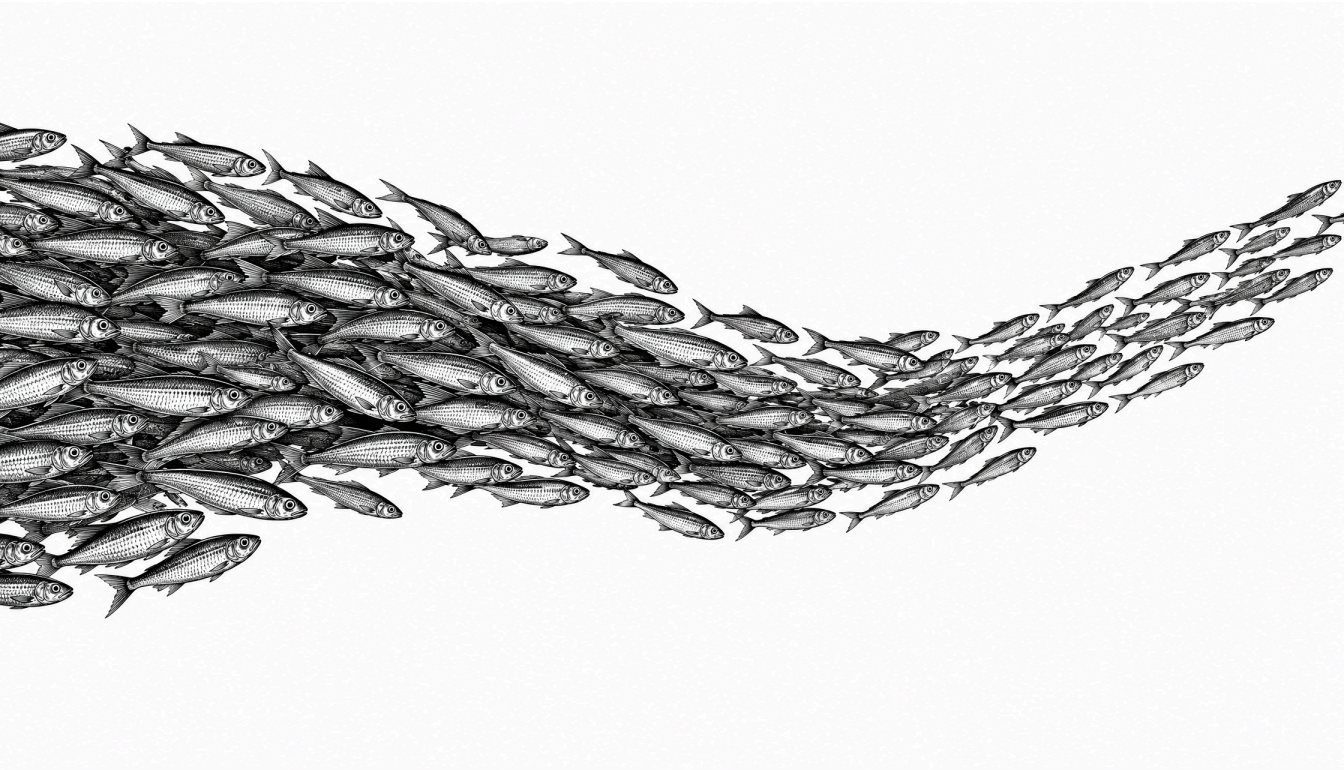


Let It Race

Observability Over Control
in Distributed Systems

Tom Nguyen

Let it race

Observability over control in distributed systems

Tom Nguyen

© 2026 Tom Nguyen. All rights reserved.

No part of this publication may be reproduced, distributed, or transmitted in any form or by any means without the prior written permission of the author.

First Edition: 2026

AI disclosure: The prose was largely generated with AI and revised through an AI-assisted editorial process. Decorative illustrations also use AI-generated artwork. Code and technical diagrams were programmatically checked; these checks are not an independent technical review.

Let It Race

FREE SAMPLE

This sample contains the preface and complete Chapters 1, 6, and 7. It is not the complete book.

Chapter 1 introduces the coordination boundary. Chapters 6 and 7 develop the runnable checkout example through atomic reservation, effect delivery, retries, and recovery.

Page numbers in reproduced chapters refer to the full edition. Use the PDF bookmarks to navigate this sample.

The full edition contains 21 chapters, an epilogue, three appendices, and a glossary. The MIT-licensed companion code is available separately.

Get the complete book

https://leanpub.com/let-it-race

Download the companion code

https://github.com/monotykamary/let-it-race-examples

Preface

I wrote this book to examine a recurring design choice: which work must be coordinated before a system accepts a request, and which work can proceed independently afterward?

Distributed services often separate durable event capture from the views people read. That separation can reduce coupling and keep useful work moving during delays. It also creates obligations: preserve retry identity, expose incomplete views, and recover effects whose outcomes are uncertain.

The book organizes this work around four patterns:

- **Append-only event stores**: Retained observations and explicit corrections
- **Incrementally maintained views**: Derived state with a known source boundary
- **Validation at the gates**: Domain checks bound to commitment when shared invariants require it
- **Aggregate-gated actions**: External effects authorized by explicit evidence and delivered through a retry-safe protocol

Let independent work race. Coordinate where invariants require it. Make lag, uncertainty, and recovery observable. Transactions, reservations, and consensus remain useful at the boundaries that need them. Waiting for a view to stop changing does not establish correctness or finality.

This book is for engineers who design or operate data-intensive services. It assumes familiarity with databases, APIs, and basic concurrency. The examples connect implementation details to product promises and operational recovery, with optional domain analogies that keep their limits explicit.

Part I introduces the decision boundary and the cost of coordination. Part II develops the patterns, including a runnable checkout and retry example shared by Chapters 6 and 7. Part III compares their use in version control, infrastructure, financial systems, and sensors. Part IV covers implementation and migration. Part V considers AI-derived state, design judgment, and hard limits on where the patterns apply. Read the exclusions in Chapter 21 alongside the early examples whenever a decision involves safety, money, or irreversible effects.

How to read the evidence

Documented cases cite public sources at the relevant claims. Author's accounts identify firsthand experience and disclose anonymization; they are attributed observations, not independently audited benchmarks. Hypothetical scenarios are teaching constructions and are labeled before the scene. They do not establish that a company used an architecture or experienced a particular outcome.

Code marked illustrative omits named adapters or surrounding application behavior. Runnable examples state their scope and ship with tests for specific failure sequences. A test demonstrates behavior under its assumptions; it does not certify a production deployment, a financial scheme, or a safety-critical device.

Use these patterns to ask more precise questions of the systems you already operate. Keep the invariant visible while you decide which work can proceed independently.

Tom Nguyen
March 2026

CHAPTER 1

The illusion of control

One tweet, many timelines

Twitter's public engineering talks describe separate paths for accepting tweets and delivering timelines. A tweet can be accepted while work remains to make it visible to its readers. Raffi Krikorian's presentations on timeline delivery explain how the service organized that work at scale.[1]

That separation gives us a concrete design question. Which work must finish before the service acknowledges a write, and which work can happen afterward? A successful acknowledgement needs a defined durability contract. Updating every derived view can follow a different contract, with its own lag and failure handling.

The talks support studying those separate paths. They do not establish a private SXSW incident sequence, a conversation that changed the design overnight, or a claim that strict chronological ordering caused every early outage. This chapter uses the published architecture as context and labels the teaching scenario below separately.

A shared bottleneck

> ***Hypothetical scenario:*** *The following request trace illustrates synchronous work on a shared resource. It is not a reconstruction of a Twitter incident.*

A service accepts a post only after it updates every follower's timeline. During a burst, that work queues behind a shared database bottleneck. The request deadline expires while the service still has work to do, leaving the caller unsure whether retrying will create another post.

Moving timeline delivery to a durable queue can shorten the acknowledgement path. It also creates responsibilities: identify retries, recover unfinished delivery, expose lag, and decide how edits or deletions affect older queued work. The queue does not remove those decisions.

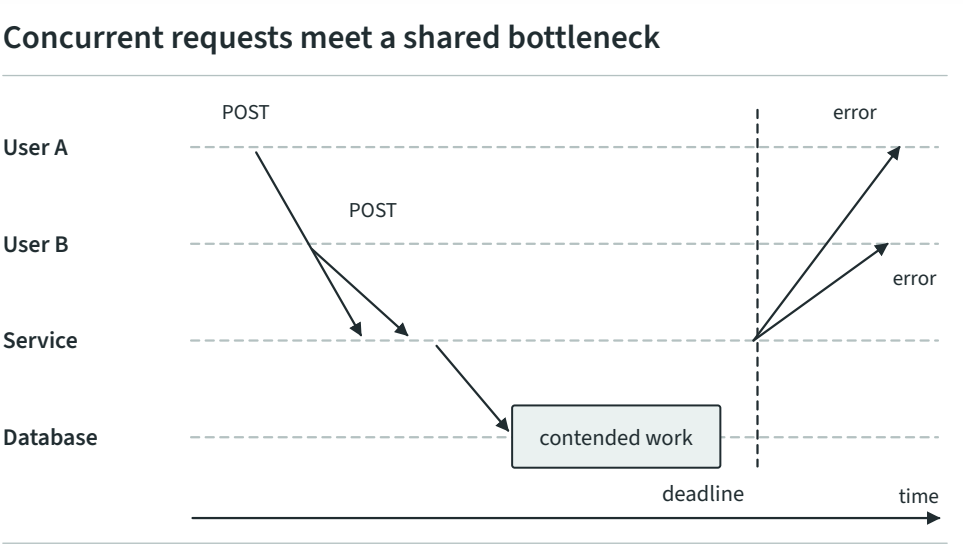


Figure 1.1: A hypothetical request trace. Shared work exceeds a deadline. The figure illustrates a bottleneck and an ambiguous response; it does not diagnose a historical outage.

The boundary comes first

Let independent work race. Coordinate where invariants require it. Make lag, uncertainty, and recovery observable.

An invariant is a property every accepted state must preserve. Inventory may have to remain nonnegative. A logical payment may need one provider effect across retries. A safety controller may have a response deadline that a background view cannot meet.

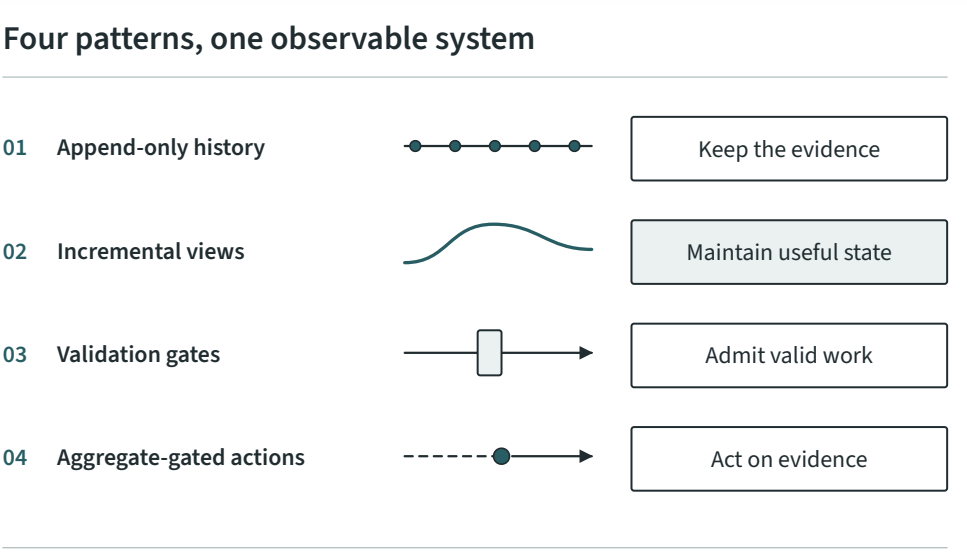
Those requirements determine where concurrency is allowed. Two consumers can build independent reports from the same history. Two buyers cannot both reserve the final item if the product forbids overselling. Reading the same correct history does not arbitrate their competing claims; a conditional commitment or allocation protocol must do that.

Before changing a system, identify:

- The acknowledgement: receipt of an attempt, a committed allocation, or a completed external effect.
- The invariant and the authority that enforces it across all writers.
- The provisional states the product permits and how users recognize them.
- The evidence required for recovery after a timeout or lost acknowledgement.
- Any hard deadline, legal finality rule, or threat model that rules out a proposed path.

Chapter 21 develops these exclusions. Use them from the start. A single unsatisfied hard requirement can disqualify a design for a particular decision.

Four patterns around that boundary



Append-only event stores preserve accepted observations and later corrections. Durable storage, identity, access controls, and retention policy determine which history is available. An append-only shape alone does not guarantee that every real-world event was captured.

Incrementally maintained views turn source data into representations that consumers can read efficiently. A view can be complete through a known source boundary while newer data waits. Publish that boundary and detect missing inputs.

Validation at the gates checks requests against domain rules. Input checks and preflight reads improve feedback. State-dependent hard invariants require the check to be bound to a conditional commitment, as the checkout example in Chapter 6 demonstrates.

Aggregate-gated actions wait for the evidence and authorization needed for an external effect. The effect needs a durable intent and a retry protocol at the destination. An idle timer can reduce churn, but silence does not prove completeness or finality.

These patterns can be combined with transactions and consensus. The design work is choosing their scope and connecting their contracts. This book presents a framework for that work, rather than attributing all four patterns to every system it discusses.

Observation: delayed views are familiar

A post can appear in one reader's feed before another reader's copy catches up. A status dashboard can lag the controller that changed the resource. A financial report can trail a ledger that already committed its entries.

Those delays do not tell us which guarantees the write path provides. Two social feeds can also differ because of personalization or access policy. To diagnose consistency, compare the same logical data under a stated read contract and inspect the source coverage.

A useful interface tells the reader which boundary it has reached. Received, reserved, and shipped describe different commitments. Last refreshed describes a job's timing; processed through offset W describes which source records a view includes.

Mechanism: communication and coordination have costs

Distributed components learn about changes through communication. Propagation takes time, and links can fail. A service must specify whether an operation waits for remote evidence, proceeds provisionally, or rejects the request when that evidence is unavailable.

Under a partition, the CAP result prevents a system from guaranteeing both linearizable behavior and availability for every request in the theorem's model.[2] It does not imply that every application must choose one consistency model for every operation. A system can accept independent observations while refusing an allocation it cannot safely coordinate.

Coordination can add latency and create contention on shared resources. Its cost depends on distance, workload, implementation, and invariant scope. One hot key can serialize requests while unrelated keys continue independently. Measure those conditions before removing a constraint that currently protects the business.

Spanner demonstrates that a distributed database can provide external consistency through an explicit protocol, including bounded clock uncertainty and commit waiting.[3] Chapter 2 examines the costs and benefits of that choice. Strong guarantees are implementable; they must be evaluated against the workload and failure model.

A note on terminology

This book uses architectural names independently of products. An append-only event store preserves observations; an incrementally maintained view updates a derived representation from changes. TimescaleDB calls its managed partitioned tables hypertables and its incremental materialized views continuous aggregates. Vendor-specific examples name their implementation because refresh, retention, and late-data behavior differ.

Git's object graph, a database projection, and an LLM synthesis share useful ideas about retained evidence and derived state. Their guarantees differ. A probabilistic synthesis does not become correct merely because its inputs stopped changing.

Implication: compare complete protocols

A queue-based design must account for duplicate delivery, worker failure, backlog, and recovery. A coordinated design must account for contention, unavailable participants, aborts, and retries. Compare them under the same product promises and load.

Start with a bounded experiment. Preserve the existing authority while a shadow consumer builds a view. Measure divergence and exercise recovery before letting that consumer authorize external effects. Chapter 16 expands this approach into a migration plan.

THOUGHT EXPERIMENT: a receipt in five milliseconds

The setup: Your API receives an order within five milliseconds, but inventory and payment checks can take longer. Users need to know whether retrying will create another order.

The constraint to remove: You cannot make the required checks finish within the acknowledgement budget.

Your task: Define a durable receipt and a stable request identity. Specify how the user learns whether allocation succeeded, how long pending can last, and which authority decides whether the item is theirs.

What you might discover: A fast acknowledgement can be useful without promising completed fulfillment. Its value depends on the recovery and status protocol behind it.

Closure: name one unnecessary dependency

Choose one request path and list the work it waits for. Mark which steps protect an invariant and which produce a view for later consumption.

- Could a derived view update later without changing the acknowledgement's meaning?
- What evidence would expose a stalled consumer or a missing source?
- Which transaction or protocol must remain to prevent incompatible commitments?
- Can a caller recover safely after receiving no response?

Measure one candidate path while preserving its existing safety boundary. Use the result to decide what can move off that path.

---

1. Raffi Krikorian, "Real-Time Delivery Architecture at Twitter" and "Timelines at Scale," presentations published by InfoQ in 2012 and 2013. https://www.infoq.com/presentations/Real-Time-Delivery-Twitter/ [https://www.infoq.com/presentations/Real-Time-Delivery-Twitter/] and https://www.infoq.com/presentations/Twitter-Timeline-Scalability/ [https://www.infoq.com/presentations/Twitter-Timeline-Scalability/] ↩

2. Seth Gilbert and Nancy Lynch, "Brewer's Conjecture and the Feasibility of Consistent, Available, Partition-Tolerant Web Services," SIGACT News 33(2), 2002, pp. 51–59. https://doi.org/10.1145/564585.564601 [https://doi.org/10.1145/564585.564601] ↩

3. James C. Corbett et al., "Spanner: Google's Globally-Distributed Database," OSDI 2012. https://www.usenix.org/conference/osdi12/technical-sessions/presentation/corbett [https://www.usenix.org/conference/osdi12/technical-sessions/presentation/corbett] ↩

CHAPTER 6

Validation at the gates

Two buyers, one item

> ***Hypothetical scenario:*** *The checkout below is a teaching example. It makes no claim about a particular retailer's implementation.*

Two buyers reach checkout while the inventory service reports one item remaining. Both requests read version 12 of the inventory history. Each validator replays that history, finds one item, and approves a purchase. If each request can then append a reservation independently, the store sells two items.

Reading the complete history did not prevent this failure. Each decision was valid for the version it read. The missing operation was a reservation that could succeed only while that version remained current.

An engineer reviewing this design should ask where the inventory invariant is enforced. Monitoring will expose an oversell, but it cannot make an already-issued promise exclusive. For a store that forbids overselling, reservation and conflict detection must share an atomic boundary.

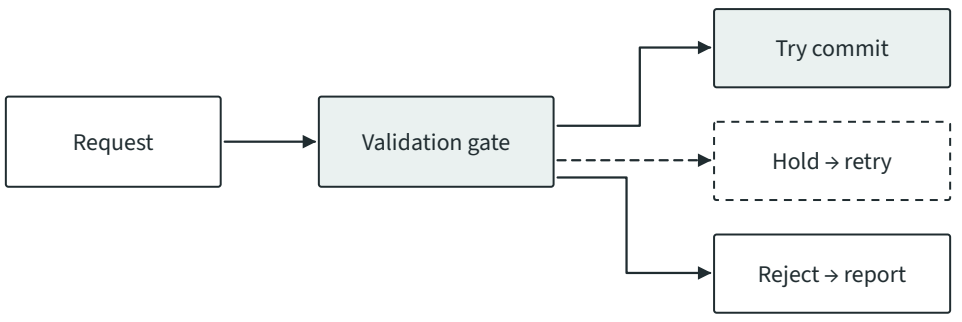
Observation: validation answers a question about a snapshot

A validator checks an input against evidence. Some checks depend only on the request: a quantity must be a positive integer, an address must have required fields, or a currency must be supported. These checks can run before persistence and produce useful errors without coordinating with other requests.

Other checks depend on shared state. Available inventory, remaining credit, and daily withdrawal limits can change while the validator runs. Reading a fresher view reduces stale decisions, but a concurrent operation can still consume the resource after that read.

Keep these outcomes distinct:

Outcome	What the caller may infer	What must happen next
Input valid	The request has the required shape and values	Check domain state
Eligible at version V	The request satisfies rules for that snapshot	Try a conditional commitment
Reservation committed	The local resource has been allocated under its invariant	Record or deliver the permitted effect
Received for review	The system durably recorded an attempt	Show pending status; do not promise allocation



Passed checks are preflight evidence; shared invariants need atomic commitment.

Mechanism: bind the check to commitment

For the last-item problem, define the invariant as `available >= 0`. A reservation must decrement inventory, record the accepted order, and create its payment intent in one transaction. The transaction also checks that inventory still has the version the caller used.

The standalone companion repository [https://github.com/monotykamary/let-it-race-examples] uses SQLite through Bun. Install Bun 1.4.2 or newer; no dependency installation is needed. To run the exact revision used in this edition:

```
git clone https://github.com/monotykamary/let-it-race-examples.git
cd let-it-race-examples
git checkout --detach 7ea672dd6ad15503649d38b294a6b6f75d4df258
bun ./gated-checkout.ts
bun test ./gated-checkout.test.ts
```

The complete example source [https://github.com/monotykamary/let-it-race-examples/blob/7ea672dd6ad15503649d38b294a6b6f75d4df258/gated-checkout.ts] and regression tests [https://github.com/monotykamary/let-it-race-examples/blob/7ea672dd6ad15503649d38b294a6b6f75d4df258/gated-checkout.test.ts] accompany the book under the MIT License. That license applies to the companion repository, not to the book or its artwork. Chapter 7 continues the same checkout through effect delivery and recovery.

The preflight check that fails

```
// Illustrative only: inventoryAt and appendReservation are domain adapters.
async function unsafeReserve(order: Order) {
  const inventory = await inventoryAt(order.expectedVersion);
  if (inventory.available >= order.quantity) {
    await appendReservation(order);
  }
}
```

Both buyers can pass this check before either append becomes visible. Replaying raw events changes the cost of `inventoryAt`; it does not close the gap between checking and appending.

The conditional write

The runnable example executes this statement inside a transaction:

```
UPDATE stock
SET available = available - :quantity,
    version = version + 1
WHERE id = 1
  AND version = :expected_version
  AND available >= :quantity;
```

One updated row means the reservation can proceed. Zero updated rows means another operation changed the version or the requested quantity is unavailable. The caller must reread and reconsider its request. It must not turn that conflict into an unconditional append.

Within the same transaction, `reserve` inserts the order and its outbox entry. If either insert fails, the stock update rolls back. SQLite's immediate transaction serializes writers in this teaching implementation. PostgreSQL can implement a similar conditional update with row-level concurrency; an event store can use an atomic expected-version append for a single stream. A multi-resource invariant needs a transaction across the affected resources, a correctly designed escrow scheme, or another explicit allocation protocol.

This example deliberately uses one SKU and a fixed price of 100 USD cents per unit. A real checkout must also cover price validity, authorization, taxes, and any cross-item constraints. A successful reservation here represents the final local decision to request payment. There is no cancellation or refund API; adding one requires a state machine that arbitrates cancellation against commitment.

Retry identity belongs inside the boundary

The caller assigns a stable order ID before retrying. Inside the transaction, `reserve` first checks whether that ID already names an accepted order. A retry with the same quantity returns that order without decrementing stock again. Reusing the ID for a different quantity fails.

Expected version and request identity solve different problems. The version detects competing changes to shared inventory. The ID identifies one logical purchase across transport retries. Store the business inputs associated with an accepted ID so a retry cannot silently change the purchase.

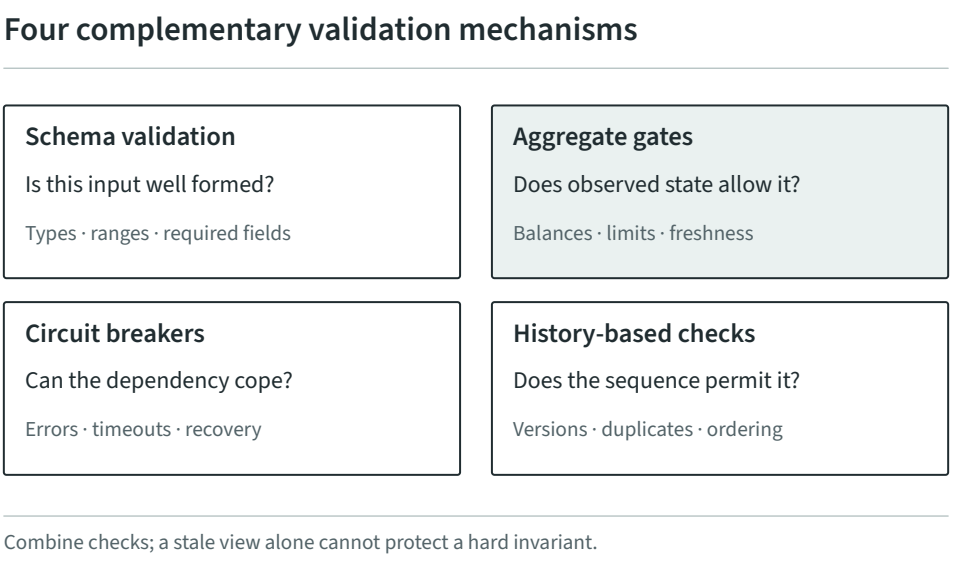
An accepted order keeps its identity even when a later retry supplies an obsolete inventory version. Its allocation already happened. A rejected attempt has no allocation; the client can reread before making another conditional attempt.

Choosing the evidence for a check

A derived view is useful for cheap preflight checks and user feedback. It can reject an obviously unavailable item before expensive fraud screening. Expose the view's source coverage and age so the caller knows which evidence it used.

A history read is useful when the rule depends on sequence or when a projection needs verification. Replay can establish the state at version V if the history is complete through V and the reducer is correct. Hard invariants still require an atomic commitment against that version.

Rule	Useful evidence	Enforcement boundary
Quantity is a positive integer	Request	Input validation and storage constraints
Stock must remain nonnegative	Authoritative allocation state	Conditional decrement in a transaction
A logical purchase counts once	Stable request ID and stored inputs	Unique identity inside the reservation transaction
A displayed count may lag	Projection plus source coverage	Freshness budget, reconciliation, and visible pending state
A withdrawal must respect a daily limit	Accepted and reserved debits for the limit period	Atomic limit reservation covering all relevant writers



Validation may still permit provisional work when the business accepts compensation. For example, recording an order attempt before stock allocation preserves demand evidence. The response must say that allocation is pending. Compensation has a cost and can fail; define the recovery owner and deadline before accepting that exposure.

Circuit breakers protect a different boundary

During the May 6, 2010 Flash Crash, the CME's Stop Logic functionality paused E-mini S&P 500 futures trading for five seconds at 2:45:28 pm. The SEC and CFTC report describes how buying interest increased during the pause and prices subsequently recovered.[1]

That pause illustrates feedback containment. A circuit breaker can stop sending work into a failing dependency or interrupt a destabilizing feedback loop. It does not prove that two inventory reservations cannot spend the same unit. Use it alongside the mechanism that enforces the business invariant.

Validation costs follow the evidence you read

Mechanism	Reads / stores	Typical cost driver	Does not establish
Schema	Input / schema	Payload size	Current business state
Aggregate gate	Derived state / view	Freshness + lookup	Safety against concurrent writers
Circuit breaker	Health window / counters	Sampling + recovery	Business-rule validity
History check	Events / history	Replay scope + indexes	Race-free commit without a fence

Qualitative comparison · costs depend on state size, indexes and scope.

Figure 6.2: Choose each check by the failure it detects. History reads and application validators still need a commitment mechanism when concurrent requests share a hard invariant.

Implication: measure the boundary you retain

Coordination has a cost, but its scope matters. A single hot inventory row can limit throughput while unrelated products proceed independently. Partitioning by a genuinely independent invariant can remove unnecessary contention without relaxing the invariant itself.

Latency alone does not determine throughput. A serial caller taking 200 milliseconds per request completes about five requests per second. A service with many independent requests in flight can sustain a different rate. Measure concurrency, hot-key distribution, conflict rate, and tail latency under the workload you intend to serve.

The runnable tests make competing calls through two database connections using the same inventory version. Exactly one reserves the last unit. Another test injects an outbox failure and verifies that the reservation and order both roll back. These tests demonstrate the local protocol; they do not benchmark a distributed database or certify storage durability under every failure.

In production, count version conflicts separately from invalid requests. Track the age of pending attempts, failed reservations, and accepted orders without completed effects. Alert when reconciliation finds a negative allocation or an accepted order missing its payment intent.

THOUGHT EXPERIMENT: a gate with a five-millisecond budget

The setup: Your order API must respond within five milliseconds. Inventory allocation is usually fast, but a required fraud service sometimes takes a second. Overselling is prohibited, and an unreviewed order cannot be charged.

The constraint to remove: You cannot make the fraud service faster.

Your task: Define what the API can durably acknowledge within the deadline. Choose when inventory is reserved, when that reservation expires, and how fraud approval races with expiry. State which transaction decides the winner.

What you might discover: A fast receipt can acknowledge an attempt while allocation or approval remains pending. Expiry introduces another competing writer that must obey the same invariant.

Closure: locate the decision that can lose a race

Take one validator in your system and write down the evidence version it reads. Then identify every operation that can change that evidence before commitment.

- Which conditional write or allocation protocol prevents two valid snapshots from authorizing incompatible commitments?
- If the response is provisional, can the user distinguish it from acceptance?
- Can a retry repeat a reservation or reuse an ID with different inputs?
- What happens when the database commits but the caller never receives its response?

Chapter 7 follows the accepted reservation across a second boundary: an external service that can succeed even when our worker dies before recording the result.

---

1. U.S. Commodity Futures Trading Commission and U.S. Securities and Exchange Commission, “Findings Regarding the Market Events of May 6, 2010,” September 30, 2010. https://www.sec.gov/files/marketevents-report.pdf [https://www.sec.gov/files/marketevents-report.pdf] ↩

CHAPTER 7

Aggregate-gated actions

The charge succeeded; the worker disappeared

> ***Hypothetical scenario:*** *This chapter continues the teaching checkout from Chapter 6. The payment provider is a durable local simulation, not an account of Amazon or a real payment gateway.*

The store reserves its last item for Alice and records a payment intent. A worker sends the charge request. The provider accepts it, but the worker stops before recording the receipt. On restart, the store still sees a pending payment.

Retrying with a new payment identity can charge Alice twice. Marking the payment complete before sending it can lose the charge if the worker stops before transmission. Waiting thirty minutes leaves both failure windows intact.

The store needs a durable intent and a provider that recognizes retries of that intent. Readiness determines whether to issue an effect; idempotency determines how to retry it.

Observation: quiet, complete, and committed are different states

A quiet view has not changed recently. A complete view includes the source data required for a particular decision. A committed order has crossed a domain boundary that allocates resources or authorizes an effect.

A partition can leave a view quiet while unseen events accumulate elsewhere. A frequently updated view can still be complete through a known offset. An accepted inventory reservation can authorize the next step immediately, even while unrelated orders continue arriving.

Define the gate for the action you want to take:

Action	Required evidence	Boundary that permits the effect
Request payment in the teaching checkout	Accepted reservation and immutable payment amount	Transaction that records the order and outbox intent
Release a real shipment	Committed allocation, required payment state, approved shipment details	Conditional shipment commitment that resolves competing edits
Publish a dashboard position	Complete ingestion prefix through offset W	Atomic publication of value and checkpoint
Send an editable draft digest	Selected revision and publishing authorization	Frozen revision plus one logical delivery identity

A quiet period can debounce draft changes or reduce repeated notifications. It is an optional scheduling policy. Safety comes from the evidence and commitment protocol required by the domain.

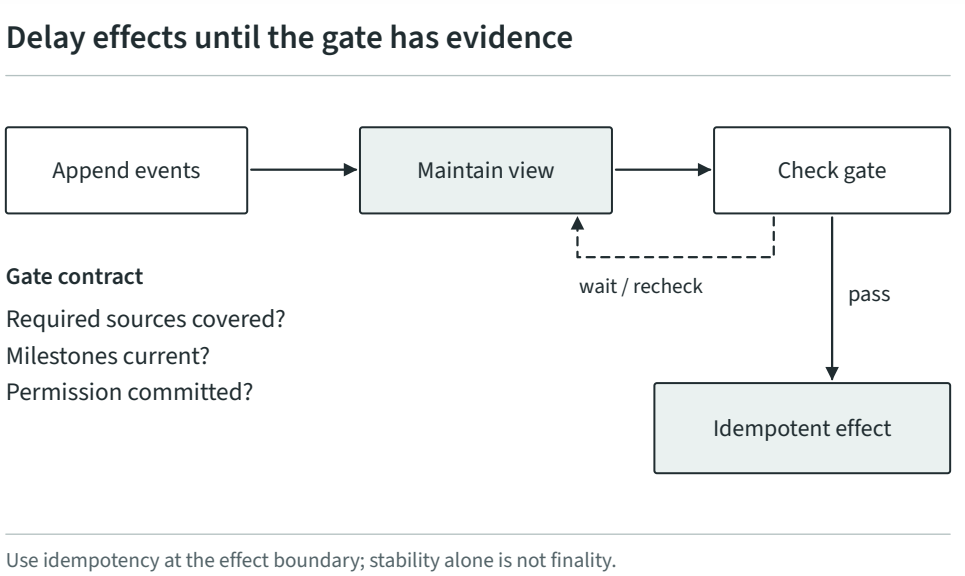


Figure 7.2: Events feed a maintained view. The gate requires source coverage and domain readiness, then commits an effect intent under the relevant invariant. The destination must recognize retries; a quiet window alone does not establish finality.

Mechanism: continue the Chapter 6 transaction

Run the complete example and its tests from the repository root:

From the companion repository checked out in Chapter 6:

```
bun ./gated-checkout.ts
bun test ./gated-checkout.test.ts
```

The source [https://github.com/monotykamary/let-it-race-examples/blob/7ea672dd6ad15503649d38b294a6b6f75d4df258/gated-checkout.ts] uses separate SQLite files for the shop and simulated provider. They do not share a transaction. The shop transaction from Chapter 6 changes stock, inserts the accepted order, and inserts an outbox row containing the order ID and charge amount.

Once that transaction commits, the worker can attempt delivery. If the process stops before delivery, the pending outbox row remains. If the database rejects the transaction, neither the reservation nor its payment intent exists.

Why a local completed set fails

```
// Illustrative only: completed and charge are application adapters.
if (!completed.has(orderId)) {
  await charge(orderId);
  completed.add(orderId);
}
```

Two workers can pass the check before either records completion. A process restart also discards an in-memory set. Moving the set into a database helps recovery, but it cannot atomically combine a local completion record with a remote charge.

The destination must enforce the retry identity at its own effect boundary, or expose another protocol that lets the sender resolve an ambiguous outcome safely. If the provider offers neither, the sender must choose between risking duplicate effects, risking missed effects, or deferring to reconciliation. A local lock cannot remove the remote failure window.

Retry the same logical effect

The runnable example's worker follows this sequence:

```
// Illustrative only: store and provider are initialized as in gated-checkout.ts.
for (const action of store.pending()) {
  const receipt = await provider.charge(action.id, action.amount_cents);
  store.acknowledge(action.id, receipt);
}
```

The simulated provider stores the charge and its receipt under a unique ID in one transaction. Repeating the same ID and amount returns the original receipt. Reusing the ID for another amount fails. Multiple workers may send the same pending action; they receive the same provider result.

The shop records completion only after receiving that result. A crash before local acknowledgement leaves work to retry. The provider then returns its existing receipt, and the shop can finish recording the result.

For a real gateway, verify the idempotency contract: key scope, retention period, behavior while the first request is in progress, parameter comparison, and lookup after a timeout. Keep retries within that contract. An expired key may require reconciliation with the provider before another send. HTTP retries alone do not establish one business effect.

A permanent decline needs its own durable state and recovery path. This teaching example leaves failed calls pending; a production worker needs backoff, attempt records, a retry deadline, and escalation. A refund is a separate logical effect with its own identity.

Source coverage and competing changes

The checkout's authoritative reservation transaction supplies the evidence needed for its simple payment intent. It does not wait for an asynchronously refreshed inventory dashboard.

If a gate depends on a derived view, record the source boundary used to compute it. For a partitioned stream this may be a vector of offsets. A single recent timestamp cannot prove that every required partition has caught up. Domain milestones must be current, with rules for revocation or expiry.

Rechecking external state immediately before an action narrows a race window but does not remove it. If a balance, authorization, or allocation must remain valid during the action, the responsible service must reserve it, conditionally commit it, or enforce it as part of the effect. Specify how cancellation competes with commitment; a reread followed by an unconditional send is insufficient.

Periodic positions while trading continues

A position dashboard can publish a new snapshot each minute while trades continue arriving. A timer that waits for one minute of inactivity does something else: an active trader can prevent publication indefinitely.

The example stores trades with stable IDs and database-assigned ingestion sequences. `refreshPositions` runs in a transaction that:

1. Reads the prior checkpoint and captures the current maximum ingestion sequence W.
2. Adds only trades after the checkpoint through W to cumulative positions.

3. Commits the new positions and checkpoint W together.

The source history stays intact. A late-arriving trade receives a later ingestion sequence and enters a later refresh. This is a processing-prefix contract, not proof that all trades with an earlier event timestamp have arrived. Multi-source systems need an explicit coverage policy for each source.

```
// Illustrative only: store is a CheckoutStore and report publishes a snapshot.
setInterval(() => {
  const snapshot = store.refreshPositions(Date.now());
  report(snapshot);
}, 60_000);
```

The timer is a scheduling target, not a hard deadline. Event-loop stalls, lock contention, and failed jobs can delay it. The example exposes unprocessed trade count and refresh age; a production service must alert or mark the view stale when its freshness budget is exceeded. If publishing to another service requires reliable delivery, record a publication intent with the checkpoint and deliver it through an outbox too.

The regression test simulates a trade every second for an hour and invokes the scheduled refresh every minute. It produces sixty snapshots with cumulative positions, without waiting for idle time or deleting source events. Another test reopens the database and verifies that refreshes neither double-count earlier trades nor omit a later arrival.

A delayed position view must not authorize exposure beyond a hard trading limit. Reserve risk capacity on the acceptance path and keep the reporting view separate. Matching needs deterministic ordering within a book; settlement has its own finality rules, covered in Chapters 11 and 21.

Evidence from the maritime workflow

> ***Author's account:*** *The maritime ticketing workflow described here is attributed to the author's experience with a major Asian port authority between 2019 and 2022. Identifying details and volumes were changed in the original account. It is not an independently audited reliability study.*

Mobile devices recorded passenger scans at dock entry and again at boarding. The design used optimistic feedback at the first checkpoint and reconciliation at the second, where staff needed the passenger's latest known boarding status. Local observations and server-side status helped the interface remain useful over unreliable cellular links.

The two checkpoints created an opportunity to reconcile before a later decision. That opportunity depended on connectivity, durable local storage, and how much time elapsed between scans. During a partition, two devices could not infer exclusivity from the absence of a competing scan. A boarding rule that prohibits duplicate admission needs an allocation protocol or a staffed exception path for unresolved cases.

The account motivates preserving scan attempts and exposing pending synchronization. It does not establish a universal no-loss or no-duplicate guarantee for offline admission, and it provides no evidence for unrelated payment-volume or revenue claims.

Failure modes and bounded waiting

Failure	Evidence to inspect	Safe response
Quiet view with stalled ingestion	Per-source offsets, consumer health, missing milestones	Defer effects that require unseen data
Provider succeeded; local receipt missing	Stable effect ID, provider receipt lookup, pending outbox	Retry or reconcile using the same identity
Readiness expired during processing	Reservation or authorization version and expiry	Let the authoritative commitment reject it
Retry deadline exceeded	Attempt history and unresolved provider outcome	Escalate without inventing a new charge identity
Projection fell behind	Source coverage and refresh age	Publish an explicit stale status; protect hard limits elsewhere

Change notifications can wake workers promptly. Durable polling can also be correct when it scans committed pending work and respects the same protocol. A lost notification should delay delivery until the next scan, rather than lose the action.

Every gate needs a maximum waiting policy. At the deadline, choose a defined outcome: cancel an uncommitted attempt, keep it pending with an alert, or route it to review. Time passing does not authorize bypassing a funds, fraud, or safety requirement. Any permitted degraded mode must have its own explicit limits and owner.

THOUGHT EXPERIMENT: the provider forgets the key

The setup: Your provider retains idempotency keys for twenty-four hours. An outage prevents your worker from receiving a charge result, and the payment remains unresolved for two days.

The constraint to remove: You cannot extend the provider's retention period.

Your task: Define what evidence lets you retry, reconcile, or escalate. Decide what the customer sees while the result remains unknown and how you prevent a second worker from inventing a new payment identity.

What you might discover: Your recovery deadline must fit the destination's protocol. A local record of intent cannot establish whether a remote effect occurred.

Closure: trace the ambiguous result

Pick one irreversible effect and follow it from the accepted business decision to the external receipt. Identify the durable record on each side of the network.

- What happens if the process stops between any two steps?
- Which service enforces the invariant that authorizes the effect?
- How does the destination recognize a retry after restart?
- What evidence ends an unresolved attempt without causing a duplicate?

Run that failure sequence in a test before relying on the gate in production.