsf{REFUSE}, e\rangle\}, \quad L_{t+1} = \mathsf{emit}(\ell^{pop})} \text{ [POP]}$$

and

$$\frac{\langle \mathsf{REFUSE}, e\rangle \in Q_t \quad \mathsf{REFUSE}_{\sigma_t}(e, H_t) = \mathsf{ADMIT}(\alpha)}{Q_{t+1} = Q_t - \{\langle \mathsf{REFUSE}, e\rangle\} \cup \{\langle \mathsf{BIND}, e, \alpha\rangle\}, \quad L_{t+1} = \mathsf{emit}(\ell^{admit})} \text{ [ADMIT].}$$

The refusal branch removes the work item, appends ℓ^{ref}, and schedules no downstream computation. POP failure and duplicate arrival

likewise append typed records; a duplicate arrival does not create a second event node.

BIND emits Bound, Deferred, or Conflict. Only Bound schedules TRANSFORM. TRANSFORM emits Candidate or Failure; only Candidate schedules VERIFY. VERIFY always emits a full typed result, and only PASS with an empty blocking residual may schedule a collapse attempt. By the frame convention all these rules satisfy $H_{t+1} = H_t$.

B.3 Commit publication

For an internal effect, a successful guard creates a durable transaction

$$w_\kappa = \langle \kappa, \ell_\kappa^{collapse} \rangle$$

and invokes $\mathsf{commit}(w_\kappa)$, which atomically, or through the recoverable protocol of Chapter 18, appends one κ to H and one witness to L. A failed guard appends only a typed collapse-failure record.

For an external effect, PREPARE places ι in durable protocol state and appends its witness. Before transmission, ACTUATE appends an attempt record. A conclusive receipt is stored; a missing receipt marks the intent ambiguous. FINALIZE validates (ι, ϵ), constructs w_κ, and calls the same commit primitive. Consequently the sole-writer premise is preserved even though two routes reach it.

B.4 Recovery rule

Given checkpoint C_j and authenticated ordered durable suffix $(r_{j+1}, \ldots, r_n)$, recovery folds the deterministic application function:

$$\mathcal{S}_n = \mathrm{foldl}(\mathsf{apply}, C_j, [r_{j+1}, \ldots, r_n]).$$

Already-applied identifiers are no-ops. Historically contingent decisions are not rerun. A prepared or attempted external intent without

a conclusive receipt enters reconciliation rather than being guessed into success or failure.

Appendix C

Core Invariants and Proof Obligations

This appendix is the canonical registry of formal claims. Each entry must give its complete hypotheses, conclusion, proof, dependencies, failure conditions, and prospective mechanization status. It must not flatten the architecture's claims into a uniform sequence of nominal theorems.

C.1 Class I: Provable as stated in the transition system

C.1.1 Early-Stage Non-Interference

For POP, REFUSE, BIND, TRANSFORM, and VERIFY, prove by inspection of the small-step rules that no transition writes operational history:

$$H_{t+1} = H_t.$$

The result is load-bearing despite its short proof because every later safety argument depends on COLLAPSE being the sole writer of H.

C.1.2 Layered Admission Non-Implication

Provide the complete counterexample matrix showing that epistemic, computational, operational, and irreversible admission are pairwise non-implying.

C.1.3 Identifier Collision Reduction

Reduce an event-identifier collision to either non-injective canonical encoding or a collision in Hash. Do not characterize this reduction as a new proof of hash collision resistance.

C.1.4 Obligation-Generation Well-Definedness

Prove deterministic obligation generation for well-typed (e, C, Γ_t, ν_B). Limit the claim to syntactic checks and declared decidable fragments; general semantic satisfiability remains outside the result.

C.1.5 Verdict Partition and Error Containment

Prove that the four verification verdicts are exhaustive and exclusive under the verifier operational semantics. Prove separately that $q = \mathsf{ERROR}$ prevents every s_i from retaining SAT.

C.1.6 Collapse Guard Respects Waivers

Prove that only $\mathcal{R}_{\text{block}}$ controls collapse eligibility, while every member of $\mathcal{R}_{\text{waived}}$ remains disclosed with its evidential status and waiver authority.

C.1.7 Ledger Completeness

Prove by induction that each $\kappa \in H_t$ has exactly one corresponding collapse witness in L_t, at every exposed quiescent state, using the single commit primitive and atomic or recoverable dual publication. Intermediate recovery states need not satisfy the pointwise invariant and must not be externally exposed.

C.1.8 Hash-Chain Tamper Evidence

Given an authentic trusted later head and collision-resistant hashing, prove that an altered record cannot validate to that fixed anchor except with negligible collision probability. If the suffix is honestly recomputed, every descendant hash changes; detection still requires the segment or an authenticated proof path.

C.1.9 Redaction Preservation

Prove that authorized payload unavailability leaves $id(e)$ and $h(e)$ unchanged when the canonical envelope, immutable handle p, and content commitment are retained. Redaction changes resolver state, not p. Record non-repudiation of erased plaintext as a non-preserved property.

C.2 Class II: Provable only under named additional hypotheses

C.2.1 History Never Forks

Assume a unique genesis history and a linearizable compare-and-append primitive implemented by a single arbiter or equivalent consensus mechanism. Prove by induction that at most one candidate can extend a given current head and that the operational history remains unique.

C.2.2 Transformation Replay Equivalence

Assume deterministic kernels, canonical inputs, pinned kernel and environment identity, equal initial state, and no undeclared nondeterminism. Prove equality of replayed outputs up to the declared trace equivalence.

C.2.3 Crash-Recovery Equivalence

Assume complete authenticated records for visible transitions, an intact ordered durable prefix, deterministic and idempotent record application, available historical validation artifacts, and recovery isolation. Prove that checkpoint plus suffix reconstructs the same canonical internal state, excluding unreconciled external effects. Historical decisions are not rerun.

C.2.4 Freshness and Substitution Resistance

Under collision resistance, signature unforgeability, nonce uniqueness, correct freshness predicates, and atomic current-head comparison, bound the probability that a probabilistic polynomial-time adversary commits a stale or substituted candidate.

C.2.5 Conditional Liveness

Under finite obligations, a bound on total ordinary attempts per root, bounded fair scheduling, bounded verifier termination, sufficiently stable policy intervals, bounded completion or typed failure for other stages, and terminating escalation, prove terminal resolution within a computable bound. Ordinary eventual fairness supports eventuality, not a numerical bound.

C.2.6 ACTUATE Idempotency

Treat external idempotency as a capability-contract assumption. Spherepop can prove stable intent, pre-transmission attempt journaling, and idempotency-key reuse, but it cannot derive the external system's behavior from its internal transition rules.

C.3 Class III: Negative results, limitations, and non-claims

C.3.1 Correlated Verifiers

Show that k verifiers sharing a dependency gain no protection against a fault in that dependency. Do not multiply marginal error rates without an independence argument.

C.3.2 Verification Is Not Truth

VERIFY establishes obligation satisfaction relative to a declared policy and verifier. It does not certify that the obligation set is complete, adequate, or ethically sound. Present counterexamples rather than an irrelevant appeal to Gödel.

C.3.3 Tamper Evidence Is Not Correctness

Hash chaining does not prove that an original record was truthful, that all observers received the same head, that data remain available, or that recorded times are honest.

C.3.4 Compensation Is Not Reversal

An irreversible external effect may be answered by a later compensating act, but the original effect is not thereby removed from history.

C.4 Mechanization plan

Formalize the abstract transition system first, then Class I invariants. Add the linearizable-CAS model before attempting History Never Forks. Encode cryptographic results as reductions to assumptions rather than reimplementing cryptographic proofs. Maintain a machine-readable dependency table connecting each theorem to its hypotheses and the chapters that cite it.

Appendix D

Canonical Serialization and Hashing

D.1 Encoding requirements

For each schema version, encode is pure and injective over well-typed normalized values. Field order and scalar representations are fixed; sum variants and object versions carry explicit tags; variable-length fields use fixed-width unsigned length prefixes; maps and sets are canonically sorted; Unicode, timestamps, integers, and floating-point values have one admitted normal form. A four-byte length prefix is sufficient only when the schema also rejects values beyond its representable bound.

D.2 Domain registry

The initial registry assigns distinct tags to event core, refusal core, admission certificate, reopening record, obligation, binding, candidate, verification core, collapse proposal, commit core, ledger core, external intent, actuation attempt, receipt, resolver record, checkpoint, and write-ahead transaction. Concrete hexadecimal values are controlled by the registry artifact, not by prose examples. A semantically

incompatible schema version receives a new versioned tag; tags are never silently reassigned.

D.3 Hash and signature agility

$$\mathsf{HashVal} = \langle algorithm, digest \rangle.$$

Equality requires equal algorithm identifiers and digest bytes. A migration record commits under the old chain to the first head under the new algorithm, and the new chain commits back to the old anchored head. Without this explicit bridge, tamper evidence holds only within each algorithm segment.

$$\mathrm{Sign}(sk, T) = \mathrm{Sig}\big(sk, \mathsf{context} \parallel \mathsf{domain}(T) \\ \parallel \mathsf{version}(T) \parallel \mathrm{encode}(T)\big).$$

The signature wraps an unsigned core. Neither the signature nor an identifier derived from that core is recursively included in its own preimage.

D.4 Normative test vector shape

An event identity vector encodes fields in the order (u, s, h, p), each preceded by its length and governed by its leaf-type encoding. Changing t or m leaves $id(e)$ unchanged but changes the full envelope hash. Changing p changes event identity; redaction tests instead change $\mathsf{Resolve}_t(p)$ and assert that both p and $id(e)$ remain unchanged. Cross-language fixtures must cover empty fields, maximum lengths, Unicode normalization, map order, every sum variant, unknown-version rejection, and each algorithm-migration boundary.

Appendix E

Schemas and Example Records

E.1 Duplicate-charge lineage

The normative example follows Interlude I. Values below are schematic; production encodings use full digests, immutable occurrence tokens, and signed cores.

```
POP {
  event: { id:"e0", u:"u0", source:"agent:priya",
           content_commit:"h0", payload_handle:"p0",
           observed_at:"2026-09-01T14:02:07Z" },
  dedup:false
}
REFUSAL {
  subject:"e0", disposition:"REPAIR", reason:"schema",
  repair:{ required:"original_transaction_id" },
  evidence_commit:"pi0", terminal:false
}
REOPEN {
  event:{ id:"e1", u:"u1", parent:"e0", creation_rank:1,
          content_commit:"h1", payload_handle:"p1" }
}
BINDING {
  subject:"e1", nu_b:"corrections-v4",
  obligations:["amount_matches","second_approval","origin_reference"]
}
CANDIDATE {
```

```
  output_commit:"x0", binding_commit:"b0",
  history_head:"H17", nu_k:"adjustment-kernel-v2"
}
VERIFICATION {
  candidate_commit:"x0", binding_commit:"b0", verdict:"FAIL",
  discharge:[["amount_matches","SAT","REQUIRED"],
             ["second_approval","UNSAT","REQUIRED"],
             ["origin_reference","SAT","REQUIRED"]],
  r_block:["second_approval"], r_waived:[]
}
```

A second descendant e_2 supplies approval, is rebound and transformed again, and receives PASS with an empty blocking residual. The new candidate may share the same output value y while remaining a distinct candidate record because its binding and history commitments differ.

```
COMMIT_CORE {
  id_c:"c0", prior_head:"H17", candidate_commit:"x1",
  verification_commit:"v1", sigma_c:"corrections-v4",
  nonce:"n0", committed_at:"...", authority:"manager:osei",
  r_waived:[]
}
SIGNED_COMMIT { object:"c0", signer:"manager:osei",
                algorithm:"ed25519", signature:"..." }
```

Every example record is accompanied in the machine-readable distribution by its canonical bytes, object identifier, signing preimage, signature, and expected successor head. The prose form above is explanatory and not itself the canonical serialization.

Appendix F

Failure and Recovery Matrix

F.1 Before external actuation

Before a transition record is durable, recovery treats the transition as not having become visible. It may retry the operation as a new attempt, but it must not claim that the earlier decision occurred. After a complete record is durable but before canonical application, recovery applies that record idempotently. It does not rerun POP authentication, REFUSE policy judgment, or COLLAPSE arbitration against the current world.

For BIND or TRANSFORM, recomputation is permitted only when the corresponding determinism and historical-input hypotheses hold. VERIFY is replayed from its signed durable result unless a separate verifier replay theorem applies. A partial or unauthenticated record is rejected or completed from its write-ahead transaction; it is never guessed into a valid transition.

F.2 Collapse publication

If a collapse transaction is absent, form a fresh proposal against the current head. If the transaction is durable and neither H nor L reflects it, apply both. If exactly one side reflects it, apply only the missing side. If both do, mark the transaction complete. Every application is keyed by the transaction identifier, preventing duplicate commits or witnesses.

F.3 External-effect states

Before PREPARE is durable, no external request may have been sent. A durable intent with no attempt witness is pending first transmission. A durable pre-transmission attempt witness with no conclusive receipt is ambiguous: recovery queries the external system when possible and retries only under the declared duplicate-handling contract. A conclusive receipt not yet finalized causes deterministic FINALIZE and commit publication. Conflicting or duplicate receipts enter reconciliation; a shared key proves common intent, not that the external system produced only one effect.

The only stages at which an irreversible effect may already have occurred are ACTUATE and later. Undeclared side effects during TRANSFORM or VERIFY are violations of confinement and require their own domain-specific reconciliation.

Appendix G

Security Claims and Non-Claims

G.1 Adversarial components

Against a Byzantine source, the architecture guarantees only that declared gates and obligations are applied; it does not infer malicious intent from a policy-compliant request. Kernel and verifier identities make artifact claims checkable, but do not prove deployment loaded those artifacts without attestation. History uniqueness holds only under the linearizable arbiter premise. Clock-relative freshness holds only to the degree the trusted clock and registries are honest.

Against storage alteration, an authentic later anchor makes rewriting detectable under the hash assumptions. This is detection, not availability, truth, or non-equivocation. Against ordinary crashes, checkpoint and record replay restore canonical internal state under Chapter 23's six hypotheses. External exactly-once execution remains a non-claim.

G.2 General non-claims

The architecture does not guarantee truth, benevolence, complete evidence, honest policy, ethical retention, infallible implementation, global ledger non-equivocation, or adequacy of a physical or institutional model. PASS is relative warrant. A redaction authorization proves authorization occurred, not that it was justified. Multiple verifiers provide no multiplicative assurance against a shared fault without an independence argument. A compensating act does not reverse the original effect.

Appendix H

Research Programme

H.1 Formal and distributed extensions

Open work includes a verifier-specific replay theorem; mechanized refinement from the reference service to Appendix B; positive quantitative models of verifier diversity; causal partial-order history safety; cross-shard atomic commitment; cross-ledger commitments; and compositional obligation systems. Ledger non-equivocation requires an extension of the authoritative-head mechanism from H to L.

H.2 Privacy, policy, and institutions

Privacy-preserving refusal proofs, completeness proofs for selective ledger projections, formally governed policy evolution, metadata-minimizing retention, and empirical methods for detecting institutional capture remain open. Terminal refusal is represented here as an orthogonal terminal flag, but a richer algebra of appeal, expiry, jurisdiction, and exceptional reopening may be preferable.

H.3 Integration hypotheses

An RSVP–Spherepop bridge still needs explicit continuous-to-event observation maps and consistency laws. KES's per-attempt contraction proposal requires an independent formalization. Distinction holonomy needs transport operators, closed-loop structure, and invariants beyond a residue distance. MEM|8 needs a technical retrieval model proving that admissibility constrains indexing and scoring before secret material can influence observable rankings.

These are not all "extensions rather than gaps." Some, such as ledger non-equivocation, verifier replay, and full implementation refinement, are unclosed assurance obligations for stronger deployments. The specified core remains coherent within its stated scope precisely because those stronger claims are withheld.

Bibliography

Distributed Systems and Consensus

[1] Seth Gilbert and Nancy Lynch. “Brewer’s Conjecture and the Feasibility of Consistent, Available, Partition-Tolerant Web Services.” *ACM SIGACT News* 33, no. 2 (2002): 51–59. Cited in Chapter 21 for the CAP theorem, with the explicit caution that this book’s safety-versus-liveness tradeoff resembles that result in shape without having been shown to reduce to it.

[2] Leslie Lamport, Robert Shostak, and Marshall Pease. “The Byzantine Generals Problem.” *ACM Transactions on Programming Languages and Systems* 4, no. 3 (1982): 382–401. Cited in Chapter 22 as the source of the Byzantine adversary model adopted by the threat analysis.

[3] Leslie Lamport. “The Part-Time Parliament.” *ACM Transactions on Computer Systems* 16, no. 2 (1998): 133–169. Cited in Chapter 16 as the original Paxos consensus protocol and as a representative mechanism capable of realizing hypothesis (H2).

[4] Diego Ongaro and John Ousterhout. “In Search of an Understandable Consensus Algorithm.” In *Proceedings of the 2014 USENIX Annual Technical Conference*, 305–319, 2014. Cited alongside Paxos in Chapter 16 as an alternative infrastructure capable of satisfying (H2).

Cryptography

[5] Marc Stevens, Elie Bursztein, Pierre Karpman, Ange Albertini, and Yarik Markov. "The First Collision for Full SHA-1." In *Advances in Cryptology: CRYPTO 2017*, Lecture Notes in Computer Science 10401, 570–596, 2017. The SHAttered result is cited in Chapter 29 as historical confirmation that collision resistance is an imported and revisable assumption rather than a merely theoretical caveat.

[6] Jonathan Katz and Yehuda Lindell. *Introduction to Modern Cryptography*. 3rd ed. Boca Raton, FL: Chapman and Hall/CRC, 2020. Cited where the book invokes standard cryptographic assumptions, including collision resistance and existential unforgeability under chosen-message attack, without reproving them.

This Author's Related Work

The following unpublished manuscripts are cited in Part VII as sources of its integration hypotheses. None is re-derived or independently validated here. Each citation marks the boundary between what this book establishes and what it borrows from elsewhere in the author's corpus.

[7] Flyxion. "Distinction Holonomy." Unpublished manuscript. Source of the term and general framing adopted, at a deliberately reduced level of formal commitment, in Chapter 31.

[8] Flyxion. "Against the Wave Machine." Unpublished manuscript. Source of the MEM|8 resonant-recall proposal discussed as a candidate model, rather than a validated mechanism, in Chapter 32.

[9] Flyxion. "MEM|8 and the Possibility of a Conscious Gaia." Unpublished manuscript. A more speculative extension of MEM|8 that is not relied upon by this book and is listed so that readers

do not conflate it with the immediate source of Chapter 32's recall model.

[10] Flyxion. Unpublished manuscripts on the Relativistic Scalar-Vector Plenum (RSVP) and Kinetic-Event Synthesis (KES). Sources of the bridge principle discussed in Chapter 30, including the standing caution that KES's detailed mathematics remains a provisional reconstruction rather than settled content.

[11] Flyxion. *Spherepop Specification and Reference Implementation Materials*. Multiple unpublished manuscripts and a Rust reference kernel. These materials constitute the broader research programme from which the book's six-transition pipeline is drawn. This book is a focused elaboration of one part of that programme, not a replacement for it.

A Note on Citation Practice in This Book

Most of the book's argumentative weight rests on results proved within its own chapters and cross-referenced internally. External citations appear where the book imports a genuinely external result, a cryptographic assumption, a named consensus protocol, a documented historical event, or separately developed work whose claims are not rechecked here. A reader examining a particular claim should therefore expect an internal chapter reference in most cases and an entry from this bibliography only at the explicitly cited boundaries.

Drafting Sequence

The recommended drafting order follows dependency rather than final reading order. First write Chapters 1–4 to stabilize the distinctions and vocabulary. Next write Chapters 5–9 to define the event types and obligation system. Then write Chapters 10–19 to complete execution, verification, collapse, and the witness plane. Draft the reference data model and operational-semantics appendix immediately afterward so that informal drift becomes visible. Only then write the implementation, evaluation, and integration parts. The ethical and institutional chapters should be written last, once the technical limits of the analogy are explicit.

Candidate Alternative Titles

Witness Before Commitment
Event Memory and the Architecture of Warrant

Execution Is Not Admission
A Calculus of Refusal, Verification, and Irreversible Action

The Admissible Event
History, Constraint, and Commitment in Spherepop

Memory Without Permission
Refusal Ledgers and the Boundary Between Evidence and Action

Collapse Under Warrant
The Spherepop Architecture of Accountable Commitment