

Proof-Driven Software Delivery with Allium and Bend

Norman Khine

Public edition: 19 October 2026
Adapted research paper

Compositional assurance for multi-repository software systems

Abstract

A distributed software system cannot be justified by a proof or test result from one repository. Observable behaviour emerges from user interfaces, identity and authorization, command handlers, stateful adapters, asynchronous publication, projections and infrastructure. This paper develops a layered assurance architecture for changes made by developers or AI agents. Allium records approved behavioural intent; a typed intermediate representation makes selected semantics explicit; Bend supplies machine-checked obligations for a formal core; and assume-guarantee reasoning relates component contracts across repository boundaries. Evidence is assembled into an immutable system manifest.

Protocol-aware testing exercises implementations under adversarial traces. A controlled simulation harness supports explicit scheduling and fault assumptions, replay and trace reduction. Integration tests cover real dependencies, while operational observation checks environmental premises. Financial examples illustrate transfers, reconciliation, foreign exchange, append-only corrections and dual approval; user-interface examples examine HTMX, Elm and Datastar ownership. A delivery-pipeline design separates untrusted candidate execution from trusted verification and evidence signing.

The claim is deliberately limited: proof applies to a formal object under declared assumptions. It reaches production code only through an established implementation or translation relation. Sampled tests, protocol evidence and operational observations remain distinct. This public edition presents an architecture, a small inspectable arithmetic example and an evaluation programme. It does not claim whole-system verification or include private deployment measurements.

An assurance-quality extension distinguishes policy-weighted usable evidence from entropy, defect probability and mathematical proof. Fixed obligation inventories, mutation controls, non-offsettable losses and explicit recovery evidence prevent aggregate scores from hiding critical degradation.

Keywords: behavioural specification; Allium v3; Bend 2; refinement; assume-guarantee; proof-carrying artifacts; PADST; deterministic simulation; TigerBeetle; reconciliation; HTMX; Elm; Datastar; CI provenance; recoverability; mutation adequacy; assurance quality.

Central objective. *Implementation* $\sqsubseteq$ *Specification*: every observable implementation trace must be permitted by the approved behavioural specification.

Contents

1 Research scope and provenance	3	11.1 Timeouts, retries and pending funds	22
1.1 Method and version discipline	3	11.2 Pending and posted are separate ledger states	24
1.2 Architectural roles and evidence scope	3		
1.3 Reading map	3	12 Money example II: reconciliation	24
1.4 Corpus baselines and language migration	3	12.1 Completeness needs an immutable cut	24
2 Five distinct assurance classes	3	12.2 Reconciliation lifecycle and faults	25
2.1 Scope is part of the result	4	12.3 Reconcile all four counters without spending pending credits	26
3 Formal objective and trace semantics	4	12.4 Sealed posted positions and live counters	27
3.1 What a proof must discharge	4	13 Money example III: foreign exchange	27
3.2 Linearizability and financial identity	5	13.1 Quote binding, rounding and retries	28
3.3 Non-vacuity	5	13.2 Fee and expiry semantics must be explicit	28
3.4 A constructed observation function	5	14 Money example IV: corrections	29
4 Assume-guarantee composition	5	15 UI observation protocols	30
4.1 A non-circular composition rule	6	15.1 HTMX: request, target and swap	30
5 A realistic repository decomposition	6	15.2 Elm islands: a typed local core	30
5.1 The dependency and proof graph	7	15.3 Datastar: versioned streams	31
5.2 Compatibility is behavioural	7	15.4 UI boundaries and adversarial delivery	32
5.3 Shared constraints and graph cycles	8	16 Authorization and dual approval	33
6 From Allium v3 to Bend 2	8	17 Cross-repository GitHub Actions design	33
6.1 Deterministic behavioural IR	8	17.1 Component PR: untrusted candidate, trusted obligations	33
6.2 Bend's contribution and its limits	9	17.2 System verification and release	34
6.3 Runnable Allium v3 and Bend 2 example	9	17.3 Credentials, triggers and attestations	35
6.4 Obligation coverage and the retry distinction	12	18 Governance of the judging criteria	36
6.5 A bounded finite external-event profile	12	19 Proven, tested, assumed and observed	36
7 Proof-carrying component artifacts	13	20 Software entropy and exergy: controlling uncertainty and preserving useful assurance	37
7.1 Freeze intent; then bind the candidate	13	20.1 Origins, scope and corrections to the analogy	37
7.2 The deployable unit is a locked system manifest	14	20.2 Entropy is uncertainty, not incorrectness	38
8 Change-impact analysis	14	20.3 Mutation adequacy with a protected denominator	38
8.1 Selective recomputation	15	20.4 A policy score that cannot compensate away critical losses	38
8.2 Closure cost and reuse limits	15	20.5 Recoverability, safety and conditional progress	39
9 PADST: testing protocol meaning	16	20.6 Financial and UI examples	39
9.1 Adversarial trace obligations	17	20.7 Artifacts, cross-repository gates and replication	40
9.2 A concrete scenario	17	20.8 Threats to validity and adoption limits	41
9.3 CDC and persistence boundaries	18	21 Limitations and research evaluation	41
9.4 A newer version is not always a complete prefix	18	21.1 Threats to validity	41
10 Simulation harness: controlled execution	18	21.2 Public evaluation programme	41
10.1 Distinguish targets from implemented coverage	19	22 Adoption and conclusion	43
10.2 Execution, replay and reduction	19	22.1 A layered assurance system	43
10.3 Minimize without changing the failure	19	Appendix: Complete runnable model and proofs	44
10.4 The empirical assurance boundary	19		
10.5 Conditional liveness and a progress certificate	21		
11 Money example I: a basic transfer	21		

1 Research scope and provenance

1.1 Method and version discipline

This public edition presents a generic architecture for proof-driven delivery across multiple repositories. It retains the original author's technical thesis, mathematical distinctions and worked examples, while replacing organization-specific infrastructure and evidence with explicit architectural roles. Public language and platform references are dated research sources; implementations must pin the exact versions they use.

The examples use Allium v3 and Bend 2. Historical v2 references provide migration context. Language versions must not be silently mixed. [1–3]

1.2 Architectural roles and evidence scope

In this paper, **PADST** means protocol-aware distributed systems testing, and the **simulation harness** is a proposed controlled execution environment. These are capability descriptions, not claims that any named product already provides the complete design. A real handler behind a simulated dependency, a bounded model proof and a production observation are different evidence classes.

Financial workflows make identity, conservation and recovery requirements concrete, but the delivery architecture also applies to other distributed software systems. The arithmetic example sources are included for inspection and independent checking. Private repository inventories, operational observations and implementation-specific reproduction instructions are not part of this public edition. Consequently no such results are attributed to Abakuc or presented as publicly reproducible evidence.

1.3 Reading map

Table 1: Research scope and provenance

Part	Purpose
Sections 2–6	Assurance classes, mathematics, component boundaries and the proposed Allium/Bend bridge.
Sections 7–10	Evidence artifacts, locked manifests, impact analysis, PADST and simulation harness.
Sections 11–16	Transfers, reconciliation, FX, corrections, UI and authorization examples.
Sections 17–21	CI, governance, assurance inventory, threats, evaluation and adoption.

All diagrams describe proposed architecture. The appendix contains a small self-contained arithmetic model; the other configuration and UI listings are illustrative. The local example-check report states exactly what was rerun for this edition.

1.4 Corpus baselines and language migration

A corpus audit should pin the source inventory and parser/checker versions, then report parse diagnostics, checker errors and warnings separately. A parser may recover from errors while returning a zero process exit status, so successful process termination alone is not sufficient evidence of valid source.

Count authored rules and invariant blocks separately from validated obligations. Record explicit language versions, keep new pilots out of a frozen baseline denominator, and reject unsupported or invalid source before proof generation. Do not weaken failing specifications to manufacture coverage. Publish the permitted corpus and its digests when reporting empirical results.

2 Five distinct assurance classes

An evidence bundle must state what its result means. A green indicator without a subject, scope and assumption set is not a useful assurance claim. The five classes below are separate dimensions of evidence, not scores that can be added together or an automatic ladder.

Table 2: Assurance classification required on every evidence record.

Assurance class	Defensible claim	What it does not establish
Proven formal model	A checked theorem holds for all behaviours represented by the specified model and assumptions.	That an independently written implementation follows the model.
Verified implementation or verified translation	An explicit checked relation connects implementation semantics, or a semantics-preserving translation, to the specification.	Unmodeled runtime, compiler, hardware or external-service behaviour.
Tested implementation conformance	Production code or a stated adapter agrees with the model for the explored inputs and traces.	Universal equivalence or refinement outside that exploration.
Protocol-aware system evidence	A named composition obeyed protocol expectations under a recorded scenario and fault set.	Every possible distributed execution or untested component.
Observed production behaviour	Measurements and reconciliation supported specified claims in the observed interval.	Future behaviour, unobserved faults, or unconditional availability.

2.1 Scope is part of the result

A receipt must identify the model, implementation, observation function, environment, exploration bounds, and exact candidate artifacts. “Model proven; adapter tested” is a valid composite statement. “System proven” is not a valid abbreviation for it. Missing evidence is unknown, not success. A timeout means the requested result was not obtained; it does not refute the theorem and cannot satisfy a promotion obligation.

A deterministic generator is useful because identical approved inputs produce identical outputs and reviewable diffs. Determinism alone says nothing about semantic correctness: a consistently wrong translator is still deterministic. Verified translation requires a preservation theorem for the language subset, source and target semantics, and compiler passes in the stated trust boundary.

Policy consequence. Each obligation declares required assurance classes. A weaker class may accompany a result but must not silently replace a stronger required class. An approved degraded manual release remains explicitly degraded.

3 Formal objective and trace semantics

Model component behaviour as $M = (S, S_0, \text{Events}, \delta, O)$, with state space S , initial states S_0 , transition relation δ and observation function O . Internal implementation actions may be hidden or represented as stuttering. The specification defines a language of permitted observable traces; financial effects, authorization outcomes and user-visible commitment claims must not be hidden by O . Refinement mappings provide a disciplined way to relate concrete state and steps to an abstract machine. [4,5]

$$\begin{aligned} \text{Implementation} &\sqsubseteq \text{Specification} \\ \text{Traces}(O(\text{Implementation})) &\subseteq \text{AllowedTraces}(\text{Specification}) \end{aligned} \tag{1}$$

3.1 What a proof must discharge

An invariant proof establishes its base case and preservation for every permitted modeled transition. A refinement argument additionally relates initial states, each concrete step to an allowed abstract step or stutter, and the observation functions. Infinite-trace liveness needs its own argument; a finite collection of successful executions cannot establish eventual completion for all executions.

$$I(s_0) \wedge \forall s, e, s'. [I(s) \wedge \delta(s, e, s') \Rightarrow I(s')] \tag{2}$$

For an intentionally changed business contract, refinement to the old specification may be inappropriate. The law delta must first be approved; the candidate is then judged against the new pinned contract and its compatibility

obligations. Preconditions, units, observation projections and assumptions are part of that semantic delta, even if the JSON schema did not change.

3.2 Linearizability and financial identity

Linearizability requires completed concurrent operations to admit an equivalent sequential history that respects real-time precedence, with each operation taking effect between invocation and response. It is a property of an object's specified operations, not a blanket property of a microservice estate. A timed-out operation may have taken effect; an admissible completion of its pending history must be considered. [6]

For a financial command, define the abstract effect, its stable business identity and the authority that linearizes it. An HTTP acknowledgement, queue delivery or DOM update is not automatically the linearization point of ledger commitment. The adapter must relate a durable ledger result to the domain observation and distinguish Submitted, OutcomeUnknown and Committed.

3.3 Non-vacuity

Prove or demonstrate feasible preconditions and exercise positive witnesses. A model that rejects every transfer may satisfy “no unauthorized transfer” while failing the business journey. Protect observation functions and require completeness checks on traces: an empty observation stream must not make a system with missing telemetry appear conformant.

3.4 A constructed observation function

For the arithmetic example, define $O_m(s) = (s.source, s.destination, e(s))$, with $e = 1$ in Committed and $e = 0$ in Pending or Rejected. The executable demonstration returns (25, 125, 1). This projection distinguishes a financial effect from rejection, but only represents one operation and cannot establish a real ledger history.

The composed reference experiment therefore observes (a, l, p, b_s, b_d, J) : authorization state, ledger-control state, visible projection, both account balances and the complete ordered financial journal. Journal records retain operation and transfer identity, amount and asset; order and multiplicity are preserved. Financial observation in the idempotency equation means (b_s, b_d, J) , not merely the net balance. Expected records are independently derived from source transitions, while actual records are read from the participants' stored state. Replies are compared separately on every step.

The observation footprint is executable: a duplicated posting changes J ; a debit followed by an offsetting reversal also changes J despite unchanged net balance; either individual account perturbation changes the observation. These negative controls guard against accidental erasure. They do not prove completeness of the chosen business vocabulary, and the reference participants are not production adapters.

4 Assume-guarantee composition

For component i , A_i states assumptions about its inputs and environment; G_i states its guarantees; O_i defines externally visible behaviour. A local argument establishes $A_i \Rightarrow G_i$ under that component's semantics. For every connected pair i, j , define C_{ij} : the producer's observable guarantees, transport semantics and adapter mapping satisfy the consumer's assumptions. [7]

$$\left(\bigwedge_i G_i \right) \wedge \left(\bigwedge_{e \in H} C_e \right) \wedge \left(\bigwedge_{K \in \mathcal{K}} J_K \right) \wedge E \Rightarrow P_{\text{system}} \quad (3)$$

This is the desired system obligation, not a theorem obtained by merely listing green repositories. Each premise must be discharged or explicitly retained as an environmental assumption. In cyclic systems, naïve circular reasoning is invalid: use a sound compositional rule, suitable induction or progress argument, and justified fairness conditions. Pairwise compatibility alone may miss a joint invariant such as two consumers spending the same shared resource. [7]

Table 3: Assume-guarantee composition

Environmental assumption E	Required treatment
Identity provider; cryptographic keys	Issuer/audience, key trust and rotation, revocation policy, freshness and provenance.
Network delivery; queues	Loss, retry, redelivery, ordering scope, partitions and bounded/unbounded delay stated separately.
Clocks; recovery	Clock domains, skew bounds, leases, availability and recovery assumptions; no unconditional liveness.
TigerBeetle	Pinned cluster/client semantics, account constraints, ledger mapping and failure model.
Amazon MQ / RabbitMQ	Chosen protocol and managed-service configuration; acknowledgement and redelivery assumptions.
DynamoDB	Exact conditional-write/transaction boundary; no implied atomicity with another database.
Browser; deployment controller	Supported browser/runtime versions, event ordering, DOM ownership, and digest-enforcing rollout policy.

Producer output schemas must be compatible, but schemas are only one part of C_{ij} . A consumer expecting committed events is incompatible with a producer that now emits acceptance events under the same field names. A version, unit, tenant scope, ordering or idempotency change can invalidate an edge without changing its transport encoding.

4.1 A non-circular composition rule

Here H is the set of compatibility hyperedges, including ordinary producer–consumer pairs and joint shared-resource obligations. Partition the dependency graph into strongly connected components $\mathcal{K}$. Verify each cyclic component as one joint product transition system: establish its initial invariant and preservation under every internal step and every permitted incoming action. J_K records these joint obligations. Internal component guarantees cannot be assumed to establish themselves. The condensation graph is acyclic; discharge incoming assumptions in its topological order from already established predecessor guarantees and explicit environmental premises. Finally prove the displayed entailment to the system property. Acyclic singleton components use the ordinary local rule.

This rule is deliberately conservative: a large cycle may require a large product proof, or a reviewed coarser abstraction with a refinement argument. A cyclic drawing is not a circular proof. Pairwise schemas do not discharge resource hyperedges. The runnable control example includes a counterexample with only 100 source cents: two independently approved 100-cent postings satisfy every local finite-control invariant but produce a source balance of -100 . Here -100 is a derived spend position, never a negative TigerBeetle counter. The joint liquidity invariant rejects that trace. The main experiment instead declares 250 source cents and two possible postings; that fixed bound must not be generalized to arbitrary liquidity.

5 A realistic repository decomposition

The repository is a delivery boundary; the semantic component is a proof boundary. One repository may export several contracts, and a journey may cross many repositories. The following decomposition is illustrative, using generic capability names—not a claim that each row is already a separately verified product.

Table 4: A realistic repository decomposition

Repository / component	Exported behavioural surface
UI shell	Navigation, region ownership, aggregate identity and render-version rules.
Elm island package	Model/Msg reducer, view projection, typed port envelopes.
HTMX / Datastar UI server	Command-to-fragment/patch semantics; selectors, versions and stream identity.
API / command handling	Validated requests, stable operation IDs, accepted/rejected outcomes.
Authentication / authorization	Identity evidence, eligibility, backend decision freshness.

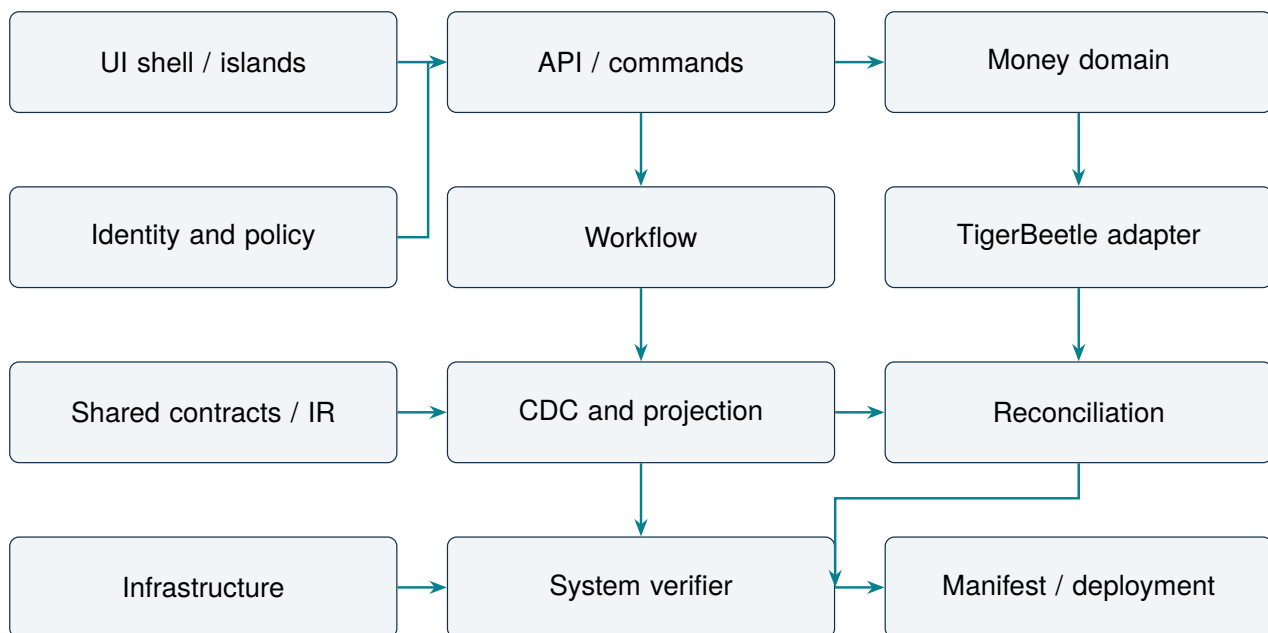
Table 4: A realistic repository decomposition

Repository / component	Exported behavioural surface
Cedar / policy definitions	Permit/forbid semantics, schema and policy-version binding.
Workflow / orchestration	Saga transitions, retries, timeouts and compensation permissions.
Money-movement domain	Amount/asset constraints, transfer and correction transition laws.
TigerBeetle adapter	Account mapping, stable transfer IDs, linked/pending request encoding.
CDC / event publication	Event identity, durable checkpointing, replay and ordering scope.
Projection / read model	Deduplication, monotonic versions, rebuild and query semantics.
Reconciliation	Snapshot boundaries, exact matching, exception and escalation closure.
Infrastructure	Queue/storage/identity configuration, deployment and recovery assumptions.
Shared contracts / schemas	Allium authority, typed IR, codecs, vocabulary and compatibility policy.
System verification	Locked manifests, compatibility closure, PADST scenarios and simulation harness runs.

A specification repository owns approved intent and governance artifacts. A delivery controller owns workflow execution, independent obligation resolution and evidence gates. Shared libraries can host reusable semantic/protocol machinery; the system-verification repository composes the result. Ownership must be explicit for interfaces that span these roles.

5.1 The dependency and proof graph

Figure 1 maps selected behavioural obligations across the repositories and into system verification.



Selected behavioural obligations; each edge requires compatibility evidence.

Figure 1: Selected cross-repository edges. Every edge carries a versioned compatibility obligation; shared contracts and infrastructure are dependencies of the whole composition.

5.2 Compatibility is behavioural

Consider a command path $UI \rightarrow API \rightarrow policy \rightarrow money\ domain \rightarrow ledger\ adapter$. The UI supplies a stable operation identity, but the API authenticates and validates it. Policy guarantees an authorization decision bound to the same command hash and policy version. The domain assumes that binding when constructing its transition.

The adapter guarantees that retries encode the same TigerBeetle transfer identity and do not substitute a new payload.

On the return path, a CDC event must denote the same aggregate and operation, a projection must preserve versions, and a fragment or patch must not advertise a stronger state than its durable observation. A contract edge therefore contains identity mapping, units, cardinality, rejection behaviour, ordering scope, and an observation mapping—not just a schema reference.

5.3 Shared constraints and graph cycles

Use hyperedges when an obligation depends on several producers, such as an FX quote plus both liquidity accounts and an authorization receipt. Condense strongly connected components for impact planning; do not mistake that graph operation for a proof that cyclic assumptions hold. Joint conservation and approval-before-execution laws live in the system layer even when all local schemas validate.

Contract metadata must identify dependencies on policy, account mapping, numeric scales, observation functions and infrastructure. Missing edges are assurance defects: a supposedly small change cannot be declared independent because the graph forgot its consumers.

6 From Allium v3 to Bend 2

Allium v3 describes entities and rules using triggers, preconditions and postconditions, with separate non-normative guidance. Its reference distinguishes rejection of an external stimulus from non-firing of a temporal trigger and termination of a chained trigger. A translator must preserve those distinctions rather than imposing an invented global rollback rule. [3]

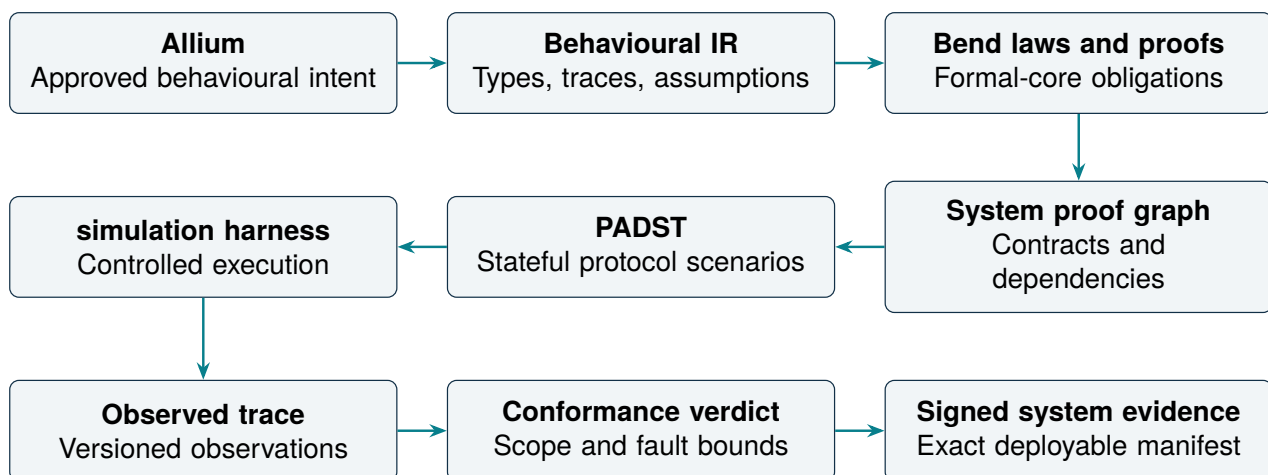


Figure 2: Proposed assurance pipeline. The proof branch and empirical trace branch meet in a signed manifest-specific result; neither silently substitutes for the other.

6.1 Deterministic behavioural IR

The proposed translator emits typed state and event vocabularies, transition relations, preconditions, postconditions, invariants, permitted traces, and observation functions. Every obligation retains its source rule ID and span, normalized specification hash, transitive imports, assumptions, numeric units, asset scales, overflow and rounding rules, and temporal interpretation. Canonical serialization and pinned generator inputs make the IR reproducible.

An unsupported or ambiguous expression is an explicit unresolved obligation. Undefined predicates, unresolved domain questions, unbounded temporal requirements and implicit unit conversions must not become assumed truths. A language-version mismatch fails before proof generation. The examples now explicitly select v3; the previous v2 edition is not evidence for a v3 corpus. Corpus conformance must be checked before generation. [2]

6.2 Bend's contribution and its limits

Bend 2 separates laws from their proofs through LAWS.bend and PROOF.bend; its dependent type checker checks proof terms. Termination restrictions support the proof discipline, while @unsafe permits behaviour outside the associated guarantees. Foreign code, axioms and dependencies require an explicit inventory. [8]

Upstream states that the compiler is not fully audited and that its Lean formalization and implementation differ. A pinned checker reduces drift; it does not eliminate the trusted computing base. The proposed bridge must not advertise audited end-to-end soundness that its toolchain has not established. [9]

6.3 Runnable Allium v3 and Bend 2 example

The following example is supplied for checking with Allium CLI 3.0.3 (language version 3) and Bend 2.0.16. It models one isolated USD operation, in integer cents, using a terminal result to make retries idempotent. The concrete fixtures start with 125 cents in the source and 25 cents in the destination; a 100-cent command yields 25 and 125 cents, including after a retry. These are reproducible test data, not production customer transactions. The Allium checker release supports language versions 1, 2 and 3; the file explicitly selects version 3. [10]

What the checks mean. Allium validates the specification's syntax and static references; it does not execute these rules or prove their invariants. Bend checks six universally quantified laws of the supplied natural-number model: reservation conservation, transfer conservation, duplicate-submit idempotency, retry state preservation, the terminal-state retry guard, and observation stability. Concrete fixtures and deliberately invalid variants can exercise the checkers. The build report for this edition names the checks actually executed. The complete model and proof terms are printed in the appendix ; editable source files are maintained separately in the research repository.

Listing 1: Complete Allium v3 specification, accepted by Allium CLI 3.0.3. Its invariants are declared intent, not theorems proved by Allium.

```
-- allium: 3

enum Phase { pending | committed | rejected }
entity Payment {
  operation_id: String
  asset: String
  amount: Integer
  initial_source: Integer
  initial_destination: Integer
  source: Integer
  destination: Integer
  phase: Phase
}
given { payment: Payment }

rule CommitPayment {
  when: SubmitFunded(operation_id, amount, asset)
  requires: operation_id = payment.operation_id
  requires: amount = payment.amount
  requires: asset = payment.asset
  requires: payment.phase = pending
  requires: amount >= 0
  requires: payment.source >= amount
  ensures: payment.source = payment.source - amount
  ensures: payment.destination = payment.destination + amount
  ensures: payment.phase = committed
  ensures: payment.operation_id = payment.operation_id
  ensures: payment.asset = payment.asset
  ensures: payment.amount = payment.amount
  ensures: payment.initial_source = payment.initial_source
  ensures: payment.initial_destination = payment.initial_destination
}

rule RejectInsufficientFunds {
  when: SubmitUnfunded(operation_id, amount, asset)
  requires: operation_id = payment.operation_id
  requires: amount = payment.amount
  requires: asset = payment.asset
  requires: payment.phase = pending
  requires: amount >= 0
  requires: payment.source < amount
  ensures: payment.source = payment.source
  ensures: payment.destination = payment.destination
}
```

```

    ensures: payment.phase = rejected
    ensures: payment.operation_id = payment.operation_id
    ensures: payment.asset = payment.asset
    ensures: payment.amount = payment.amount
    ensures: payment.initial_source = payment.initial_source
    ensures: payment.initial_destination = payment.initial_destination
  }

rule RetryPayment {
  when: Retry(operation_id, amount, asset)
  requires: operation_id = payment.operation_id
  requires: amount = payment.amount
  requires: asset = payment.asset
  requires: payment.phase != pending
  ensures: payment.source = payment.source
  ensures: payment.destination = payment.destination
  ensures: payment.phase = payment.phase
  ensures: payment.operation_id = payment.operation_id
  ensures: payment.asset = payment.asset
  ensures: payment.amount = payment.amount
  ensures: payment.initial_source = payment.initial_source
  ensures: payment.initial_destination = payment.initial_destination
}

invariant Conservation {
  payment.source + payment.destination =
    payment.initial_source + payment.initial_destination
}

invariant NonNegativeBalances {
  payment.source >= 0 and payment.destination >= 0
}

surface PaymentCommands {
  facing item: Payment
  exposes: item.phase
  provides:
    SubmitFunded(operation_id: String, amount: Integer, asset: String)
    SubmitUnfunded(operation_id: String, amount: Integer, asset: String)
    Retry(operation_id: String, amount: Integer, asset: String)
}

```

Listing 2: Actual Bend 2 laws. The paired PROOF.bend discharges all six without unsafe annotations or unresolved proof holes in the example sources.

```

import Base
import ./Payment.bend as Payment
import ./Balance.bend as Balance

law reservation_conserve:
  for amount: Nat
  for source: Nat
  Balance.Conserves(Balance.reserve(amount, source), amount, source)

law duplicate_submit_idempotent:
  for +amount: Nat
  for state: Payment.Payment
  {Payment.apply(amount, Payment.apply(amount, state)) ==
    Payment.apply(amount, state) : Payment.Payment}

law transfer_conserve:
  for +amount: Nat
  for state: Payment.Payment
  {Payment.total(Payment.apply(amount, state)) ==
    Payment.total(state) : Nat}

law retry_preserves_state:
  for state: Payment.Payment
  {Payment.retry_state(Payment.retry(state)) == state : Payment.Payment}

law retry_requires_terminal:
  for state: Payment.Payment
  {Payment.retry_accepted(Payment.retry(state)) == Payment.terminal(state) : Bool}

law observed_retry_stable:
  for +state: Payment.Payment

```

```
{Payment.observe(Payment.retry_state(Payment.retry(state))) ==
  Payment.observe(state) : Payment.Observation}
```

Listing 3: Executable submit-then-retry demonstration, with a structured financial observation.

```
import Base
import ./Payment.bend as Payment
import ./PROOF.bend as Proof

def main() -> Payment.Observation:
  Payment.observe(Payment.retry_state(Payment.retry(
    Payment.apply(100n, Payment.Pending{125n, 25n}))))
```

From the paper directory, use the example-check command below with installed Allium 3.0.3 and Bend 2.0.16 binaries. The command records tool versions, source digests and actual results; no result is inferred from the presence of a proof file.

Listing 4: Check the included arithmetic example with explicit local tool paths.

```
ALLIUM=/path/to/allium BEND=/path/to/bend \
python3 check_examples.py
```

The report is an unsigned local reproducibility record. It is not a deployment attestation, an independent review or a proof about a production adapter.

Boundary. This arithmetic Bend model remains manually written against selected Allium intent. The finite-control profile discussed below is separate and does not cover this Integer/arithmetic specification and is not a verified translator. One operation identity, matching payload, asset and account mapping, non-negative initial balances, and sequential atomic execution are assumptions at this projection boundary. The Allium preconditions state identity/payload matching; Bend’s single-operation state does not implement ID lookup, payload hashing or authorization. Natural numbers avoid fixed-width overflow in the model but do not establish machine-integer or asset-scale correctness in an adapter. Zero-sized commands are allowed by this pedagogic model; it is not an executable TigerBeetle API specification. There is no network, concurrency, persistence or bank integration in this example.

Table 5: From Allium v3 to Bend 2

Implementation connection	Result and remaining obligation
Execute proven Bend core	Bind its source/proof to the compiled artifact and real adapter. Compiler/runtime assumptions remain.
Verified translation to another language	Supply a preservation proof for the actual supported source/target semantics and passes.
Deterministic generation only	Reproducibility, not semantic proof. Add verified passes or explicitly weaker conformance evidence.
Language-specific verification	Prove the production-language component refines the same IR under its memory/effect model.
Differential / property-based comparison	Record generated inputs, states and counterexamples; classify as tested conformance.

The implementation relation must include error paths, serialization and observable effects. A proof of a copied pure function cannot cover a handler that bypasses it, converts amounts differently, or trusts a UI permission signal. Reachability and adapter evidence belong in the binding.

6.4 Obligation coverage and the retry distinction

Table 6: Arithmetic example: source obligations, discharge, and residual binding.

Intent	Model discharge	Remaining connection
Conservation	<code>transfer_conserve</code> s; supporting reservation law	Initial source/destination sum bound to source initial-total fields.
NonNegativeBalances	Every balance is a Nat; preservation follows from the result type	Allium Integer inputs require checked nonnegative conversion; negative and Boolean boundary controls are tests.
CommitPayment / RejectInsufficientFunds	Total reservation branches; transfer conservation	Identity, asset, matching payload and account mapping are outside this projection.
RetryPayment	<code>retry_requires_terminal</code> <code>retry_preserves_state</code>	and Source operation identity and payload guards still require adapter evidence.
Additional algebraic strengthening	<code>duplicate_submit_idempotent</code>	Double application of the reducer is distinct from the source Retry event.
Observation stability	<code>observed_retry_stable</code>	Single-operation observation; no production journal binding is proved.

Typing and explicit proof terms are different discharge techniques within the same formal-model assurance class. A Nat representation does not upgrade an unverified Integer-to-Nat adapter to verified implementation. The four boundary controls test selected invalid inputs; they do not prove that all external adapters enforce this conversion.

Bend's `+amount` annotation permits reusable access to a Data value. It does not mean a positive number and does not narrow Nat: zero remains included. The reservation law needs a single use; laws that reuse an amount bind it with `+`. All six laws quantify over their stated types under the checker assumptions.

6.5 A bounded finite external-event profile

A candidate compiler profile `allium3-finite-external-v1` consumes the official v3 AST, source bytes and an explicit initial valuation. It accepts one owned entity and one binding; enum-valued state and event parameters; equality, inequality, conjunction and disjunction guards; and simultaneous pre-state assignments to every field. Each external event has exactly one rule. Every frame assignment is explicit. It rejects parser error diagnostics even if the parser process exits zero.

The profile rejects imports, Integer arithmetic, temporal triggers, chained triggers, implicit frames, conditional effects and ambiguous dispatch. External precondition failure returns the unchanged state and a rejected reply. This rule is not applied to unsupported trigger kinds. There is no claim to translate the full Allium language or an arbitrary specification corpus.

Bounds are six state fields, eight values per enum, 256 states, 128 parameter valuations per event and 8,192 transition rows; predicate depth is at most six. Recursive predicate processing decreases AST depth. Enumeration visits finite Cartesian products; emitted Bend functions use finite case analysis without recursive transition evaluation. These bounds give a termination argument and explicit rejection before an unbounded expansion, not a proof of semantic preservation.

Every rule, requires clause, ensured field and invariant has an IR coverage entry. Enabled rules and effects require an admissible witness, preventing impossible preconditions from passing silently. Source spans and source hashes retain provenance; the semantic hash excludes source locations but includes initial state and transition meaning. A compiler library does not itself prove that a supplied AST corresponds to supplied bytes: the driver runs the pinned parser and records both.

A validation programme should exercise independent authorization, ledger-control and monotonic-projection examples. An independently implemented source interpreter should construct expected transitions directly from the parsed source. Generated step and accepted-reply results should be compared against those expectations for every row in the declared finite input universe.

Publish the number of states, parameter valuations, transition rows, generated obligations, unsupported constructs and rejected mutants for each example. Repeated generation should produce identical bytes under

pinned inputs. This establishes evidence about those finite instances; it does not establish a general compiler-correctness theorem.

Maintain one canonical owner for compiler code and one for approved specifications. A public replication archive must include the permitted source, tool identities and actual output before its measurements can be presented as independently reproducible results. This edition does not distribute an organization-specific compiler or its private corpus.

7 Proof-carrying component artifacts

Every component repository publishes immutable evidence tied to its source and built subject. This illustrative record satisfies the basic interchange requirements; a production schema must additionally encode assurance class, applicability, observation hash, trust closure, timestamps and signer identity. A signature authenticates an assertion's origin and subject—it does not make an incorrect assertion true. [11,12]

Listing 5: Illustrative evidence schema; placeholders are not real verification results.

```
component: ledger-adapter
source_sha: <full-commit-sha>
artifact_digest: sha256:...
contract:
  name: money-transfer
  version: 4
contract_ir_hash: sha256:...
assumptions:
  - authorized-command.v2
  - unique-operation-id.v1
  - account-mapping.v3
guarantees:
  - single-ledger-conservation.v3
  - retry-idempotency.v2
  - stable-observation.v1
observations:
  - TransferAccepted
  - TransferRejected
  - TransferCommitted
proof:
  checker: bend
  checker_version: <pinned-version>
  laws_hash: sha256:...
  proof_hash: sha256:...
  result: pass
unsafe_axioms: []
test_evidence:
  padst_trace_digest: sha256:...
  simulation_scenario_digest: sha256:...
```

The system verifier resolves the signed subject digest, checks signer/workflow identity, validates the schema and expiry, compares the declared obligation set with its independently resolved set, and checks the full dependency closure. Non-Bend repositories must declare their actual assurance method or an approved not-applicable scope; they must not manufacture a Bend pass merely to populate this schema.

7.1 Freeze intent; then bind the candidate

Two snapshots prevent a circular lifecycle. An **obligation snapshot** freezes approved intent, assumptions, required assurance classes, bindings, scope and baseline policy before an agent changes code. A **candidate snapshot** records the exact produced source, model, proof, configuration and artifacts after the work. Demanding final proof hashes before writing the code would make the workflow impossible.

Table 7: Proof-carrying component artifacts

Checkpoint	Frozen content	Decision
Before execution	Ratified rules, IR semantics, binding scope, required claim kinds, baseline and policy.	May perform the scoped work; no deployment permission yet.
After build	Candidate source SHAs, artifact digests, proofs, toolchain and dependency closure.	Run checks against this immutable candidate.
Before promotion	All required candidate-bound receipts, controls and independent verdict.	Promote only if the required evidence is complete and current.
Before deployment	Approved system manifest, deployment policy, expiry and current assumptions.	Deploy the named artifacts only; drift blocks or triggers approved recovery.

7.2 The deployable unit is a locked system manifest

Listing 6: Illustrative manifest skeleton, not a deployment-ready configuration.

```

system_manifest:
  baseline_manifest: sha256:...
  components:
    ledger-adapter:
      source_sha: <full-commit-sha>
      artifact_digest: sha256:...
      contract_version: money-transfer.v4
      contract_ir_hash: sha256:...
      proof_hash: sha256:...
    # Every other participating component has an exact entry.
  compatibility_proofs: [sha256:...]
  padst_scenario_set: sha256:...
  padst_evidence: sha256:...
  simulation_scenario_evidence: sha256:...
  infrastructure_version: <immutable-version-and-digest>
  deployment_policy_version: <immutable-version-and-digest>
  verifier_workflow_sha: <full-commit-sha>

```

Hash a canonical manifest representation. References must resolve to immutable content, including browser bundles and schemas. Never prove a composition assembled from moving branch heads. Preserve old receipts as historical facts; invalidate their applicability to a changed candidate rather than rewriting their recorded outcomes.

8 Change-impact analysis

When repository *k* changes, compare its exported semantic contract with the approved baseline. The impact classifier examines internal implementation, exported schemas, behavioural semantics, preconditions, guarantees, observation functions, event ordering, identity/idempotency semantics, units and asset scales, authorization policy, and infrastructure assumptions. Paths and dependency files are inputs, not sufficient semantic evidence.

Table 8: Change-impact invalidation flow: classify → resolve closure → rerun obligations → bind a new manifest.

Change class	Invalidation / required work
Internal code; exported contract unchanged	Re-prove or re-test the implementation binding, build a new artifact, rerun affected local and journey evidence. Compatible downstream code need not be rebuilt indiscriminately.
Schema or behavioural contract changed	Resolve the transitive consumer closure, using both old and new edges so deleted dependencies cannot hide impact.
Precondition strengthened / guarantee weakened	Treat as potentially breaking even with identical schema; re-discharge every affected consumer assumption.

Table 8: Change-impact invalidation flow: classify → resolve closure → rerun obligations → bind a new manifest.

Change class	Invalidation / required work
Observation, ordering, identity or units changed	Invalidate trace projections, idempotency and numeric compatibility obligations, and dependent UI/reconciliation evidence.
Policy or infrastructure assumptions changed	Re-evaluate all claims relying on the assumption, including unchanged application repositories.
Impact cannot be established	Block automatic promotion; require clarification or an explicitly governed degraded path.

8.1 Selective recomputation

Within the affected proof closure, rerun local component proofs, repository-edge compatibility proofs, affected end-to-end trace laws, PADST protocol scenarios, controlled simulations, and relevant browser and ledger integration tests. Cache reuse requires equality of all relevant semantic and trust inputs, not merely a matching contract version string. An unchanged theorem can be reused only with an established binding to the newly built implementation.

The system verifier issues four separate assertions: **component safety** for k ; **compatibility** of the affected edges; **end-to-end trace refinement** for affected journeys at the actual evidence class; and **validity of one exact deployable manifest**. If the implementation connection is only tested, the third assertion is tested trace conformance—not a universal trace-refinement proof.

Review example. Changing a projection’s version comparison from “strictly newer” to “arrival order” affects $CDC \rightarrow \text{projection} \rightarrow SSE \rightarrow UI$ even when every field and endpoint is unchanged.

8.2 Closure cost and reuse limits

Compute the affected closure over both old and new dependency graphs so that deleted edges cannot hide impact. Report local obligations, downstream consumers and mandatory manifest validation separately. Graph size alone does not establish verification savings.

With adjacency lists, reachability and strongly connected component condensation cost $O(|V| + |E|)$. Discharging the resulting obligations may dominate that cost: joint state spaces multiply, theorem checking has its own complexity, and a shared semantic or toolchain change can invalidate a broad closure. Measure those costs on an explicitly published corpus rather than extrapolating from a private pilot.

Separate semantic theorem reuse from artifact binding and receipt reuse. Unchanged semantic inputs may permit reusing a theorem; changed implementation bytes still require binding evidence and new manifest-specific authorization. A signature over the old manifest cannot be reused for a new one. Cache keys include compiler/checker identities, axioms, observation definitions and environmental profiles. Strict reuse can be expensive; partitioning contracts and naming assumptions precisely reduces avoidable invalidation without weakening these conditions.

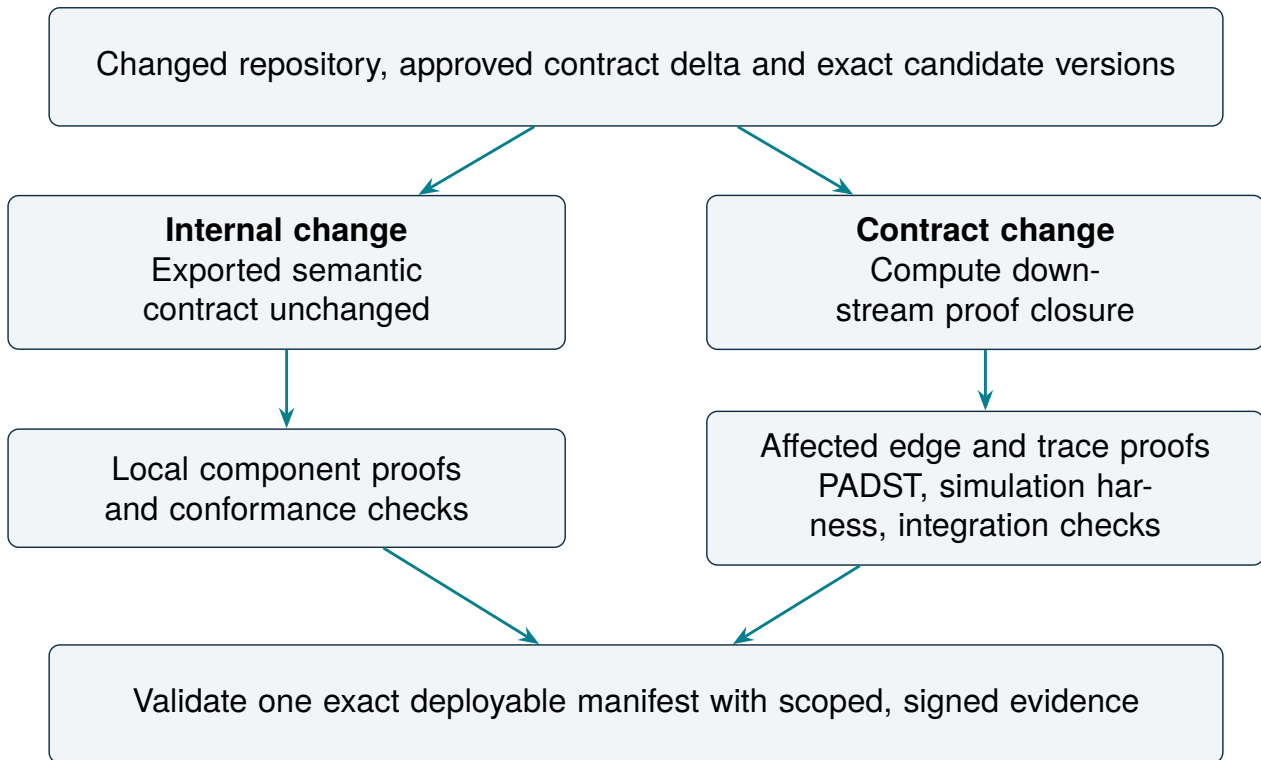


Figure 3: Change-impact invalidation. Semantic contract changes traverse the affected proof graph; internal changes retain local verification and exact-manifest validation.

9 PADST: testing protocol meaning

PADST is the proposed bridge from static obligations to protocols exercised by real distributed components. It operates on typed messages and protocol state, not only arbitrary byte traffic. A generator knows that a post resolves a particular pending transfer, an approval belongs to a particular payment version, or an SSE replay cursor belongs to a particular stream epoch. State-machine testing supplies a useful foundation, but its explored executions remain a sample unless completeness is separately established. [13,14]

$$\begin{aligned} \text{Scenario} = & \text{InitialState} + \text{ProtocolTrace} \\ & + \text{FaultSchedule} + \text{ExpectedObservations} \end{aligned} \quad (4)$$

$$\text{ObservedTrace} \in \text{AllowedTraces}(\text{Specification}) \quad (5)$$

PADST consumes the system proof graph, approved Allium behaviours, generated behavioural IR, state-machine definitions, protocol schemas, component assumptions and guarantees, a candidate manifest, a fault model, and protected observation functions. It first constructs feasible positive journeys, then perturbs identity, order, timing, state and dependencies. Deliberately invalid requests are legitimate inputs: rejection and absence of financial effect are expected observations, not necessarily test failures.

Table 9: PADST: testing protocol meaning

Protocol family	Semantic obligations to encode
HTTP; JSON / Protobuf	Methods, correlation IDs, status versus domain outcome, validation and version negotiation.
OIDC / OAuth; policy	Issuer/audience, token type, expiry, principal, policy version and backend re-evaluation.
HTMX; Elm ports	Request/fragment target, swap lifecycle, island ownership and typed boundary decoding.

Table 9: PADST: testing protocol meaning

Protocol family	Semantic obligations to encode
Datastar SSE	Element/signal patch type, aggregate version, stream identity, framing and reconnect cursor.
AMQP; EventBridge; S3	Acknowledgement, duplication, redelivery, ordering scope, object/version and event identity.
TigerBeetle; CDC	Stable transfer IDs, linked/pending semantics, durable event identity and checkpoint recovery.
DynamoDB; Step Functions	Conditional writes, transaction boundaries, task retries and execution-event correlation.
Deployment manifests	Exact artifact/config versions, rollout phases and compatibility of mixed versions.

Related protocol-oriented work includes state-machine languages and consensus-testing DSLs. They motivate explicit protocol state and controlled adversarial scheduling; they are not citations for a public PADST product or proof that the reference platform's proposed machinery already exists. [14, 15]

9.1 Adversarial trace obligations

Table 10: Minimum scenario catalogue; expand it from actual obligations rather than treating this table as exhaustive.

Scenario family	Generated perturbations	Required observation
Grammar and state	Valid and invalid sequences; malformed messages; version mismatch.	Valid transitions proceed; invalid inputs reject without unauthorized effects.
Delivery	Duplicate, reorder, delay or drop messages; dependency outage.	Declared safety survives; liveness only under the stated recovery assumptions.
Identity	Replay attempts; duplicate operation IDs; different payload under the same ID.	Same logical command retains identity; conflicting identity/payload rejects.
Commit ambiguity	Timeout after commit; retry before commit; retry after commit.	No second financial effect; unknown outcome resolves through the same identity.
Security	Authentication expiry; policy-version change; stale UI permissions.	Backend re-evaluates eligibility and records the policy decision.
Streams	Partial delivery; SSE disconnect/reconnect; duplicate or stale patches.	No premature event application or regression of authoritative visible state.
Projections	CDC consumer restart; replay; stale projection; out-of-order versions.	Durable checkpoint and monotonic state; missing input is observable.
Composition	Concurrent commands; cross-repository contract mismatch; incompatible account mapping.	Compatibility failure is detected before automatic deployment.

9.2 A concrete scenario

Initial state: payment p is authorized, the source has sufficient constrained balance, no transfer with identity t exists, and the projection is at version 11. Send $\text{Submit}(p, t)$, let the ledger commit, drop the response, crash the publisher, retry $\text{Submit}(p, t)$, restart CDC, deliver its event twice, and reconnect the UI stream after rendering version 12. Expected observations include one committed transfer, a stable original result, one effective projection advance, and no return to version 11.

Check the complete observation history against the language, not only the final account balance. A final balance may look correct after an unauthorized transfer followed by a compensating transfer. The prohibited intermediate effect still matters. Preserve the first counterexample, its source observations and the causal relationships needed to reproduce it.

Meaning of a pass. A PADST pass supports conformance for the explored initial states, traces, schedules and fault bounds. It is a universal result only if exhaustive coverage and the correspondence to the implementation have themselves been established.

9.3 CDC and persistence boundaries

TigerBeetle documents CDC publication to AMQP 0.9.1 brokers with at-least-once semantics: recovery can replay a message that was delivered but not acknowledged. Consumers are responsible for idempotent processing. Event bodies retain nanosecond timestamps; the AMQP timestamp header has lower precision. Large integers are encoded as strings for interoperability. These details belong in the protocol contract and decoder tests. [16]

Table 11: PADST: testing protocol meaning

Boundary	Proposed safety rule	Failure to explore
Ledger → CDC	Preserve cluster/event/transfer identity, event kind and source timestamp.	Drop, replay or truncate a committed event; restart the publisher.
Broker → projection	Apply effect and checkpoint atomically where supported, or use a recoverable idempotent protocol.	Crash before/after projection update and before/after acknowledgement.
Projection → read / SSE	Reject version regression; distinguish source progress from display freshness.	Reorder CDC events, reuse a cursor, deliver a stale snapshot.
Numeric decoder	Preserve integer identity and amount exactly across JSON and languages.	Round a 128-bit ID or amount through a JavaScript floating-point number.

RabbitMQ reliability documentation makes acknowledgements and duplicate handling part of application reliability. Configure Amazon MQ/RabbitMQ assumptions explicitly; broker delivery is not end-to-end exactly-once business execution. A producer retry, consumer restart and lost acknowledgement can all expose duplication. [17]

DynamoDB conditional writes and transactions can guard authoritative records within their documented boundary. They do not create an atomic transaction with TigerBeetle. Persist stable business/ledger identities and define recovery at each cross-store step; then exercise those exact boundaries. A read model should not infer that an absent local receipt means the ledger never committed. [18]

TigerBeetle’s own safety/testing practices are valuable dependency evidence, but they do not prove the application’s codecs, mappings, queue configuration or recovery protocol. The candidate manifest and adapter tests must establish that connection separately. [19]

9.4 A newer version is not always a complete prefix

A full version-19 snapshot may supersede version 17 if its observation contract includes the intervening state. An incremental version-19 delta cannot simply skip version 18 unless the update algebra proves that safe. Track gaps separately from monotonicity. Equal versions with conflicting payloads must surface a conflict; a strict numeric comparison alone is insufficient. PADST should vary whether a message is a snapshot or delta and crash between recording the version and applying its content.

$$\begin{aligned} &\text{older version} \Rightarrow \text{no regression} \\ &\text{equal version} + \text{different authoritative payload} \Rightarrow \text{conflict} \end{aligned} \tag{6}$$

10 Simulation harness: controlled execution

simulation harness is the proposed controlled substrate used by PADST, not another unit-test framework. Its target is sufficient control and observation of distributed execution to replay meaningful failures against an exact manifest. Deterministic simulation has strong precedent in FoundationDB; those techniques depend on deliberate control of nondeterminism, not merely starting containers with a seed. [20,21]

Table 12: simulation harness: controlled execution

Required capability	Control and evidence boundary
Hermetic manifest deployment	Resolve exact artifact digests, dependencies and configuration; deny undeclared egress and ambient state.

Table 12: simulation harness: controlled execution

Required capability	Control and evidence boundary
Time and scheduling	Virtual time, bounded clock drift/skew, controllable task scheduling, timers and deterministic randomness.
Network and protocols	Controllable links; delay, loss, duplication, reordering and partial delivery with typed event identities.
Process and storage faults	Crash/restart, disk/persistence faults, dependency outages and recovery ordering.
Identity and policy changes	Credential/token expiry, group changes, key rotation and policy-version changes at chosen steps.
Queue and publication recovery	Queue redelivery, CDC replay, consumer restarts and checkpoint/publish crash boundaries.
Browser and stream lifecycle	Browser reload, island lifecycle, SSE disconnection/reconnection and projection/-patch timing.
Reproduction and diagnosis	Seeded runs, complete traces, state snapshots, record/replay and minimized failing traces.

10.1 Distinguish targets from implemented coverage

A protocol runtime may execute real handlers with simulated storage, queues or ledgers. That is useful evidence but does not mean every production binary and cloud service was exercised. The exact executed paths, substituted adapters and excluded participants belong in the receipt.

Therefore the table above is a target capability contract. Each run must declare which effects were simulated, which production handlers or binaries executed, and which participants were absent. A missing node or uncontrolled effect cannot be hidden by calling the whole environment deterministic.

10.2 Execution, replay and reduction

A seed is necessary but not sufficient. Replay also fixes code and artifact versions, initial persistent state, RNG streams, event tie-breaking rules, clock domains, injected faults, dependency responses, and relevant runtime configuration. The scheduler records enough information to distinguish semantic divergence from a changed harness. Snapshot hashes and the event log identify the executed world.

Use two explicitly labelled lanes. A deterministic semantic lane runs real handlers or verified cores with controlled adapters. An exact-artifact lane runs the candidate binaries with an execution substrate capable of controlling their nondeterministic inputs. If instrumentation changes the artifact, publish that relationship and the instrumented digest; do not claim the production digest was executed. Real cloud/browser integration outside sufficient control is separate empirical evidence.

10.3 Minimize without changing the failure

Reduce events, inputs and faults while preserving protocol feasibility, the same manifest, and the failing property. Re-run every reduction and retain the original trace. A small failure that cannot be replayed is not a replacement for the original witness. Record/replay research shows that faithful interception boundaries matter; a process recorder alone does not reproduce an entire distributed environment. [22]

Targeted fault selection can use dependency/causal information rather than uniform random faults. Lineage-driven fault injection provides a relevant precedent, but its bounded guarantees must not be generalized into unlimited production assurance. [23]

10.4 The empirical assurance boundary

The following claims require observations from the exercised implementation paths, with explicit scenario and fault bounds. They are not consequences of a simulator reporting success alone.

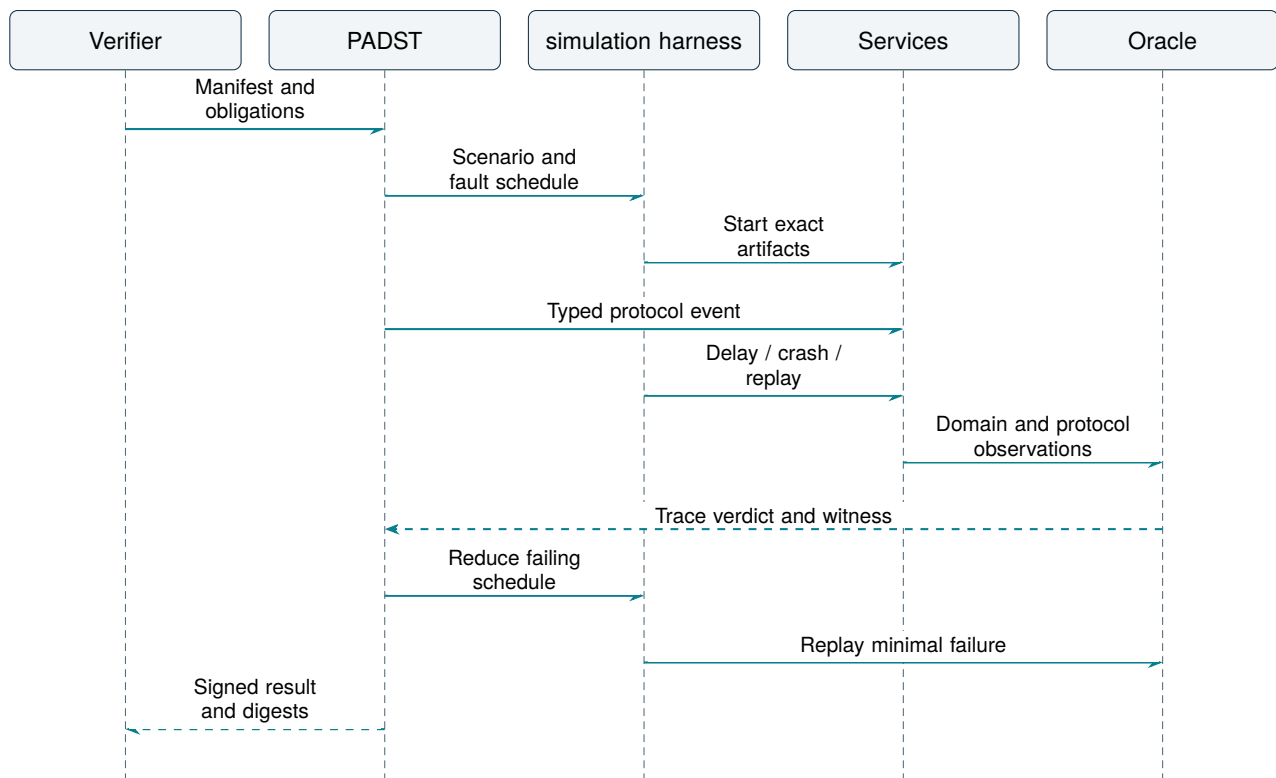


Figure 4: Proposed protocol-testing and simulation sequence. Signing follows verified trace evaluation, not successful job completion alone.

Table 13: simulation harness: controlled execution

protocol-testing and simulation can ascertain in explored runs	Necessary evidence
Only schema-valid messages are emitted.	Capture every relevant egress boundary; reject missing capture.
Transitions occur after their preconditions.	Versioned pre-state, command, authorization and post-state observations.
Retries retain identity and do not double-post.	Ledger effects joined to the same business operation across restarts.
Projections and SSE views do not regress.	Aggregate versions, CDC checkpoints, stream cursors and render observations.
Authorization is enforced by the backend.	Adversarial client requests plus authoritative decision receipts.
Reconciliation closes only matched or escalated items.	Sealed bounded snapshots, matching decisions and accountable exceptions.
FX commits both linked legs or neither.	Actual linked request/result and per-ledger observations in the stated cluster.
Corrections append history.	Original record hashes, new transfer lineage and net-effect comparison.

What they cannot establish unconditionally

Neither framework can determine that the approved specification expresses the right business intent; that every possible production fault was included; or that an external SaaS/cloud dependency always respects its contract. A non-exhaustive test set does not prove universal correctness. A correctly bound rate snapshot does not prove a commercially supplied FX rate is economically correct. No finite campaign establishes unconditional availability.

Likewise, semantic DOM correctness does not imply correct pixels, accessibility or usability. Browser testing must still evaluate focus, keyboard interaction, announcements, layout and real event behaviour. Security, performance and recovery claims need their own workloads and assumptions.

Coverage denominator. Report obligations checked / required, real handlers or artifacts exercised / required, adapters simulated / real, seeds and schedules explored, fault bounds, and unresolved gaps. A green result with an unspecified denominator is not a system claim.

10.5 Conditional liveness and a progress certificate

Safety excludes forbidden effects; payments also require progress obligations. Write $\text{Accepted}(o) \leadsto \text{Settled}(o) \vee \text{Escalated}(o)$ under explicitly named recovery, delivery and decision assumptions. Escalation is an accountable terminal outcome, not evidence that the money arrived. Actual settlement additionally assumes the external rail accepts and eventually processes the instruction.

For an illustrative composition with two operations, define rank $R = u + 2(2 - p) + p - v$, where u is the number lacking approval, p the number posted, and v the visible aggregate version. Initially $R = 6$. A checker should verify nonnegative rank, non-increase on every edge, a decreasing enabled action at every nonterminal state, and persistence of the enabled progress set across rank-preserving edges. Under weak fairness for those continuously enabled progress actions, rank reaches zero within at most six progress actions. A bound of $6B$ total steps additionally requires a progress action within every B steps.

Without fairness the infinite schedule $\text{drop}, \text{drop}, \dots$ is a counterexample; a validation harness should retain this repeating trace. Checking a finite graph certificate with a source interpreter is distinct from proving an infinite-trace theorem in Bend. Unbounded queues, failing banks, expired authorization and production recovery are outside it. Each real journey needs its own progress obligation and assumption monitoring, alongside its safety proofs.

11 Money example I: a basic transfer

Let a command carry tenant, operation ID, debit account, credit account, ledger, positive integer amount and canonical payload hash. Each account stores four unsigned counters: posted debits D^p , posted credits C^p , pending debits D^u , and pending credits C^u . TigerBeetle counters and amounts are uint128; the mathematical model must separately discharge overflow bounds. A derived signed position is $q_a = C_a^p - D_a^p$. It is not a stored negative counter or a universal account-normal balance. [24,25]

For an ordinary posted transfer of $m > 0$ from debit account d to credit account c , the only counter changes are $D_d^{p'} = D_d^p + m$ and $C_c^{p'} = C_c^p + m$. The signed conservation equation follows from that representation:

$$\begin{aligned} q'_d &= q_d - m, & q'_c &= q_c + m, \\ \sum_{a \in \text{ledger}(l)} q'_a &= \sum_{a \in \text{ledger}(l)} q_a. \end{aligned} \tag{7}$$

The conservation proof sketch is cancellation of $-m$ and $+m$. For a credit-funded debit account the reservation-aware requirement is $D_d^p + D_d^u + m \leq C_d^p$; pending credits cannot fund the transfer. Apply the actual account-normal definition: TigerBeetle keeps debit and credit counters and supports configurable balance constraints; a single unsigned “balance” is not a faithful model of every account type. [24,25]

Machine-checked obligations must cover per-asset conservation, account/ledger compatibility, constrained available balances including pending reservations, atomic movement, stable transfer identity, and at-most-one committed financial effect. The arithmetic argument above is a model derivation, not a claim that this paper executed a Bend proof of the production adapter.

Table 14: Money-transfer event/state trace. Different acknowledgements must not collapse into one “success” state.

Step	Observed event	Authoritative interpretation
0	Submit(operation=o, transfer=t)	Identity and payload fixed; backend authorization still required.
1	TransferAccepted(o)	Command accepted; this is not ledger commitment.
2	CreateTransfer(t) sent	Outcome may still be unknown to the caller.

Table 14: Money-transfer event/state trace. Different acknowledgements must not collapse into one “success” state.

Step	Observed event	Authoritative interpretation
3	TransferCommitted(t, version=v)	Durable ledger result is the financial observation.
4	CDC / projection advance	A replayable view of commitment, not a second transfer.
5	Committed fragment / patch	Visible success is justified by the same operation and durable observation.

Atomic movement here means the two sides of the ledger transfer commit together. It does not assert an atomic transaction across TigerBeetle, a bank rail, a workflow database and a browser. Those are separate protocol boundaries.

11.1 Timeouts, retries and pending funds

Stable identity is the essential bridge

Record or derive a stable transfer ID for the business operation and reuse the same command fields on retry. TigerBeetle’s reliable-submission guidance treats application-level retry identity as essential; generating a new ID after an uncertain result can create another valid transfer. Duplicate-result handling must also distinguish a matching existing transfer from an ID collision or conflicting payload. [26,27]

Table 15: Money example I: a basic transfer

Injected execution	Required recovery	Financial oracle
Timeout before commit	OutcomeUnknown; retry or lookup using the same transfer ID and payload.	Zero effects until one valid commit; never manufacture a replacement ID.
Timeout after commit	Recover the existing result or retry the identical command.	One effect, even if the API or queue delivered the command twice.
Restart after send, before local receipt	Recover the persisted identity and pending workflow state.	No loss of the business-to-ledger identity mapping.
Same ID, different amount or account	Reject the conflict; escalate possible mapping/collision defect.	No second financial effect under the reused identity.

Pending transfers are not mutable records

TigerBeetle pending transfers reserve funds without posting them. Posting, voiding and expiration resolve that reservation; resolution uses a new transfer referring to the pending ID. A pending transfer can be resolved only once. A partial post posts the selected amount and releases the remainder; it is not a reusable reservoir for repeated partial posts. The original transfer remains immutable. [28]

For a 10000-unit reservation partially posted for 6000, the remaining 4000 is released. A second post against the same pending transfer must not be used to consume it later. If the business needs staged capture, model separate reservations or a different explicit protocol. PADST must explore post/void races, expiration, retry of the resolution transfer and restart between observing the ledger and persisting the workflow result.

Proof sketch for retry idempotency

$$\text{financialObs}(\text{apply}(\text{apply}(s, c), c)) = \text{financialObs}(\text{apply}(s, c)) \quad (8)$$

The abstract transition returns the stored outcome when the scoped identity and canonical payload already exist. Prove the first successful application records that identity atomically in the modeled authority; later identical applications preserve the financial projection. Real persistence, cross-store recovery and ID-retention assumptions must then be discharged by the implementation relation and integration evidence.

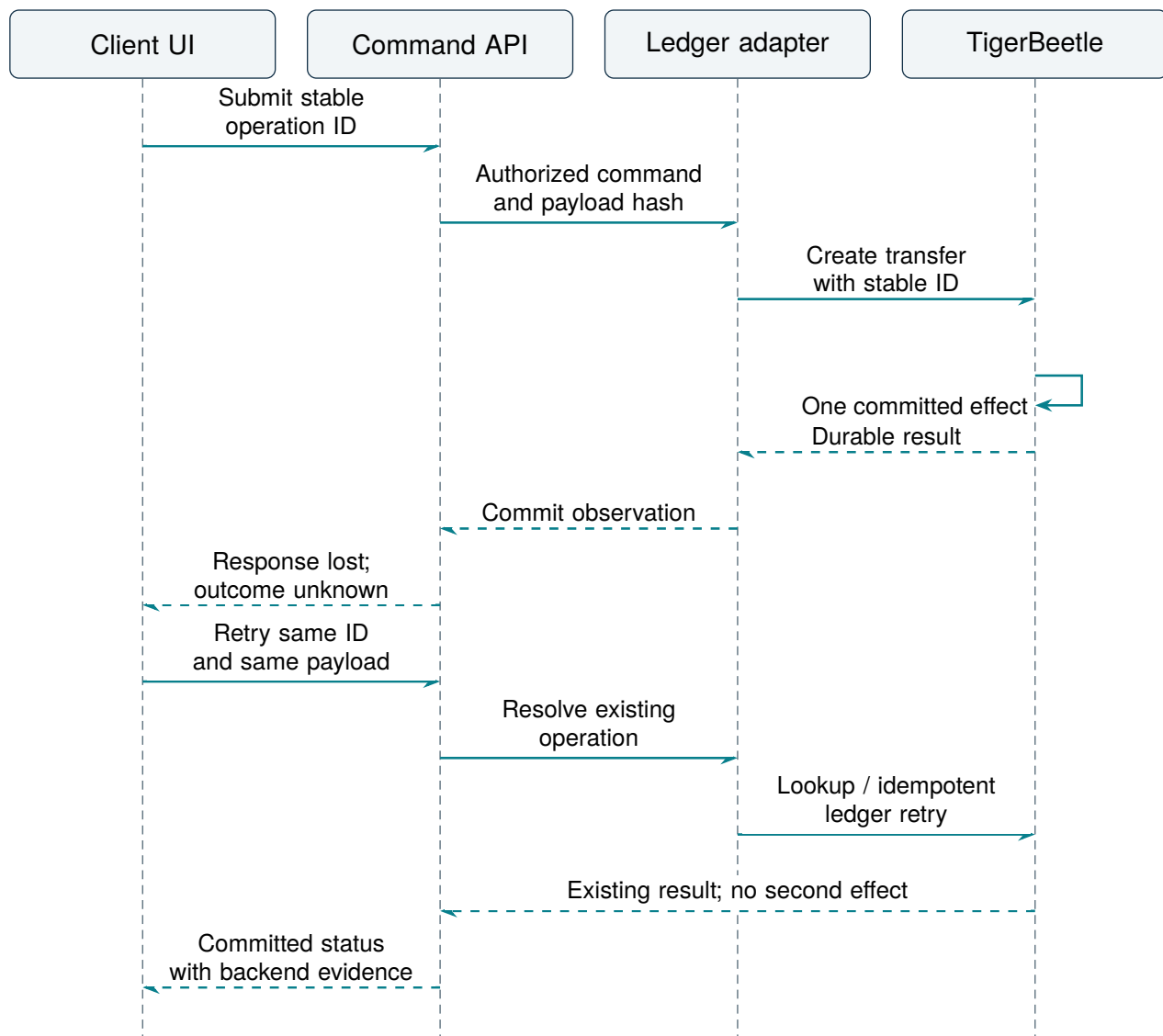


Figure 5: Timeout after commit: a retry retains the original operation identity and observes the existing financial effect. This is a proposed protocol trace, not a claim about a deployed adapter.

11.2 Pending and posted are separate ledger states

For a reservation $r > 0$, increment D_d^u and C_c^u by r , leaving both posted counters unchanged. Resolving it with a post amount $0 < p \leq r$ removes the full r from both pending counters and adds only p to the corresponding posted counters. Thus a partial post releases $r - p$; it cannot be posted a second time. A void removes the reservation without changing posted counters. An explicit post or void appends a new transfer referencing the pending ID; the original transfer is immutable. Expiry releases the reservation when processed, and its observed counter release may lag the deadline. [27,28]

$$\begin{aligned} \text{reserve } r : \quad D_d^{u'} &= D_d^u + r, & C_c^{u'} &= C_c^u + r, \\ \text{post } p : \quad D_d^{u'} &= D_d^u - r, & C_c^{u'} &= C_c^u - r, \\ & D_d^{p'} &= D_d^p + p, & C_c^{p'} &= C_c^p + p. \end{aligned} \quad (9)$$

The subtraction releases an existing reservation; it never permits negative counters. Both legs of a linked pay-out have pending debit *and* pending credit entries. Linked chains must cover the reservation and resolution operations separately. A batch alone is not atomic, and a chain ends with linked=false. [29]

Ledger adapter units. The arithmetic fixture uses cents. An illustrative adapter using a 10^7 scale would represent GBP 100.00 as 1,000,000,000 ledger units rather than 10,000 pence. Currency identity and numeric scale are separate mappings. Check each conversion before encoding a real ledger request; the earlier figures are explanatory minor-unit examples, not production payloads.

12 Money example II: reconciliation

Reconciliation relates three bounded populations: **B**, business/control-plane operations; **T**, TigerBeetle transfers; and **X**, external bank or processor records. Define deterministic matching over operation ID, transfer ID, asset, integer amount, direction, account mapping, settlement reference and source high-water mark. A duplicate or ambiguous key is an exception, not permission to select an arbitrary match.

$$\begin{aligned} \forall b \in \text{Posted}(B) : \exists! c \in \text{Outcomes}(b). \text{ valid}(c), \\ \text{Matched}(b) \Rightarrow \exists! t \in T. \text{ match}(b, t), \\ \forall t \in \text{InScope}(T) : (\exists b \in B. \text{ match}(b, t)) \vee \text{classified}(t). \end{aligned} \quad (10)$$

Uniqueness applies to the classification at an immutable snapshot cut. A duplicate yields one Duplicate outcome retaining all candidates, not a fabricated unique match. The Matched branch requires exactly one transfer for a one-leg operation. Expand FX, fees and corrections into ExpectedLegs before applying that branch. Administrative closure additionally requires every outcome to be Matched or Escalated; classifying a discrepancy alone does not close it.

“Classified” means a permitted, versioned classification with provenance and an accountable disposition, such as an identified correction or an escalated orphan. It is not a catch-all predicate that can be set true to satisfy the equation. External completeness additionally requires each in-scope settled leg to have its expected X association or a declared timing difference/escalation.

12.1 Completeness needs an immutable cut

Table 16: Money example II: reconciliation

Snapshot field	Why it is required
Window and scope	Tenant, accounts, assets, operation kinds, business-time interval and inclusion policy.
Source cuts H_B, H_T, H_X	Named per-source or per-partition high-water marks; event time is not ingestion progress.
Content and identity hashes	Canonical records, mapping/normalization versions, source query and pagination boundaries.
Completeness witnesses	Every page consumed, source cut acknowledged, no unresolved cursor gaps or unexplained record loss.

Table 16: Money example II: reconciliation

Snapshot field	Why it is required
Decision provenance	Matching algorithm/policy version, exception classifications, reviewer and snapshot digest.

Seal these inputs before claiming completeness. TigerBeetle historical balance queries require the relevant account history capability; capture the account configuration and pagination/cut semantics. Automatic pending-timeout expiry is not a new historical-balance entry, so a historical row is not a universal pending-state snapshot. A current balance alone cannot show that every expected transfer was reconciled. [25,30]

12.2 Reconciliation lifecycle and faults

Unseen items become Matched, TimingDifference, AmountDifference, MissingInternal, MissingExternal or Duplicate. Unresolved discrepancies move to Escalated with owner, reason and follow-up. A window may close administratively only when every in-scope item is matched or explicitly escalated. “Closed with exceptions” must remain distinct from “all funds matched.” A late correction creates a linked amendment or a new window; it does not rewrite the original completeness receipt.

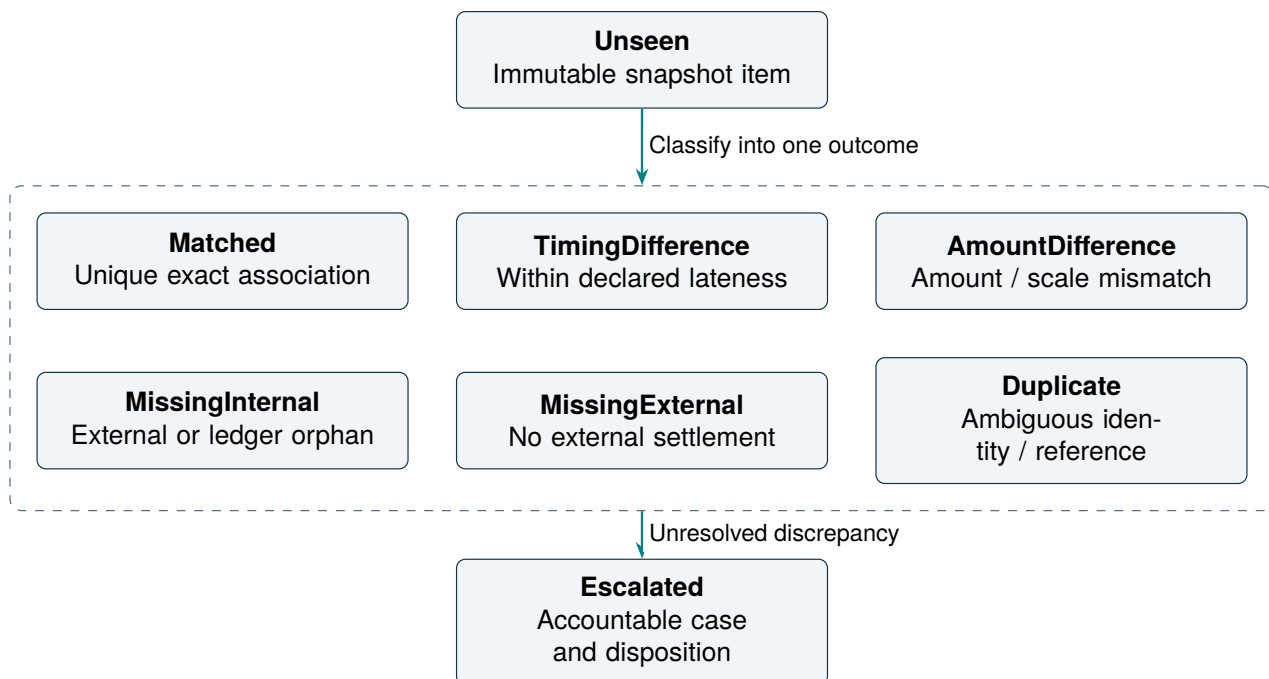


Figure 6: Reconciliation state classification. Choose one outcome; unresolved discrepancies escalate. Reclassification uses new evidence or an amended snapshot; the sealed earlier cut remains immutable.

Table 17: Money example II: reconciliation

PADST fault	Expected finding
Missing or duplicated CDC event	Cut/gap check or deduplication; no silent omission/double count.
Late external record	TimingDifference, then a versioned match/amendment.
Wrong currency or amount	Asset/scale mismatch or AmountDifference; no fuzzy auto-match.
Duplicate settlement reference	Duplicate/ambiguity with both candidates retained.
Correction after window close	New lineage-linked snapshot; prior snapshot digest unchanged.
Snapshot high-water mismatch	Reject completeness claim and identify the inconsistent cut.

The proof target is the matching/classification algorithm over the declared finite snapshots, including uniqueness and closure accounting. It does not prove that an external institution supplied all reality. PADST exercises the protocol obtaining those cuts; the simulation harness replays the precise failed schedule, including delayed publication and record arrival.

12.3 Reconcile all four counters without spending pending credits

A posted trial balance is only one control. For a complete closed ledger population L at a consistent cut, require both:

$$\sum_{a \in L} D_a^p = \sum_{a \in L} C_a^p, \quad \sum_{a \in L} D_a^u = \sum_{a \in L} C_a^u. \quad (11)$$

These equalities neither identify the correct beneficiaries nor establish agreement with the bank. A subset such as the real-bank corridors alone need not balance; include its actual counterpart accounts or prove the boundary-flow relation. Shared edge accounts serving multiple tiers cannot be assumed to close a selected subset.

For a credit-funded corridor, distinguish the signed posted position $C^p - D^p$, the unclamped spend position $C^p - D^p - D^u$, and both gross pending counters. Compare a bank closing position with the sum of *posted* positions in its reviewed corridor-plus-allocation set at the declared cut. Pending reservations have their own identity, status, expiry and resolution checks. Do not mix them into the bank-closing amount to obtain equality.

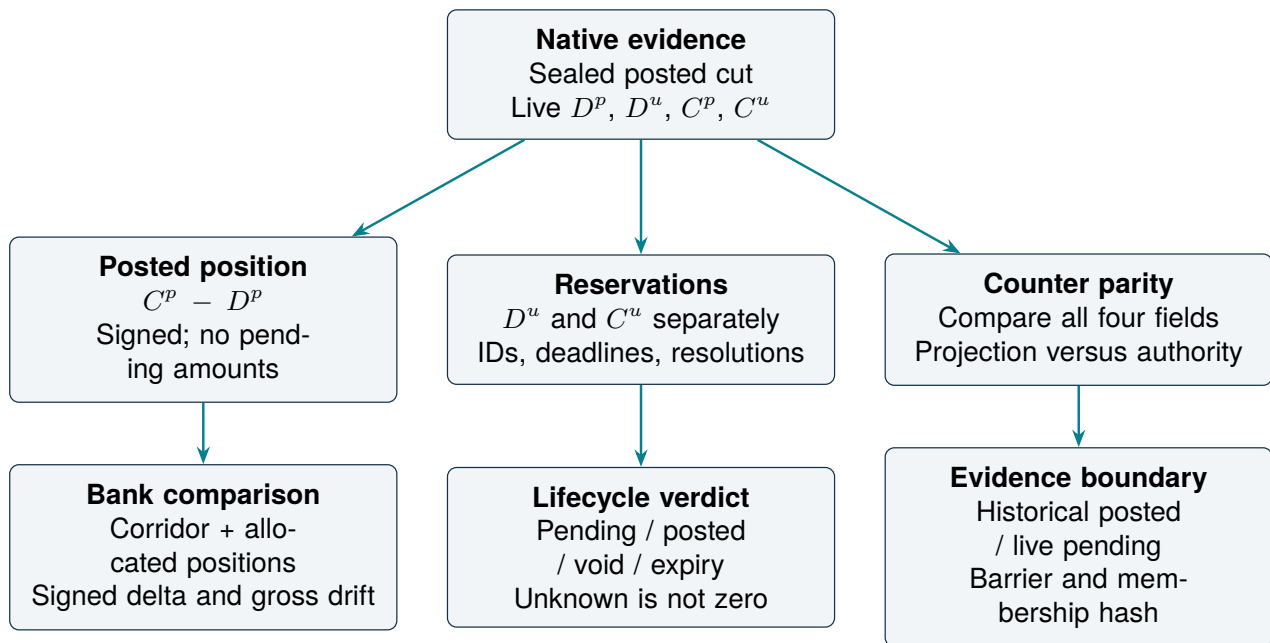


Figure 7: Counter-aware reconciliation. Bank closing agreement, reservation lifecycle, and four-counter parity are distinct assertions. All observations require compatible identity, scale, provenance and cuts.

Table 18: Illustrative E7 example: GBP 100 inbound, GBP 30 reserved, GBP 20 partially posted. Every displayed stored value is nonnegative.

Step / account	D^p	D^u	C^p	C^u
Inbound / external	1000000000	0	0	0
Inbound / corridor	0	0	1000000000	0
Reserved / external	1000000000	0	0	300000000
Reserved / corridor	0	300000000	1000000000	0
Partial / external	1000000000	0	200000000	0
Partial / corridor	200000000	0	1000000000	0

During reservation the corridor has GBP 100 posted and GBP 70 spend position. Afterwards it has GBP 80 posted and no pending amount. A void instead leaves GBP 100 posted. The paired pending credit is visible throughout; an unallocated destination cannot be omitted from a reservation.

12.4 Sealed posted positions and live counters

A useful observation is $O_4(s) = (K_s, R_s, J_s)$, where K_s retains all four counters per account, R_s retains reservations, and J_s is the resolution journal. Exercise full posting, partial posting, void and expiry separately. Reject conflicting retries, excess posts, second resolutions, unknown accounts and invalid amounts. A discrepancy that increases pending debits and credits equally can escape a net-only check; comparing all four counters exposes it.

For an illustrative reclassification policy, allocation moves value from a bank-linked pool to a project account. The bank position is the sum over a reviewed attribution set S_c containing the pool and its allocated positions. Every member is counted once; sets for different bank accounts must be disjoint. For example, a bank balance of GBP 100 may correspond to GBP 75 in the pool plus GBP 25 in a project. A pool-only comparison would report false drift.

$$B_c(t) = \sum_{a \in S_c} (C_a^p(t) - D_a^p(t)), \quad S_c \cap S_d = \emptyset \quad (c \neq d). \quad (12)$$

Bind membership, currency and business day into the receipt. Missing or overlapping members must fail the comparison. Arithmetic equality cannot establish the business completeness of an incorrectly approved attribution set. Keep legal entities, bank accounts and account ownership distinct even when they share a currency.

A historical posted-position collector needs a bounded, consistent source query and an explicit cutoff. Pending and void records do not themselves contribute a posted amount; a posting of a pending transfer contributes its actual amount once. Bind record hashes, counts and source identity, and fail rather than silently truncate when query limits are exhausted.

An imported-transfer timestamp barrier is one possible premise for stabilising a historical prefix under the documented TigerBeetle contract. It requires careful treatment of the cluster's last committed timestamp and the observation cutoff; a collector must not create a financial transfer merely to manufacture that premise. This is a protocol design consideration, not a theorem supplied by this paper. [31]

Store historical posted cuts separately from current counter observations. Live pending values and expiry effects cannot be reconstructed from posted history alone. Preserve their actual observation times, and never relabel a projection-only result as a native historical observation. Validate the design against the selected ledger version before making an operational claim.

13 Money example III: foreign exchange

Tigerlings is TigerBeetle's official exercise repository, including linked and two-phase transfers. It is useful for validating adapter understanding, not as a formal certification. The official FX recipe uses separate ledgers for currencies and a liquidity provider. Two same-ledger transfers are atomically linked; direct transfers between accounts on different ledgers are not the mechanism. [32,33]

Table 19: Money example III: foreign exchange

Leg	Ledger	Debit → credit	Amount	Linked
T1	USD	$A_1 \rightarrow L_1$	10000	true
T2	INR	$L_2 \rightarrow A_2$	824214	false

For 100.00 USD at the documented rate 82.42135 INR per USD, the quoted destination amount rounds to 8,242.14 INR. With two decimal places for both assets, the minor-unit amounts are 10000 and 824214. The final event has linked=false; the preceding flag links its outcome to the next event. The chain succeeds or fails as a unit within one TigerBeetle cluster. An ordinary batch is not automatically one atomic chain. [29,33]

$$\begin{aligned} \Delta \sum q_{\text{USD}} &= -10000 + 10000 = 0, \\ \Delta \sum q_{\text{INR}} &= -824214 + 824214 = 0. \end{aligned} \quad (13)$$

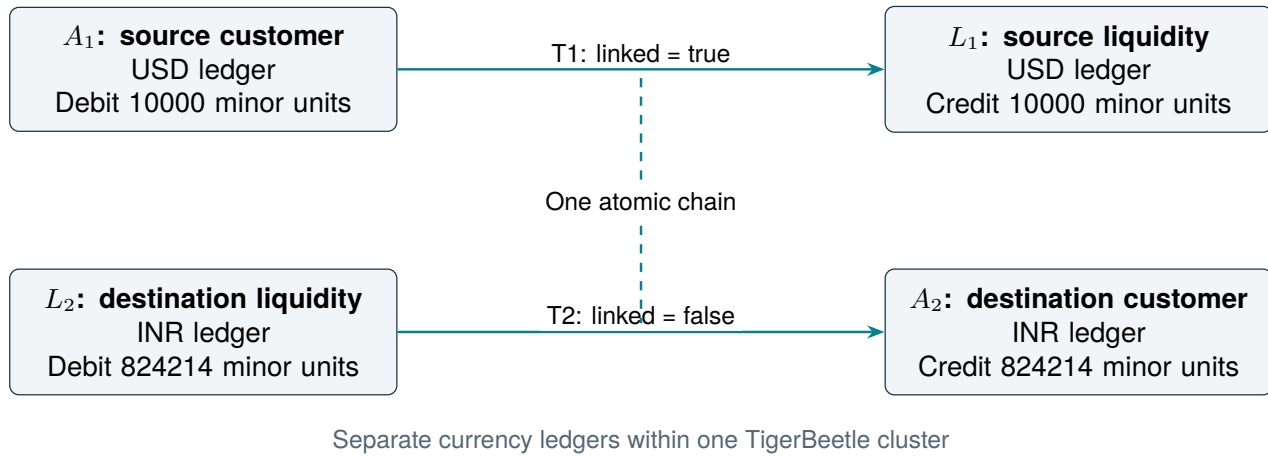


Figure 8: Official recipe structure, with the documented numerical example. The dotted connection denotes linked atomic execution, not an exchange of native ledger units between currencies.

These are independent conservation laws. Adding USD and INR as though they were a single asset is dimensionally invalid. The economic value of the trade, liquidity sufficiency and commercial fairness of the rate are separate questions. The model must bind each account to the correct ledger, asset and scale and verify both liquidity constraints before claiming a successful atomic exchange.

13.1 Quote binding, rounding and retries

$$destination_minor = \text{Round}_{\text{mode}} \left(source_minor \times \frac{rate_num}{rate_den} \times 10^{scale_{\text{destination}} - scale_{\text{source}}} \right) \quad (14)$$

Represent a rate as an exact rational or a reviewed fixed-decimal value. For the example, 8242135/100000 is the rate; 10000 minor units produce 824213.5 destination minor units before rounding. A declared half-up rule yields 824214. This derivation demonstrates deterministic arithmetic for the example; it does not establish that the quoted market rate was correct.

The immutable quote receipt records assets and scales, rational rate, source/provider, timestamp, expiry, rounding mode, spread/fees, and a payload hash bound to the authorized operation. Prevent overflow in bounded production types even when the formal model uses unbounded naturals. Confirm that decimal scale changes invalidate dependent proofs and fixtures.

13.2 Fee and expiry semantics must be explicit

Record a fee/spread as an explicit additional transfer where applicable; the official recipe illustrates a separate fee leg in the linked chain. A business decision to incorporate a fee into a quote instead still needs an auditable decomposition. Do not hide accounting consequences inside a display rate. [33]

Define the expiry linearization point. “Quote valid when the backend accepted this operation” is different from “ledger commit must occur before expiry.” The latter needs an enforceable execution protocol and timing assumptions; a clock check before an RPC does not prove it. After timeout with unknown outcome, preserve the original authorized quote and transfer IDs while resolving that outcome. Never reprice an uncertain operation into new financial effects.

Table 20: Money example III: foreign exchange

Obligation	Adversarial evidence
All-or-nothing linked execution	Reject one liquidity/account leg; observe no committed leg in the chain.
Stable quote and identity	Drop response, expire displayed quote, retry; same IDs and authorized payload.
Scale and rounding	Boundary fractions, extreme amounts, wrong scale and overflow inputs.

Table 20: Money example III: foreign exchange

Obligation	Adversarial evidence
Explicit fees and liquidity limits	Missing fee leg, insufficient liquidity, mismatched accounts and chain-flag errors.

The formal law covers arithmetic and the modeled linked transition. Actual cluster atomicity, adapter encoding, price-provider authenticity and clock bounds remain separately identified premises or integration results.

For an illustrative E7 adapter, the example amounts become 1,000,000,000 USD ledger units and 82,421,400,000 INR ledger units, respectively. Both currency ledgers use E7 here; their minor-unit conversions remain separately checked. Conservation applies to each ledger's debit/credit counters, not to a sum of USD and INR magnitudes.

14 Money example IV: corrections

TigerBeetle transfers are immutable. A mistake is corrected by new transfers that reverse or adjust the earlier effect; lineage belongs in explicit application/accounting metadata and identifiers. A posted transfer is not edited in place, and voiding an unresolved pending transfer is not the same operation as reversing a posted one. [28,34]

$$\text{net}(\text{originals} + \text{corrections}) = \text{intended_net},$$

(15)

originals remain immutable.

Define net as a vector indexed by ledger and account, in each asset's own integer units. If the original posted effect is (+m at destination, −m at source), a full reversal appends the opposite vector. Their sum is zero, while the original record remains addressable. A partial correction appends only the authorized difference. The lineage must bind the original transfer, correction operation, amount, reason, approver and associated reconciliation amendment.

For a 10000-unit pending reservation posted for 6000, correction entitlement is based on the 6000 actually posted, not the original reservation. Returning 10000 would over-reverse. Concurrent corrections must atomically allocate the remaining entitlement in the modeled authority; distinct correction IDs alone do not prevent cumulative excess.

Table 21: Money example IV: corrections

Scenario	Expected behaviour
Nonexistent original	Reject or explicitly escalate; never invent matching lineage.
Excess reversal	Reject when cumulative corrections exceed the authorized posted effect.
Duplicate reversal ID	Recover the prior result; no additional correction effect.
Reversal after partial post	Use actual posted amount; released pending funds are not reversal entitlement.
Wrong asset or account mapping	Reject before ledger submission; preserve the original record.
Original + correction reconciliation	Match the complete lineage and net effect; retain both historical entries.

The abstract proof composes additive correction vectors and an immutable-history invariant. Backend integration must show that crashes between entitlement allocation and ledger acknowledgement cannot lose identity or permit another effect. If those steps cross stores, specify a recoverable protocol rather than assuming a distributed transaction that does not exist.

In counter vocabulary a reversal increments the new debit account's D^p and the new credit account's C^p . It does not decrement the original posted counters. Net correction equality is an equality of derived signed positions plus immutable lineage, not an instruction to store a negative transfer amount.

15 UI observation protocols

The UI is an observation and command boundary, not an authorization authority. Let U be visible semantic state, L an island/reducer model, R a read projection and D the domain state. An honest interface may optimistically show Submitting or Pending; it must not claim Committed without versioned backend evidence for the same aggregate and operation.

$$\begin{aligned} (U, L, R, D) &\rightarrow request \rightarrow command \rightarrow D' \rightarrow R' \\ &\rightarrow response/patch \rightarrow (U', L', R', D') \end{aligned} \quad (16)$$

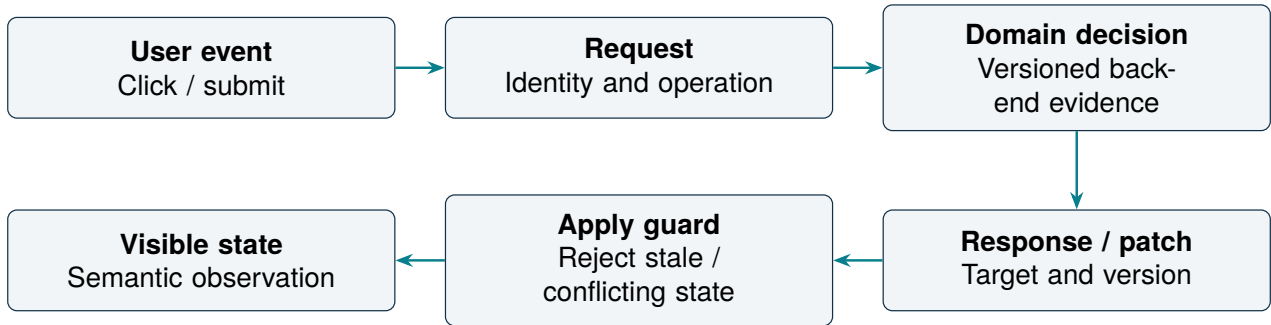


Figure 9: UI assurance path. The version/identity guard is an application requirement, not a guarantee automatically supplied by a UI library.

15.1 HTMX: request, target and swap

HTMX connects a user event to an HTTP request and an HTML-fragment update. The application declares the event, request method, target and swap behaviour. Request, response and swap lifecycle events are observable integration seams. An afterRequest or afterSwap event is not evidence that a financial operation committed. [35–37]

Model the chain explicitly: user event $\rightarrow$ HTTP request $\rightarrow$ authenticated server command $\rightarrow$ versioned fragment response $\rightarrow$ target selection $\rightarrow$ DOM swap $\rightarrow$ resulting visible state. The server must distinguish transport acceptance from durable domain completion. HTTP 202 does not mean that asynchronous processing completed successfully. [38]

Before applying an authoritative fragment, check aggregate ID, operation ID and version against the rendered region. A response from an older request must not overwrite a newer result. A Committed fragment requires an identified durable observation; a timeout fragment must not masquerade as a rejection if the effect may already exist.

Ownership and lifecycle

Give each live DOM region one owner. HTMX must not replace an Elm-owned root or a Datastar-owned region without an explicit mount/unmount protocol. Integration tests inspect target resolution, out-of-band updates, cancellation, browser reload, focus and island reinitialization. Proving a server transition does not prove that the browser swapped the intended fragment or preserved accessibility state.

15.2 Elm islands: a typed local core

The Elm Architecture separates Model, messages, Update and View. Ports bridge Elm to JavaScript and therefore form an explicit integration boundary. The Elm compiler's type checks are useful, but they do not by themselves constitute a refinement proof for an application or prove that every external port value has the intended domain meaning. [39, 40]

Listing 7: Illustrative algebraic model, not compiled Elm or Bend code.

```
Model = Idle
  | Submitting(OperationId)
  | Pending(OperationId, Version)
  | Committed(OperationId, Version, Receipt)
  | Rejected(OperationId, Version, Reason)
  | Conflict(OperationId, LatestVersion)

Msg = Submit | Retry | Cancel
  | Accepted(OperationId, Version)
  | DomainObserved(OperationId, Version, Status, Receipt)
  | TransportFailed(OperationId)
```

Local laws to prove in a formalized reducer

$$\begin{aligned} \forall \text{ valid } (m, \mu) : \quad & \text{Update}(m, \mu) \text{ is defined,} \\ & \text{version}(\text{Update}(m, \mu)) \geq \text{version}(m), \\ & \text{View}(m) \text{ reflects the declared semantic state of } m. \end{aligned} \quad (17)$$

A total reducer handles every valid Model/Msg pair; an invalid external message is rejected by a typed decoder before it enters that domain. Algebraic variants prevent contradictory combinations such as a Committed state with no operation identity or receipt. A stale DomainObserved message is ignored or triggers a controlled resynchronization, never a regression. Tie every status to the correct operation; two operations on one aggregate must not share an ambiguous success banner.

A view theorem is relative to a normalized observation of the rendered view, not an arbitrary screenshot. A production connection can use verified reducer translation or language-specific verification; otherwise compare the actual Elm implementation with the model through property/state-machine tests. Ports must preserve IDs, versions, units and rejection semantics in both directions.

End-to-end check

PADST sends accepted, stale, duplicated, reordered and malformed port/domain messages around a payment submission. The simulation harness controls the relative timing of the domain event, projection advance and browser response. Browser integration verifies the real decoder, port wiring, region ownership and rendering lifecycle. UI state remains untrusted input when the backend decides whether a command may execute.

15.3 Datastar: versioned streams

Datastar supplies reactive signals and backend actions. Its SSE response protocol includes datastar-patch-elements and datastar-patch-signals; element patches normally morph ID-matched elements. These mechanisms do not themselves supply the application's aggregate ordering, deduplication, authorization or durable-commit guarantees. [41–43]

Listing 8: Illustrative authoritative-patch envelope. The integration must define where this metadata is encoded and how it is checked before application.

```
# Proposed application metadata; not native Datastar fields.
aggregate_id: payment-42
operation_id: payment-42-submit-7
aggregate_version: 18
stream_id: payment-42-epoch-3
stream_sequence: 91
payload_hash: sha256:...
observation_receipt: sha256:...
patch_kind: datastar-patch-elements # or patch-signals
```

$$\begin{aligned}
&\text{applied}(v_2) \text{ after } \text{applied}(v_1) \Rightarrow v_2 \geq v_1, \\
&\text{same}(\text{stream_id}, \text{sequence}) \Rightarrow \text{same}(\text{payload_hash}), \\
&\text{Committed}(\text{version} = v) \Rightarrow \exists \text{ durable domain observation at version } \geq v, \\
&\text{reconnect}(\text{last_event_id}) \Rightarrow \text{rendered aggregate cannot regress.}
\end{aligned} \tag{18}$$

Interpret these laws per aggregate and operation, not as one global version across the browser. Strengthen the commitment law to require a matching committed observation for that same identity and meaning; a newer unrelated event is not sufficient. Hash the canonical patch meaning, including target/mode where relevant, so a repeated sequence with a different payload is a protocol violation.

A stream restart needs an explicit epoch and cursor policy. Last-Event-ID is a transport cursor, not a domain version or an authorization token. Native EventSource semantics and Datastar’s Fetch-based backend-action transport are distinct; the selected client/adaptor must define and test how reconnection state is carried. Reconnect alone does not establish application replay correctness. [42, 44]

Client-editable signals may express intent or optimistic state. They must never serve as evidence of permission, approval threshold, or durable commitment. The backend authenticates and authorizes every effect independently.

15.4 UI boundaries and adversarial delivery

Table 22: HTMX, Elm and Datastar proof boundaries. Library adoption does not automatically discharge the formal-core target.

Approach	Formal-core target	Required implementation evidence
HTMX	Request/fragment observation relation; version and command identity; permitted visible outcomes.	Actual events, target resolution, swap/cancel behaviour, CSRF handling and DOM ownership.
Elm island	Total Update over valid inputs, representable-state invariants, monotonic versions, View projection.	Compiled implementation relation, port decoders/adapters, mount/unmount and browser behaviour.
Datastar	Signal/patch transition model, stream dedupe, version monotonicity and commit-receipt relation.	Actual SSE framing, Fetch/reconnect handling, selectors, signal updates and pre-application guards.

An SSE failure trace

The browser renders aggregate version 18. The simulation harness delays its next projection update, delivers a duplicate patch (epoch 3, sequence 91), disconnects halfway through a new SSE event, then reconnects and offers a stale version-17 patch followed by version 19. PADST expects the duplicate to be harmless, incomplete framing to produce no partial authoritative update, version 17 to be rejected, and version 19 to apply only after valid identity and observation checks.

A sequence reused with a different payload hash is not another harmless duplicate: stop or resynchronize and record the conflict. If several element/signal patches together represent one authoritative state, define a versioned patch bundle and completion barrier. Do not expose a success label halfway through an incoherent bundle. These are proposed adapter semantics, not claims that Datastar supplies an atomic application transaction.

Table 23: UI observation protocols

Additional fault	Expected response
Signal conflict with backend result	Server authority wins under a declared merge policy; never trust a client permission flag.
Expired replay cursor	Receive an authoritative snapshot/new epoch; no regression or silent message loss.
Browser reload / duplicate island	Restore identity/version safely; one owner and one effective command path.
Projection newer than response	Reject stale fragment/patch or revalidate before application.

Visual layout, pixel correctness, focus, usability and assistive-technology behaviour remain browser/accessibility evidence. A semantic-state theorem cannot substitute for those checks.

16 Authorization and dual approval

Authentication establishes identity evidence; authorization decides whether that identity may perform a specific action. Model a payment with creator, canonical command hash, approvals, policy version, amount/risk band, status and an authorization epoch. OIDC token validation and Cedar policy evaluation are environmental/implementation boundaries, not consequences of a pure payment-state theorem. [45,46]

$$\begin{aligned} & \text{creator}(p) \notin \text{eligibleApprovers}(p, \text{policy_version}), \\ & \text{Executable}(p) \Rightarrow |\text{distinctEligibleApprovals}(p)| \geq \text{required}(p), \\ & \text{approval.command_hash} = \text{payment.command_hash}. \end{aligned} \quad (19)$$

For maker-checker with dual approval, require the specified number of distinct eligible identities, excluding the creator. Approval evidence is bound to the payment, action, command hash, policy version and validity interval. Execution re-evaluates current authorization according to an explicit policy for membership and policy changes. A stale approval cannot be silently grandfathered unless approved business policy explicitly permits that semantics.

The formal transition cannot construct Executable until the threshold and separation rules hold. Concurrent approvals require atomic deduplication or serialization in the implementation relation. Replayed or expired evidence must not increase the count. An ID token, a disabled button, or a client signal is not a backend payment permission.

Table 24: Authorization and dual approval

PADST / simulation harness scenario	Expected boundary behaviour
Group removal after page load	Re-evaluate at approval/execution; stale UI permission grants nothing.
Token expiry during approval	Validate at the declared decision point; reject expired evidence.
Policy version changes	Invalidate/re-evaluate affected approvals or apply explicit approved migration rules.
Duplicate / concurrent approvals	One eligible identity contributes at most once; threshold is not raced.
Policy-store outage	Fail closed or use an explicitly bounded, approved cached-policy contract.
Break-glass request	Separate accountable flow with scoped authority, reason, expiry and audit; not an ordinary approval bypass.
Stale UI permissions	Backend denial remains authoritative even if the control is visible/enabled.

Declare assumptions for issuer behaviour, key validity/rotation, token audience and replay controls, membership freshness, policy-store consistency, clock skew and availability. A proof assuming fresh identity data must not be reported as unconditional revocation enforcement.

17 Cross-repository GitHub Actions design

17.1 Component PR: untrusted candidate, trusted obligations

A component PR begins by resolving behavioural impact against protected baseline policy. The agent's task file cannot choose the entire verification set. Reusable workflows are pinned to full commit SHAs, and proof tools, rules, observation functions and artifact-verification policy are independently controlled. GitHub documents SHA references as the safest reusable-workflow option. [47]

Table 25: Cross-repository GitHub Actions design

Step	Action and immutable output
1 · Classify	Detect exported semantic changes, resolve affected rule IDs and required assurance classes.
2 · Validate intent	Validate the selected Allium version; require domain approvals for law or assumption changes.
3 · Generate	Produce deterministic IR, source maps and Bend laws; reject unsupported or dropped obligations.
4 · Check proofs	Use pinned checker/dependencies; reject unapproved unsafe assumptions and proof holes; run positive/mutation controls.
5 · Connect implementation	Check verified bindings or clearly labelled differential/property conformance; run local tests.
6 · Build candidate	Publish an immutable quarantined artifact with full source SHA and digest; attach build provenance.
7 · Carry evidence	Publish component contracts, proof receipts, test scope and trust inventory.
8 · Request composition	Ask the system verifier to evaluate this exact candidate against a locked baseline manifest.

Untrusted PR code executes in isolated, disposable runners without signing or cross-repository credentials. A separate trusted verification/signing stage consumes validated immutable outputs and must not execute arbitrary candidate scripts with privileged tokens. A build job may produce malicious data; paths, archives, manifests and reports all need validation before a privileged consumer touches them. [48]

A reusable workflow pin does not by itself protect against executing attacker-controlled code in that workflow. Review its checkout target, tool installation, cache usage, permissions and artifact-handling boundary. The exact verifier workflow SHA becomes part of the evidence subject relation.

17.2 System verification and release

Table 26: Cross-repository GitHub Actions design

System-verification stage	Required result
Assemble candidate	Lock every component SHA/digest, contracts, IR, proofs, infrastructure and deployment policy.
Resolve affected closure	Include changed assumptions, observations, ordering, identities, units and deleted graph edges.
Compose formal arguments	Verify edge compatibility and affected end-to-end Bend trace laws, under explicit environmental assumptions.
Generate protocols	Construct PADST scenario sets, positive witnesses and obligation-specific negative controls.
Execute exact target	Deploy the locked manifest into the supported simulation harness execution lane; disclose simulated or instrumented boundaries.
Inject and observe	Run deterministic schedules, browser/ledger integration cases and recovery paths; capture complete traces.
Diagnose and attest	Replay/minimize failures; sign only a complete, correctly scoped verdict with exact input/output hashes.
Report to origin	Publish the required check on the original source SHA/PR using an authorized cross-repository identity.

After merge

Publish an immutable component release, then open an automated system-manifest PR containing exact versions. Re-run composition and the affected evidence for that manifest; approve and deploy only its digest. Preserve the evidence alongside the deployment attestation. If a merge commit, merge queue result, build configuration or artifact differs from the tested candidate, rebind or rerun the affected checks rather than transferring a green badge by name.

A release controller verifies manifest approval, signature identity, artifact digests, policy version, result expiry and any required assumption freshness. A rollback is another explicit manifest selection: a previously tested old version may be incompatible with new schemas or infrastructure. Historical evidence is retained, but current deployment eligibility is evaluated anew.

Cross-repository completion is a protocol

A request ID correlates component PR, candidate manifest and verifier result. Deduplicate duplicate dispatches; ignore stale results for superseded PR heads; reject result substitution across repositories or manifests. The component repository must wait for the authenticated system verdict, not merely successful delivery of a dispatch request. [49,50]

17.3 Credentials, triggers and attestations

Use OIDC federation with a narrowly scoped trust policy for artifact storage and signing. Validate issuer, audience, repository identity and workflow identity/ref as applicable; id-token permission is not authority to modify another GitHub repository. The normal GITHUB_TOKEN is scoped to its repository. Cross-repository dispatch and check reporting therefore require a separately authorized, short-lived credential—for example a narrowly installed GitHub App accessed through a trusted broker. [50–53]

Table 27: Cross-repository GitHub Actions design

Threat boundary	Required design choice
repository_dispatch / workflow events	Treat payloads as untrusted requests. Dispatch runs are not proof, and their default-branch SHA is not the original PR SHA.
Privileged PR triggers	Do not execute untrusted PR code under privileged pull_request_target or workflow_run credentials.
Workflow reuse	Pin caller/callee/action dependencies to reviewed full SHAs; protect changes to trust policy and verifier logic.
Signing and provenance	Validate subject digest and signer/workflow identity; an attestation is origin evidence, not a correctness theorem.
Cross-repository result	Bind result to source repository ID, full source SHA, PR and exact manifest; report through scoped checks permission.
Expiry and drift	Revoke applicability when assumptions, policy, dependencies or artifacts differ; do not rewrite historical receipts.

Result binding: minimum signed fields

Listing 9: Illustrative result envelope; signing also binds the schema, assurance scope, assumptions and evidence locations.

```
original_repository_id: <stable-repository-id>
original_source_sha: <full-commit-sha>
candidate_manifest_hash: sha256:...
artifact_digests: [sha256:...]
contract_hashes: [sha256:...]
proof_hashes: [sha256:...]
padst_scenario_set: sha256:...
simulation_seed: <seed>
simulation_trace_digest: sha256:...
verifier_workflow_sha: <full-commit-sha>
result: <pass|fail|unknown>
expires_at: <timestamp>
```

GitHub artifact attestations and SLSA provenance provide relevant identity/subject machinery. The behavioural verdict is an additional governed assertion with its own verification policy, not an automatic interpretation of build provenance. [11,12]

18 Governance of the judging criteria

AI agents and developers may write code and propose proofs. They must not silently weaken the laws, observations or assumptions against which their changes are judged. Protect Allium, generated-law sources, semantic IR definitions, translators, observation functions, policy and verifier workflows. Use CODEOWNERS and enforced review rules; where multiple approval domains are required, enforce each explicitly—listing several owners on one CODEOWNERS line does not require approval from all of them. [54]

Table 28: Governance of the judging criteria

Protected change	Accountable review
Business invariants, preconditions and permitted traces	Domain owner; separate from implementation author.
Authorization laws, principal/role mapping, policy freshness	InfoSec plus the accountable authorization owner.
Money movement, FX, corrections and reconciliation	Ledger/accounting owner; approve scales, rounding, rate binding and completeness semantics.
Observation functions and implementation mappings	Independent verification owner; ensure relevant effects cannot be hidden.
Axioms, unsafe operations, foreign code and checker changes	Formal-methods/security owners; scope every admitted assumption and invalidate affected claims.
Deployment constraints and infrastructure assumptions	Platform/security owners; retain exact policy and runtime evidence.

Non-vacuity and mutation discipline

Every important law has feasible positive witnesses and mutations that should be rejected: remove an authorization guard, alter an asset scale, change rounding, hide an effect in the observation function, drop a proof obligation, regress a version comparator, or substitute a stale artifact. Mutations evaluate the gate’s ability to detect a defined defect; they do not prove the original implementation correct.

Compare the required obligation inventory with the baseline, including removed paths and dependencies. Explicitly list all axioms and unsafe operations; an approved exception remains an assumption with a named owner, affected scope and expiry. An implementation author cannot convert “unsupported” to “proven” by editing metadata. Preserve signed provenance and minimized failing traces with access control, retention policy and original witnesses.

Manual release is not proof. Where policy permits break-glass, record the exact degraded manifest, missing/failed obligations, human authority, reason, expiry and follow-up. Never relabel its assurance as passed or allow it to satisfy an automatic promotion gate.

19 Proven, tested, assumed and observed

Table 29: Assurance inventory. “Proven target” identifies an obligation, not a claim that this paper supplies an executed proof for each row.

Concern	Proven target	Tested / observed evidence	Explicit assumptions
Transfer safety	Conservation, constraints and idempotent abstract transition.	Adapter encodings; retry/crash histories; ledger observations.	Ledger semantics, identity mapping and storage recovery.
FX	Per-asset arithmetic, rounding and linked model law.	Real linked chain, liquidity rejection, quote/retry scenarios.	Cluster atomicity, trusted quote source and clock policy.
Corrections	Net-effect and immutable-history model.	Lineage, excess/duplicate reversal and partial-post cases.	Authoritative entitlement allocation and recovery.
Reconciliation	Matching uniqueness and bounded closure algorithm.	Sealed cuts, missing/late/duplicate records and amendments.	Source completeness at declared high-water marks.

Table 29: Assurance inventory. “Proven target” identifies an obligation, not a claim that this paper supplies an executed proof for each row.

Concern	Proven target	Tested / observed evidence	Explicit assumptions
UI semantics	Reducer/view and version-observation relations where formalized.	HTMX swaps, Elm ports, Datar SSE, browser/ally behaviour.	Transport authenticity, browser and adapter semantics.
Authorization	Separation, threshold and policy-bound transitions.	Expiry, revocation, concurrent approvals and policy outage.	Identity/key validity, membership/policy freshness.
System composition	Affected compatibility and trace laws under E.	protocol-testing and simulation scenarios and real integration boundaries.	Complete proof graph, supported fault model and environment.
Deployment	Manifest/policy admission logic if verified.	Signed provenance, exact artifact rollout and telemetry.	Trusted verifier, signing identity and deployment controller.

Production monitors assumption drift

Telemetry should watch policy/config digests, key and identity-provider state, clock skew, queue redelivery, CDC lag and gaps, projection-version anomalies, reconciliation exceptions and actual deployment digests. An observed violation triggers the appropriate containment, re-verification or rollback process. Monitoring can detect breaches of E; a quiet dashboard cannot establish E for every unobserved instant or future execution.

Keep actual execution receipts separate from the architectural claims in this paper. A signed “pass” must state its formal subject, tested participants, scenario/fault bounds, and time of observation. Reviewers should be able to answer: which property, about which exact system, by which method, under which assumptions?

20 Software entropy and exergy: controlling uncertainty and preserving useful assurance

20.1 Origins, scope and corrections to the analogy

Physical exergy is useful work available relative to a specified reference environment; irreversible physical processes destroy it. Exergy analysis therefore depends on both system boundaries and environmental conditions. NASA’s systems-engineering treatment supplies a useful engineering reference, but does not establish a thermodynamic law of software correctness. Here, *assurance exergy* is a proposed policy metric: the useful assurance retained for an approved obligation inventory, reference manifest, observation map and fault model. Its units are policy points, not joules. A known-safe software state is a recovery target, not a thermodynamic dead state. Running a correct application does not imply equilibrium. [55]

The analogy motivates concrete questions: can an operation’s identity, authorization, debit/credit effects and corrective lineage still be reconstructed? Can a failed projection be rebuilt within a declared recovery envelope? It does not replace the five assurance classes, refinement obligations or independent approval of business intent. Allium records specifications; it does not prove them. Bend checks selected formal obligations. Testing examines implementations within its explored scope.

Table 30: Useful correspondences and their limits

Concept	Proposed engineering interpretation
System and environment	Exact manifest plus declared infrastructure, identity, ledger, network and operational assumptions.
Reference state	Approved baseline and lawful recovery target; neither is automatically the current branch head.
Useful assurance	Applicable evidence with preserved provenance, observation and recovery scope.
Loss or destruction	Positive per-obligation degradation under a fixed comparison policy. Gains elsewhere cannot cancel it.

Table 30: Useful correspondences and their limits

Concept	Proposed engineering interpretation
Residual potential	A policy-score shortfall. It is not a measured defect probability, energy balance or count of hidden bugs.

20.2 Entropy is uncertainty, not incorrectness

Shannon entropy requires a random variable and probability distribution, not merely a set of traces. If a justified distribution over a bounded random trace T exists, $H(T \mid S, E)$ is conditional uncertainty given specification S and evidence E . It is not generally computable for an arbitrary distributed implementation. Multiple permitted outcomes can retain positive entropy while every outcome is correct; a deterministic, certainly incorrect implementation can have zero entropy. Conditioning on approved intent does not make the implementation obey it. [56]

In particular, for an estimated violation probability p , binary entropy is

$$h_2(p) = -p \log_2 p - (1-p) \log_2 (1-p), \quad h_2(0) = h_2(1) = 0. \quad (20)$$

Minimizing this quantity can favor certainty of failure. It must not be a safety merge gate. Where calibrated probabilities exist, expected loss $\sum_o w_o p_o$ is a different quantity; unknown probabilities remain unknown, rather than defaulting to zero. An assumed environment does not eliminate uncertainty about whether that environment will hold.

Similarly, $H(\Omega_{\text{implementation}}) - H(\Omega_{\text{evidenced}})$ is undefined without compatible random variables, distributions and conditioning. This implementation does not publish that difference or assert that residual Shannon entropy approaches zero. Its enforceable objects are declared obligations, remaining evidence gaps, assumption inventories and mutation outcomes. New specified FX behavior can legitimately enlarge a state space; only a separately approved comparison policy can decide its required assurance.

Fotínós and Cabral’s 2026 preprint proposes a statistical-mechanics account of software entropy, with test constraints on implementation space and mutation analysis as a local approximation. That is a relevant emerging research direction, not a validated system-risk estimator or an established engineering standard. Mutation operators explore selected local syntactic changes, not all possible implementations or all production faults. [57]

20.3 Mutation adequacy with a protected denominator

Let $\mathcal{M}_o$ be the independently approved, executable mutant inventory for obligation o , with positive weights v_m . Let K_o and V_o denote killed and surviving mutants. Define a weighted survival ratio:

$$U_M(o) = \frac{\sum_{m \in V_o} v_m}{\sum_{m \in \mathcal{M}_o} v_m}. \quad (21)$$

This is an adequacy diagnostic, not Shannon entropy. Its denominator must be nonempty. A timeout, compiler error, infrastructure failure or unexecuted mutant is not a killed behavioral mutant. An equivalent mutant requires an independently reviewed exclusion and new inventory; a candidate cannot declare equivalence to improve its own score. Missing or additional inventory entries block the proposed comparison. Correlated and redundant mutants can inflate adequacy, so report operators, scope and excluded cases alongside the ratio. [58]

Financial operators should target stable transfer IDs, debit/credit direction, ledger and asset scale, pending resolution, linked-chain flags, authorization predicates, rounding, reconciliation matching and projection versions. Killing one ID-changing mutant does not eliminate every possible retry defect. A proof eliminates a modeled counterexample class only when the corresponding universal statement has actually been checked; testing an implementation mutant remains a separate result.

20.4 A policy score that cannot compensate away critical losses

Freeze a comparison frame F : reference manifest, candidate manifest, environmental assumptions, affected obligation closure, observation functions, scenario/mutant inventories, weights and score policy. For each obligation, retain consequence weight $w_o > 0$, evidence credit $e_o \in [0, 1]$ and scoped recovery credit $r_o \in [0, 1]$. Then the

proposed diagnostic is

$$X_A(F) = \sum_{o \in O_F} w_o e_o r_o, \quad X_P(F) = \sum_{o \in O_F} w_o, \quad A(F) = X_P(F) - X_A(F). \quad (22)$$

Only when $X_P > 0$ is $\eta_A = X_A/X_P$ defined. This is a policy-weighted coverage/quality score. It does not measure risk converted into certainty. Evidence classes are not interchangeable numbers: a bounded exhaustive model, sampled protocol test and production observation concern different subjects and assumptions. The earlier illustrative ranking from 1.00 for connected proof to 0.15 for review is not adopted as a universal ordering. Even a connected machine-checked proof retains specification and trusted-computing-base assumptions.

For an illustrative policy, the policy assigns a clearly illustrative ceiling of 700/1000 to each semantic-simulation obligation. Earned evidence credit is that ceiling multiplied by $1 - U_M$, rounded down to integer thousandths. Recovery credit is 1000 only when every required scenario reports recovery, attributable trace and matching replay, otherwise zero. These conservative implementation rules make the score reproducible; they do not calibrate it scientifically. A score below potential can coexist with a policy pass and must remain visible as a shortfall.

Baseline and candidate must use the same approved frame. Define non-offsettable loss:

$$D_F = \sum_{o \in O_F} \max\{0, w_o e_o^{\text{base}} r_o^{\text{base}} - w_o e_o^{\text{cand}} r_o^{\text{cand}}\}. \quad (23)$$

For each protected obligation, require zero individual loss, all required critical mutants killed, and all required recovery observations satisfied. Also require $D_F \leq B_F$ across the declared closure. New obligations, retired behavior and changed weights require a reviewed baseline transition; an unrelated easy obligation cannot dilute a hard obligation or conceal its removal. Policy changes invalidate composition evidence.

The proposed identity “total potential = usable assurance + residual anergy + destroyed assurance” would double-count loss because $A = X_P - X_A$ already includes the candidate shortfall. No software conservation law follows from these definitions. Report X_P , X_A , A and D_F separately. Likewise, a final-state score cannot establish that a trace was safe at intermediate states.

20.5 Recoverability, safety and conditional progress

Existence of a recovery path is weaker than a strategy that works against the admitted environment. Let I_{hard} be protected safety, Q a quiescent operational target, $\mathcal{F}$ the declared fault model, $\Pi(C)$ strategies over permitted corrective operations C . The intended obligations have the form

$$\forall \tau \in \text{Traces}(M, E, \mathcal{F}) : \quad \mathbf{G}I_{\text{hard}}(\tau), \quad (24)$$

$$\exists \pi \in \Pi(C) \forall \tau \in \text{Traces}(M, \pi, E, \mathcal{F}) : \quad \text{Fair}_E(\tau) \Rightarrow \mathbf{G}(\text{RecoveryNeeded} \Rightarrow \mathbf{F}Q). \quad (25)$$

Here $\mathbf{G}$ means always and $\mathbf{F}$ means eventually. A bounded deadline requires additional delivery, scheduling, availability and repair bounds. Repeated faults can prevent convergence indefinitely. A PADST run returning to Q establishes one explored recovery trace, not this universal strategy property. Safety and liveness must remain separately stated. [59]

Legitimate financial commitment is often intentionally irreversible. The requirement is not that every payment can be undone: it is that identity, authorization, accounting history and any lawful correction or escalation remain attributable under declared constraints. Restoring an internal accounting position does not reverse an external settlement. An approved escalation preserves responsibility but does not itself restore money or resolve a pending ledger reservation.

20.6 Financial and UI examples

Transfer and pending counters. Model four nonnegative counters per account: D_p, C_p, D_r, C_r for posted debits/credits and pending debits/credits. A same-ledger transfer of unsigned amount a appends a to the source’s posted debits and destination’s posted credits; it does not store a negative account balance. Pending creation increases the corresponding pending counters. Posting and voiding resolve the reservation according to Tiger-Beetle’s API; a partial post releases the remainder. Signed net observations are explicitly derived views. Per-ledger conservation constrains corresponding aggregate debit and credit movements, separately for posted and pending effects.

A response lost after commit must lead to lookup or retry with the same transfer identity. The protected observation includes operation ID, transfer ID, ledger, amount and account mapping. Creating a new ID on retry

may conserve debits and credits while producing a second, unauthorized business effect: conservation alone is insufficient. No recovery score can excuse that violation.

TigerBeetle pending transfers can post, void or expire when configured with a timeout. Expiration disallows a later manual post or void; removal of expired pending balances is not guaranteed at the exact expiry instant. Thus a ledger liveness claim includes expiration and explicitly declared cluster/operational progress assumptions. A business escalation is recorded separately. The proposed drill should distinguish pending/void from timeout behaviour; a pending/void result alone does not certify a timeout service. [27]

FX. The USD and INR legs occupy different ledgers, using four accounts and a single linked chain. For the paper's example, the source leg is 10,000 USD minor units and the destination leg 824,214 INR minor units. Each ledger conserves its own debits and credits; their numeric amounts must not be added as one conserved asset. In the rejection drill, insufficient INR liquidity must prevent both transfer creations. Removing linkage permits the USD leg to commit and is detected. Quote identity, asset scale, expiry, rounding and explicit fees remain separate obligations; linked atomicity does not establish commercial rate correctness. [33]

Corrections and reconciliation. Original transfers remain immutable. Corrective transfers append a lineage satisfying the intended net accounting effect under permitted limits. A recovery procedure must not silently replace the original operation or obscure a duplicate. Reconciliation closes only over immutable, hashed, bounded source cuts whose inventory is complete for the declared scope. Every item must be matched or assigned an accountable approved terminal exception, such as escalation. A difference classified as escalated is explained, not necessarily economically resolved; outstanding exposure, owner and deadline remain visible.

UI, identity and migration. A stale projection may recover by replay while the authoritative ledger never violates safety. An authoritative Datastar patch needs aggregate identity, operation identity, monotonic version and stream/sequence binding; matching sequence with conflicting payload must fail. An elevation banner or client signal is not authorization evidence. Recovery drills should include SSE reconnect, policy change during approval and interrupted migrations. Their absence is an explicit coverage gap, not something the three financial simulator drills can hide.

20.7 Artifacts, cross-repository gates and replication

A proposed recoverability policy pins a reference manifest, system evidence requirements, per-rule inventories, integer weights, baseline credits and a loss budget. The system receipt should bind a content-addressed report to the candidate manifest. Signature, freshness, build-provenance, assurance-class, execution-mode and participant checks still apply. Declared assumptions, traces, state snapshots and fault schedules must exist at their recorded hashes.

The verifier must recompute the score and hard gates rather than accepting a candidate's precomputed total. Missing required evidence blocks admission. A changed policy invalidates the corresponding composition evidence. Protected reviewers approve baseline values and weights; a signature cannot make a false observation true.

A replication campaign should capture typed requests and all four ledger counters, inject response loss after commitment, run the declared recovery procedure, replay the same schedule and compare traces. It should also execute the corresponding mutants. Label simulated-adaptor evidence separately from native-ledger integration and exact-artifact execution.

A finite admission model can check that admission requires the declared coverage and protection inputs. This does not prove a quantitative evaluator or financial adaptor correct. Independently test missing reports, altered manifests, changed assumptions, unsafe trace prefixes, survivor substitution and score offsetting.

Publish an immutable input inventory, tool versions, permitted fixtures and a runnable driver before reporting campaign results. Keep failed runs and per-command outputs. A public reproduction command must not depend on inaccessible repositories, private pull requests or an unrecorded workstation layout. This edition describes that discipline without redistributing private operational evidence.

For a purely illustrative score, potential credit of 300 million units and earned credit of 210 million leave 90 million residual units. Such a number is policy accounting, not a measured probability of correctness; it cannot replace a critical invariant or missing recovery observation.

For scenario selection, consequence and irrecoverability can guide a reviewed priority order. A product $\Pr(\tau) \times \text{Consequence}(\tau) \times \text{Irrecoverability}(\tau)$ is an expected-loss heuristic only with justified probabilities and compatible scales. Fault injection frequency is not a production probability. Keep mandatory critical scenarios plus exploration outside the ranking so unknown or underestimated rare failures are not excluded.

20.8 Threats to validity and adoption limits

Weights, ceilings and baseline judgments are subjective and sensitive to decomposition. Splitting an obligation can inflate potential; reusing correlated evidence can double-count confidence. Freeze the semantic inventory, preserve the per-obligation vector and test sensitivity to policy choices. Noncritical scores must never override critical failures, unavailable evidence or undeclared environmental premises.

Mutation results depend on operator quality and scope. Replay depends on the controlled substrate, recorded inputs and implementation revision; a seed alone is insufficient. A successful simulator recovery cannot establish a production repair deadline, external bank truth, SaaS availability or pixel correctness. An explainability claim needs domain checks for causation and lineage, not merely nonempty logs. Loss of cryptographic keys, storage history or observation integrity can invalidate previously useful evidence.

The contribution is therefore an implementable assurance-quality and recoverability discipline, not a new proof calculus. Formal reasoning establishes the declared modeled properties; the proof graph composes scoped contracts; controlled protocol tests expose implementation divergence; governance prevents score and denominator manipulation. Entropy language is retained as a research motivation, while the actual gates use explicit obligations and non-offsettable losses.

21 Limitations and research evaluation

21.1 Threats to validity

Intent and translation. A correct proof of the wrong law remains wrong for the business. Allium's behavioural authority depends on approval and precise selected semantics. The proposed translator, source maps and IR need their own correctness argument; deterministic output is insufficient.

Trusted computing base. The proof kernel, compiler/runtime, adapters, libraries, CI runner, artifact store, signing identity and deployment controller can invalidate the connection. Bend's stated maturity limitations make independent qualification and narrow adoption important. Version pinning is necessary, not a proof of those dependencies. [9]

Abstraction and coverage. A simulated dependency may omit production behaviour; a "real node" wrapper may not execute the deployed path; an observation function can erase a violation. State explosion limits trace exploration. Finite controls and selected seeds cannot establish an unbounded universal claim.

Liveness and external truth. Availability depends on fairness, recovery and delivery assumptions. An external bank feed may be incomplete, a quote may be wrong, or an identity provider may be unavailable. Tests cannot remove these premises. Browser semantic correctness does not imply visual/usability correctness.

Evidence and governance. Signatures cannot repair a dishonest or compromised verifier. Incomplete impact graphs, stale caches, weak approval rules and untrusted workflow execution can create convincing but inapplicable evidence. Private source inspection is not independent experimental replication.

21.2 Public evaluation programme

Evaluate a narrow payment journey against a conventional CI baseline. Seed defects across numeric conversion, authorization, duplicate commands, linked-chain construction, CDC ordering, SSE regressions and reconciliation cuts. Record detection by formal checks, local tests, protocol scenarios, deterministic simulation and real integration separately. Retain undetected defects and false positives rather than collapsing them into one pass rate.

Measure obligation and real-adaptor coverage, mutation survival, reproducibility of failing traces, reduction effectiveness, proof maintenance effort, verification latency, graph-invalidation precision and operational escapes. Repeat at fixed manifest versions and compare samples with comparable fault budgets. Publish the corpus, search bounds and exclusions where confidentiality permits. This paper supplies the architecture and evaluation design; it does not invent performance or correctness measurements for a completed deployment.

The included arithmetic example is a deliberately narrow formal core. Its local check report identifies the supplied laws, example execution and negative control exercised for this edition. It does not establish full-language translation, cross-repository release certification, production connectivity or native ledger integration.

Native integration studies should pin a disposable server and client, record source and executable identities, and test both sides of a historical cutoff. Keep these observations separate from simulation, durability and failover claims. Publish runnable artifacts and actual results before claiming execution.

22 Adoption and conclusion

Table 31: Adoption and conclusion

Stage	Exit criterion
1 · Bind one journey	Domain, security and accounting owners approve a precise behavioural contract and observation map.
2 · Prove a small core	Pinned laws/proofs check; non-vacuity and seeded mutants are exercised; trust inventory is explicit.
3 · Connect production	Verified core/translation or clearly labelled conformance relation reaches real adapter paths.
4 · Compose and perturb	Affected edges and trace obligations evaluated; protocol-testing and simulation evidence states real/simulated coverage.
5 · Govern the manifest	Independent required checks, least privilege, signed receipts and exact-digest deployment operate.
6 · Expand with evidence	Extend FX, reconciliation, corrections and UI/security journeys using measured gap and maintenance data.

22.1 A layered assurance system

The complete approach is not “formal proof versus testing.” Allium records approved intent. Bend proves mathematical properties of the formal core. Cross-repository assume-guarantee reasoning composes component contracts. PADST exercises real implementations using protocol-aware stateful traces, while the simulation harness makes the supported distributed execution deterministic, fault-controlled and replayable.

Integration tests validate external boundaries. Production telemetry checks whether environmental assumptions continue to hold where they can be observed; it cannot establish them universally. Governance prevents agents or developers from weakening invariants, preconditions, observations, assumptions, authorization, reconciliation completeness, asset scales, rounding, FX rate binding or deployment constraints merely to make CI pass.

Strongest defensible system claim. For the exact component versions and artifacts named in the approved system manifest, the formal core satisfies the declared laws; affected repository contracts compose under the stated assumptions; and the deployed implementations have passed protocol-aware adversarial traces in a deterministic execution environment.

That claim is earned only when the required implementation connections and execution capabilities are established for the named manifest. The proposed architecture is a route toward it, not evidence that the reference platform already satisfies it. It must not be shortened to “the whole platform is mathematically proven.”

Appendix: Complete runnable model and proofs

These are the same source files consumed by `make research-examples`; they are included directly into this document rather than maintained as separate listings. The natural-number model is deliberately small. The fixture amounts are bounded examples, while the checked laws quantify over all states represented by the model under the checker assumptions.

Listing 10: `Balance.bend`: total reservation function and its conservation proposition.

```
import Base

type Decision is Data:
  Accepted{remaining: Nat}
  Rejected{}

def reserve(request: Nat, remaining: Nat) -> Decision:
  match request:
    case 0n:
      Accepted{remaining}
    case 1n+r:
      match remaining:
        case 0n:
          Rejected{}
        case 1n+b:
          reserve(r, b)

def Conserves(outcome: Decision, -request: Nat, -remaining: Nat) -> Type:
  match outcome:
    case Accepted{left}:
      {Nat.add(request, left) == remaining : Nat}
    case Rejected{}:
      Unit
```

Listing 11: `Payment.bend`: the executable single-operation transition function and total-balance observation.

```
import Base
import ./Balance.bend as Balance

type Payment is Data:
  Pending{source: Nat, destination: Nat}
  Committed{source: Nat, destination: Nat}
  Rejected{source: Nat, destination: Nat}

def finish(outcome: Balance.Decision, amount: Nat,
           source: Nat, destination: Nat) -> Payment:
  match outcome:
    case Balance.Accepted{left}:
      Committed{left, Nat.add(amount, destination)}
    case Balance.Rejected{}:
      Rejected{source, destination}

def apply(+amount: Nat, state: Payment) -> Payment:
  match state:
    case Pending{+source, +destination}:
      finish(Balance.reserve(amount, source), amount, source, destination)
    case Committed{source, destination}:
      Committed{source, destination}
    case Rejected{source, destination}:
      Rejected{source, destination}

def total(state: Payment) -> Nat:
  match state:
```

```

    case Pending{source, destination}:
      Nat.add(source, destination)
    case Committed{source, destination}:
      Nat.add(source, destination)
    case Rejected{source, destination}:
      Nat.add(source, destination)

type RetryReply is Data:
  Refused{state: Payment}
  Replayed{state: Payment}

def retry(state: Payment) -> RetryReply:
  match state:
    case Pending{source, destination}:
      Refused{Pending{source, destination}}
    case Committed{source, destination}:
      Replayed{Committed{source, destination}}
    case Rejected{source, destination}:
      Replayed{Rejected{source, destination}}

def retry_state(reply: RetryReply) -> Payment:
  match reply:
    case Refused{state}:
      state
    case Replayed{state}:
      state

def retry_accepted(reply: RetryReply) -> Bool:
  match reply:
    case Refused{state}:
      False{}
    case Replayed{state}:
      True{}

def terminal(state: Payment) -> Bool:
  match state:
    case Pending{source, destination}:
      False{}
    case Committed{source, destination}:
      True{}
    case Rejected{source, destination}:
      True{}

type Observation is Data:
  Snapshot{source: Nat, destination: Nat, committed_effects: Nat}

def observe(state: Payment) -> Observation:
  match state:
    case Pending{source, destination}:
      Snapshot{source, destination, 0n}
    case Committed{source, destination}:
      Snapshot{source, destination, 1n}
    case Rejected{source, destination}:
      Snapshot{source, destination, 0n}

```

Listing 12: PROOF.bend: complete proof terms for the six laws, including the supporting arithmetic lemmas. Checked by Bend 2.0.16.

```
import Base
import ./Balance.bend as Balance
import ./Payment.bend as Payment
import ./LAWS.bend as Laws

def lift(outcome: Balance.Decision, -amount: Nat, -source: Nat,
        proof: Balance.Conserves(outcome, amount, source)) ->
    Balance.Conserves(outcome, 1n+amount, 1n+source):
  match outcome:
    case Balance.Accepted{left}:
      %proof : {1n+Nat.add(amount, left) == 1n+ : Nat}
      {==}
    case Balance.Rejected{}:
      Unit{}

def Laws.reservation_conserve(amount, source):
  match amount:
    case 0n:
      {==}
    case 1n+ +rest:
      match source:
        case 0n:
          Unit{}
        case 1n+ +remaining:
          lift(Balance.reserve(rest, remaining), rest, remaining,
              Laws.reservation_conserve(rest, remaining))

def finish_idempotent(outcome: Balance.Decision, amount: Nat,
                     source: Nat, destination: Nat) ->
    {Payment.apply(amount, Payment.finish(outcome, amount, source, destination)) ==
     Payment.finish(outcome, amount, source, destination) : Payment.Payment}:
  match outcome:
    case Balance.Accepted{left}:
      {==}
    case Balance.Rejected{}:
      {==}

def Laws.duplicate_submit_idempotent(amount, state):
  match state:
    case Payment.Pending{+source, destination}:
      finish_idempotent(Balance.reserve(amount, source), amount, source, destination)
    case Payment.Committed{source, destination}:
      {==}
    case Payment.Rejected{source, destination}:
      {==}

def add_successor(a: Nat, +b: Nat) ->
    {1n+Nat.add(a, b) == Nat.add(a, 1n+b) : Nat}:
  match a:
    case 0n:
      {==}
    case 1n+ +p:
      %add_successor(p, b) : {1n+1n+Nat.add(p, b) == 1n+ : Nat}
      {==}

def reassociate(a: Nat, +b: Nat, +c: Nat) ->
    {Nat.add(a, Nat.add(b, c)) == Nat.add(Nat.add(a, b), c) : Nat}:
  match a:
    case 0n:
      {==}
    case 1n+ +p:
      %reassociate(p, b, c) :
        {1n+Nat.add(p, Nat.add(b, c)) == 1n+ : Nat}
```

```

{==}

def swap_add(a: Nat, +b: Nat, +c: Nat) ->
{Nat.add(a, Nat.add(b, c)) == Nat.add(b, Nat.add(a, c)) : Nat}:
match a:
  case 0n:
    {==}
  case 1n+ +p:
    %add_successor(b, Nat.add(p, c)) :
    {1n+Nat.add(p, Nat.add(b, c)) == _ : Nat}
    %swap_add(p, b, c) :
    {1n+Nat.add(p, Nat.add(b, c)) == 1n+_ : Nat}
    {==}

def finish_conserve(outcome: Balance.Decision, +amount: Nat,
                    source: Nat, +destination: Nat,
                    proof: Balance.Conserves(outcome, amount, source)) ->
{Payment.total(Payment.finish(outcome, amount, source, destination)) ==
 Nat.add(source, destination) : Nat}:
match outcome:
  case Balance.Accepted{+left}:
    %proof : {Nat.add(left, Nat.add(amount, destination)) ==
              Nat.add(_, destination) : Nat}
    %reassociate(amount, left, destination) :
    {Nat.add(left, Nat.add(amount, destination)) == _ : Nat}
    swap_add(left, amount, destination)
  case Balance.Rejected{}:
    {==}

def Laws.transfer_conserve(amount, state):
match state:
  case Payment.Pending{+source, +destination}:
    finish_conserve(Balance.reserve(amount, source), amount,
                    source, destination,
                    Laws.reservation_conserve(amount, source))
  case Payment.Committed{source, destination}:
    {==}
  case Payment.Rejected{source, destination}:
    {==}

def Laws.retry_preserves_state(state):
match state:
  case Payment.Pending{source, destination}:
    {==}
  case Payment.Committed{source, destination}:
    {==}
  case Payment.Rejected{source, destination}:
    {==}

def Laws.retry_requires_terminal(state):
match state:
  case Payment.Pending{source, destination}:
    {==}
  case Payment.Committed{source, destination}:
    {==}
  case Payment.Rejected{source, destination}:
    {==}

def Laws.observed_retry_stable(state):
%Laws.retry_preserves_state(state) :
{Payment.observe(Payment.retry_state(Payment.retry(state))) ==
 Payment.observe(_ : Payment.Observation)}
{==}

```

Listing 13: cases.json: concrete input data and expected states used by the executable fixture checks. Every amount is an integer number of USD cents.

```
{
  "asset": "USD",
  "scale": 2,
  "scope": "Concrete test fixtures for a single operation; not production transactions",
  "cases": [
    {
      "id": "first_submit",
      "operation_id": "demo-usd-001",
      "source": 125,
      "destination": 25,
      "expected": {
        "status": "Committed",
        "source": 25,
        "destination": 125
      },
      "amount": 100,
      "events": [
        {
          "kind": "submit"
        }
      ]
    },
    {
      "id": "retry_after_commit",
      "operation_id": "demo-usd-001",
      "source": 125,
      "destination": 25,
      "expected": {
        "status": "Committed",
        "source": 25,
        "destination": 125
      },
      "amount": 100,
      "events": [
        {
          "kind": "submit"
        },
        {
          "kind": "retry"
        }
      ]
    },
    {
      "id": "insufficient_funds",
      "operation_id": "demo-usd-002",
      "source": 125,
      "destination": 25,
      "expected": {
        "status": "Rejected",
        "source": 125,
        "destination": 25
      },
      "amount": 200,
      "events": [
        {
          "kind": "submit"
        }
      ]
    },
    {
      "id": "retry_after_rejection",
      "operation_id": "demo-usd-002",
      "source": 125,
```

```

    "destination": 25,
    "expected": {
      "status": "Rejected",
      "source": 125,
      "destination": 25
    },
    "amount": 200,
    "events": [
      {
        "kind": "submit"
      },
      {
        "kind": "retry"
      }
    ]
  },
  {
    "id": "retry_before_submit",
    "operation_id": "demo-usd-003",
    "source": 125,
    "destination": 25,
    "amount": 100,
    "events": [
      {
        "kind": "retry"
      }
    ],
    "expected": {
      "status": "Pending",
      "source": 125,
      "destination": 25
    }
  },
  {
    "id": "duplicate_submit_model",
    "operation_id": "demo-usd-004",
    "source": 125,
    "destination": 25,
    "amount": 100,
    "events": [
      {
        "kind": "reduce_submit"
      },
      {
        "kind": "reduce_submit"
      }
    ],
    "expected": {
      "status": "Committed",
      "source": 25,
      "destination": 125
    },
    "scope": "algebraic reducer property; not credited as RetryPayment"
  }
]
}

```

References

- [1] JUXT. Allium v2 language reference. Official v2 tag, commit 346440ca338c6e0d6cf375ddd41f4278ec6382a0. Retrieved from the versioned source; original v2 website URL was not retrievable. <https://github.com/juxt/allium/blob/346440ca338c6e0d6cf375ddd41f4278ec6382a0/references/language-reference.md>. Accessed 20 September 2026.
- [2] JUXT. Migrating from Allium v2 to v3. Official language migration reference. <https://github.com/juxt/allium/blob/main/skills/allium/references/migration-v2-to-v3.md>. Accessed 20 September 2026.
- [3] JUXT. Allium v3 language reference. Pinned source revision 4870af97e4d742c66158e82a44fabec00d7a120a. <https://github.com/juxt/allium/blob/4870af97e4d742c66158e82a44fabec00d7a120a/skills/allium/references/language-reference.md>. Accessed 20 September 2026.
- [4] Abadi, Martín, and Leslie Lamport. The Existence of Refinement Mappings. Digital Systems Research Center Report 29, 1988; published in Theoretical Computer Science 82(2), 1991, pp. 253–284. <https://lamport.azurewebsites.net/pubs/abadi-existence.pdf>. Accessed 20 September 2026.
- [5] Lamport, Leslie. Specifying Systems: The TLA+ Language and Tools for Hardware and Software Engineers. Addison-Wesley, 2002. <https://lamport.azurewebsites.net/tla/book.html>. Accessed 20 September 2026.
- [6] Herlihy, Maurice P., and Jeannette M. Wing. Linearizability: A Correctness Condition for Concurrent Objects. ACM TOPLAS 12(3), 1990, pp. 463–492. <https://www.cs.cmu.edu/~wing/publications/HerlihyWing90.pdf>. Accessed 20 September 2026.
- [7] Abadi, Martín, and Leslie Lamport. Composing Specifications. ACM Transactions on Programming Languages and Systems 15(1), 1993, pp. 73–132. <https://www.microsoft.com/en-us/research/wp-content/uploads/2016/12/Composing-Specifications.pdf>. Accessed 20 September 2026.
- [8] Bend Project. Bend 2 language guide: laws, proofs, termination, quantities, modules and tooling. <https://github.com/bendlang/bend/blob/main/guide/GUIDE.md>. Accessed 20 September 2026.
- [9] Bend Project. Bend README, especially Limitations: unsafe recursion, axioms, compiler audit and formalization mismatch. <https://github.com/bendlang/bend#limitations>. Accessed 20 September 2026.
- [10] JUXT. Allium Tools 3.0.3. Official checker release; supports Allium language versions 1, 2 and 3. <https://github.com/juxt/allium-tools/releases/tag/v3.0.3>. Accessed 20 September 2026.
- [11] GitHub. Using artifact attestations to establish provenance for builds. Official attestation guidance. <https://docs.github.com/en/actions/how-tos/secure-your-work/use-artifact-attestations/use-artifact-attestations>. Accessed 20 September 2026.
- [12] SLSA Project / Linux Foundation. SLSA v1.2 Provenance. Approved provenance specification and build/-source distinctions. <https://slsa.dev/spec/v1.2/provenance>. Accessed 20 September 2026.
- [13] Hypothesis Project. Stateful tests: rule-based state machines and generated action sequences. Official documentation. <https://hypothesis.readthedocs.io/en/latest/stateful.html>. Accessed 20 September 2026.
- [14] P Project. P: state-machine modelling and systematic testing of distributed systems. Official documentation. <https://p-org.github.io/P/>. Accessed 20 September 2026.
- [15] Dragoi, Cezara; Constantin Enea; Srinidhi Nagendra; and Mandayam Srivas. A Domain Specific Language for Testing Consensus Implementations. arXiv:2303.05893v2, 2023. <https://arxiv.org/abs/2303.05893v2>. Accessed 20 September 2026.
- [16] TigerBeetle. Change Data Capture. Official AMQP publication, message-format, replay and version guidance. <https://docs.tigerbeetle.com/operating/cdc/>. Accessed 20 September 2026.
- [17] RabbitMQ. Reliability Guide. Official acknowledgement, recovery and duplicate-delivery guidance. <https://www.rabbitmq.com/docs/reliability>. Accessed 20 September 2026.
- [18] Amazon Web Services. Amazon DynamoDB Transactions: How it works. Official transaction and conditional-write documentation. <https://docs.aws.amazon.com/amazondynamodb/latest/developerguide/transaction-apis.html>. Accessed 20 September 2026.
- [19] TigerBeetle. Safety. Official discussion of testing and fault exploration. <https://docs.tigerbeetle.com/concepts/safety/>. Accessed 20 September 2026.

- [20] Zhou, Jingyu, et al. FoundationDB: A Distributed Unbundled Transactional Key Value Store. ACM SIGMOD, 2021. DOI: 10.1145/3448016.3457559. <https://www.foundationdb.org/files/fdb-paper.pdf>. Accessed 20 September 2026.
- [21] FoundationDB Project. Simulation and Testing. Official FoundationDB documentation. <https://apple.github.io/foundationdb/testing.html>. Accessed 20 September 2026.
- [22] O’Callahan, Robert; Chris Jones; Nathan Froyd; Kyle Huey; Albert Noll; and Nimrod Partush. Engineering Record and Replay for Deployability. USENIX ATC, 2017, pp. 377–389. <https://www.usenix.org/conference/atc17/technical-sessions/presentation/ocallahan>. Accessed 20 September 2026.
- [23] Alvaro, Peter; Joshua Rosen; and Joseph M. Hellerstein. Lineage-driven Fault Injection. ACM SIGMOD, 2015, pp. 331–346. DOI: 10.1145/2723372.2723711. <https://people.ucsc.edu/~palvaro/molly.pdf>. Accessed 20 September 2026.
- [24] TigerBeetle. Financial Accounting. Official coding guide. <https://docs.tigerbeetle.com/coding/financial-accounting/>. Accessed 20 September 2026.
- [25] TigerBeetle. Account. Official reference for ledgers, debit/credit counters, history and balance constraints. <https://docs.tigerbeetle.com/reference/account/>. Accessed 20 September 2026.
- [26] TigerBeetle. Reliable Transaction Submission. Official guide to application-level identity and retries. <https://docs.tigerbeetle.com/coding/reliable-transaction-submission/>. Accessed 20 September 2026.
- [27] TigerBeetle. Transfer. Official reference for transfer fields, flags, constraints and immutability. <https://docs.tigerbeetle.com/reference/transfer/>. Accessed 20 September 2026.
- [28] TigerBeetle. Two-Phase Transfers. Official guide to pending, post, void, expiration and partial posting. <https://docs.tigerbeetle.com/coding/two-phase-transfers/>. Accessed 20 September 2026.
- [29] TigerBeetle. Linked Events. Official guide to atomic event chains. <https://docs.tigerbeetle.com/coding/linked-events/>. Accessed 20 September 2026.
- [30] TigerBeetle. get_account_balances. Official request reference, including history and pending-expiration limitations. https://docs.tigerbeetle.com/reference/requests/get_account_balances/. Accessed 20 September 2026.
- [31] TigerBeetle. create_transfers: imported-event timestamps must not regress behind the last transfer committed in the cluster. https://docs.tigerbeetle.com/reference/requests/create_transfers/#import_event_timestamp_must_not_regress. Accessed 21 September 2026.
- [32] TigerBeetle. Tigerlings. Official learning exercise repository; includes history, balance constraints, linked and two-phase transfers. <https://github.com/tigerbeetle/tigerlings>. Accessed 20 September 2026.
- [33] TigerBeetle. Currency Exchange. Official four-account linked-transfer recipe, USD/INR example and fee treatment. <https://docs.tigerbeetle.com/coding/recipes/currency-exchange/>. Accessed 20 September 2026.
- [34] TigerBeetle. Correcting Transfers. Official recipe. <https://docs.tigerbeetle.com/coding/recipes/correcting-transfers/>. Accessed 20 September 2026.
- [35] HTMX Project. Documentation. Request triggers, responses, targets, swaps and synchronization. <https://htmx.org/docs/>. Accessed 20 September 2026.
- [36] HTMX Project. Events. Official request and swap lifecycle event reference. <https://htmx.org/events/>. Accessed 20 September 2026.
- [37] HTMX Project. hx-swap. Official swap modes and timing reference. <https://htmx.org/attributes/hx-swap/>. Accessed 20 September 2026.
- [38] Fielding, Roy T.; Mark Nottingham; and Julian Reschke, editors. HTTP Semantics. RFC 9110, IETF, June 2022. <https://httpwg.org/specs/rfc9110.html>. Accessed 20 September 2026.
- [39] Elm Project. The Elm Architecture. Official guide to Model, Update and View. <https://guide.elm-lang.org/architecture/>. Accessed 20 September 2026.
- [40] Elm Project. Ports. Official JavaScript interoperability guide. <https://guide.elm-lang.org/interop/ports.html>. Accessed 20 September 2026.

- [41] Datastar Project. Reactive Signals. Official guide. https://data-star.dev/guide/reactive_signals. Accessed 20 September 2026.
- [42] Datastar Project. Actions. Official backend-action and transport options reference. <https://data-star.dev/reference/actions>. Accessed 20 September 2026.
- [43] Datastar Project. SSE Events. Official datastar-patch-elements and datastar-patch-signals protocol reference. https://data-star.dev/reference/sse_events. Accessed 20 September 2026.
- [44] WHATWG. HTML Living Standard: Server-sent events. EventSource, event IDs and reconnection semantics. <https://html.spec.whatwg.org/multipage/server-sent-events.html>. Accessed 20 September 2026.
- [45] OpenID Foundation. OpenID Connect Core 1.0 incorporating errata set 2. Identity/token validation specification. https://openid.net/specs/openid-connect-core-1_0.html. Accessed 20 September 2026.
- [46] Cedar Policy Language. Basic Cedar syntax. Official policy reference. <https://docs.cedarpolicy.com/policies/syntax-policy.html>. Accessed 20 September 2026.
- [47] GitHub. Reuse workflows. Official workflow_call, reusable workflow and commit-reference documentation. <https://docs.github.com/en/actions/how-tos/reuse-automations/reuse-workflows>. Accessed 20 September 2026.
- [48] GitHub. Secure use reference. Official workflow hardening and untrusted-input guidance. <https://docs.github.com/en/actions/reference/security/secure-use>. Accessed 20 September 2026.
- [49] GitHub. Events that trigger workflows. Official repository_dispatch, pull_request_target, workflow_run and merge event semantics. <https://docs.github.com/en/actions/reference/workflows-and-actions/events-that-trigger-workflows>. Accessed 20 September 2026.
- [50] GitHub. REST API endpoints for check runs. Official permissions and commit-bound check reporting. <https://docs.github.com/en/rest/checks/runs>. Accessed 20 September 2026.
- [51] GitHub. OpenID Connect. Official short-lived federation and workflow identity overview. <https://docs.github.com/en/actions/concepts/security/openid-connect>. Accessed 20 September 2026.
- [52] GitHub. OpenID Connect reference. Official token claims and trust-policy reference. <https://docs.github.com/en/actions/reference/security/oidc>. Accessed 20 September 2026.
- [53] GitHub. GITHUB_TOKEN. Official repository-scoped token description. https://docs.github.com/en/actions/concepts/security/github_token. Accessed 20 September 2026.
- [54] GitHub. About code owners. Official review semantics and branch-protection integration. <https://docs.github.com/en/repositories/managing-your-repositorys-settings-and-features/customizing-your-repository/about-code-owners>. Accessed 20 September 2026.
- [55] Watson, M. D., Mesmer, B. L., and Farrington, P. A. Engineering Elegant Systems: Theory of Systems Engineering. NASA/TP-20205003644, June 2020, Section 4.1: system exergy and reference environments. https://www.nasa.gov/wp-content/uploads/2018/09/nasa_tp_20205003644_interactive2.pdf. Accessed 22 September 2026.
- [56] Shannon, Claude E. A Mathematical Theory of Communication. Bell System Technical Journal 27 (1948), pp. 379–423 and 623–656. <https://people.math.harvard.edu/~ctm/home/text/others/shannon/entropy/entropy.pdf>. Accessed 22 September 2026.
- [57] Fotinós, Jerónimo, and Cabral, Juan B. Software Entropy: A Statistical Mechanics Framework for Software Testing. arXiv:2603.20528, submitted 20 March 2026. Research preprint; not an engineering standard. <https://arxiv.org/abs/2603.20528>. Accessed 22 September 2026.
- [58] Jia, Yue, and Harman, Mark. An Analysis and Survey of the Development of Mutation Testing. IEEE Transactions on Software Engineering 37(5), 2011, pp. 649–678. DOI: 10.1109/TSE.2010.62. <https://kellogg.github.io/martinjkellogg.com/teaching/cs684-sp24/assets/mutation-testing.pdf>. Accessed 22 September 2026.
- [59] Alpern, Bowen, and Schneider, Fred B. Defining Liveness. Cornell technical report, revised February 1985; Information Processing Letters 21(4), 1985, pp. 181–185. <https://www.cs.cornell.edu/fbs/publications/85-650.pdf>. Accessed 22 September 2026.