PART I · THE ONE OBJECT AND THE ONE RELATION

1 · The Diagram That Cannot Answer

A verdict was committed, but the broker was down. Did the system lose the truth, delay the truth, expose a visible failure, or create a different truth downstream?

A software architect can draw the flow in minutes. A command enters RootCause. The aggregate checks its rules. A transaction commits the verdict. An integration event is placed in an outbox. A publisher sends it through a broker. Compliance opens a review. Audit records the sequence. Reporting updates a projection. The diagram looks responsible because every box has a clear name and every arrow points in the expected direction.

But the diagram does not settle the architecture. It does not say whether the verdict and outbox record are atomic. It does not say what happens if publication is retried after an ambiguous acknowledgement. It does not say whether Compliance may process the event twice, whether Reporting may display stale truth, or whether a translated event may accidentally grant authority that the source model never possessed.

THE PRESSURE

Once truth crosses a transaction, process, database, or service boundary, architecture is no longer only a picture of components. It is a set of claims about every allowed behavior through time and failure.

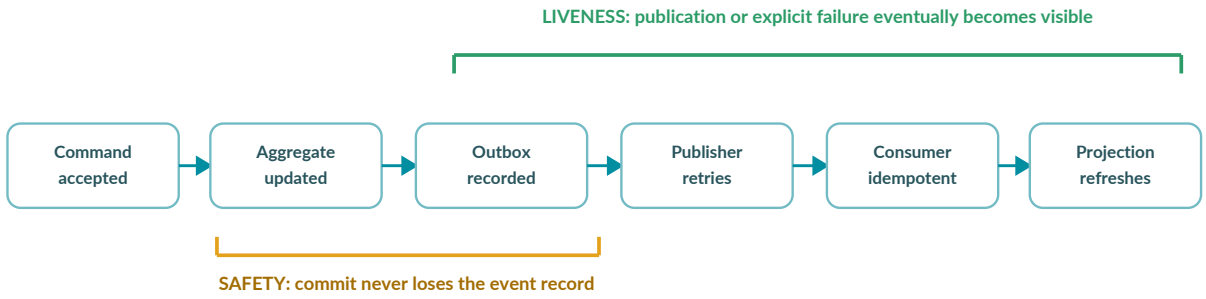

Figure 1.1 - The same flow becomes proof-oriented only after obligations are attached. The lower property forbids silent loss after commit. The upper property requires eventual publication or a visible failure.

What you already know

You already know the ingredients from ordinary backend work. Saving to a database and sending to a broker are not one magical action. A process can crash between them. A publish acknowledgement can be lost. A retry can deliver twice. A read model can lag. A consumer can misunderstand a contract. Formal reasoning does not create these problems; it gives the team a language for separating them.

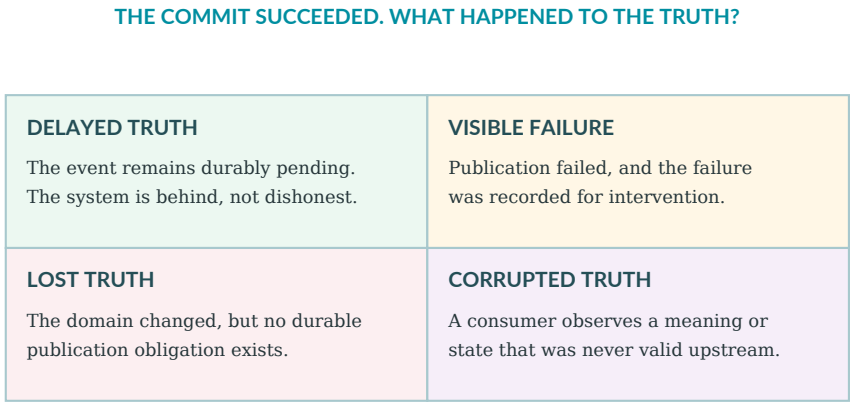

Figure 1.2 - Four outcomes that are often blurred together. A robust design distinguishes delay, visible failure, silent loss, and semantic corruption.

The everyday example: approval and publication

Consider a document-review process outside NEXUS-1. A manager approves a document. The database records the approval, and a separate publishing process makes the approved version visible to readers. The publisher is temporarily unavailable.

Question	Plain answer	Property kind
What must never happen?	The system must not show a version that was never approved.	Safety / invariant
What must eventually happen?	An approved version must become published, or an explicit failure must remain visible.	Liveness
What may be temporary?	Readers may see the previous version for a bounded interval.	Allowed staleness
What is silent loss?	Approval committed, but no durable publication obligation exists.	Safety violation
What is corrupted meaning?	A draft is exposed as approved because translation widened its status.	Semantic violation

No formal notation is needed to care about those distinctions. That is the point. The mathematics will later sharpen a real engineering question rather than manufacture one.

The same example as a small state machine

We now introduce the first formal object, but only as much as this chapter needs. A labelled transition system is written as:

$$M = \langle S, s_0, \rightarrow, L \rangle$$

The symbol S names the set of states. The symbol s_0 names the initial state. The arrow $\rightarrow$ names the allowed transition relation. The labelling function L records which atomic facts are true in each state. A run is a sequence of states connected by allowed transitions.

State	Meaning	Allowed next states
Draft	Author may still change the document.	Submitted
Submitted	The document waits for review.	Approved, Rejected
Approved	Approval is durable; publication is pending.	Published, PublicationFailed
Published	Readers can observe the approved version.	-
PublicationFailed	Failure is visible and retry policy may act.	Approved, Published

At this stage the machine is deliberately small. It does not model users, authorization, storage engines, retries, or network partitions. It models only enough to expose the property. This discipline - simplifying without hiding the load-bearing behavior - will govern every formal

model in the book.

The NEXUS-1 decision

The running NEXUS-1 case begins when a RootCauseCase closes with evidence and a verdict. The RootCause service owns that decision. Compliance must eventually receive a public event suitable for opening a review. Audit must retain evidence of the change. Reporting may lag, but it must remain derived truth.

Plain architectural statement	Formal direction
A root-cause case cannot close without evidence.	Local invariant and command precondition.
Closing the case and recording the publication obligation are one atomic decision.	Safety property over the transaction boundary.
A pending publication eventually becomes published or visibly failed, assuming the publisher continues to receive execution opportunities.	Liveness property with an explicit fairness assumption.
Compliance processes one business effect for one event identity, even if delivery repeats.	Idempotency invariant.
Reporting may be stale but cannot write RootCause-owned verdict truth.	Bounded-staleness and ownership properties.
The public event conveys a verdict, not permission to actuate.	Meaning-preservation and authority boundary.

From plain language to two first properties

A property is always stated first in ordinary language and then in symbols. The formula is not a replacement for the sentence. It is a compact commitment whose terms have been defined.

SAFETY PROPERTY

Plain: If the verdict transaction commits, a durable outbox record for that verdict already exists.
Formal: $\Box(\text{VerdictCommitted} \Rightarrow \text{OutboxRecorded})$

LIVENESS PROPERTY

Plain: A pending outbox record eventually becomes published or explicitly failed, provided the publisher is not permanently denied execution.
Formal: $\text{OutboxPending} \rightsquigarrow (\text{Published} \vee \text{ExplicitlyFailed})$

The square $\Box$ is read “always.” The relation $\rightsquigarrow$ is read “leads to” or “eventually results in.” These symbols are introduced here only to show the direction of the book. Their exact semantics, including fairness, come later. A safety property forbids a bad state. A liveness property requires progress. The outbox needs both: durability without eventual processing leaves permanent pending truth; retries without atomic durability can repeatedly publish nothing.

A first machine for the verdict flow

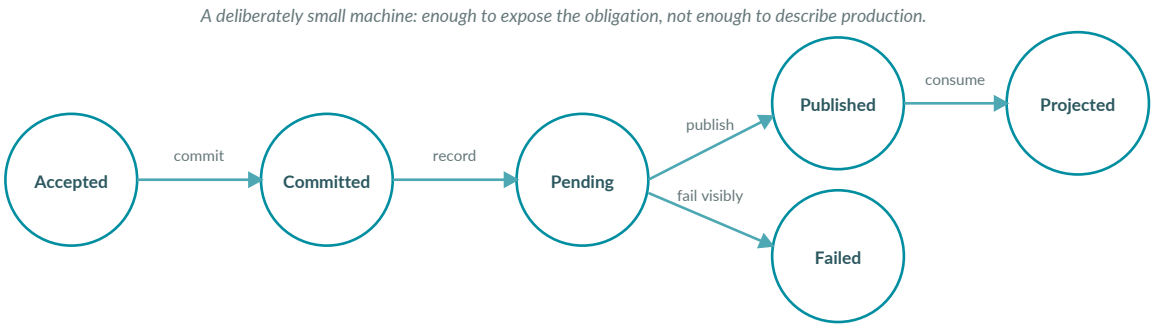

Figure 1.3 - A small state machine for the public life of a verdict. The machine excludes infrastructure detail, but includes the distinction between pending, published, and visibly failed.

The state called Committed is not allowed to exist without the fact OutboxRecorded. In a later TLA+ specification, those facts may be modelled as variables changed by one atomic action. In this introductory chapter, the important point is architectural: the outbox is not merely a table that appears in a diagram. It is the mechanism chosen to make a safety property true.

The broken example: remove the atomic outbox write

The signature move of this book is deliberate falsification. We remove the rule that writes the outbox record in the same transaction as the verdict. The implementation now saves the aggregate first and intends to publish afterwards. That design appears simpler. It also admits the following trace:

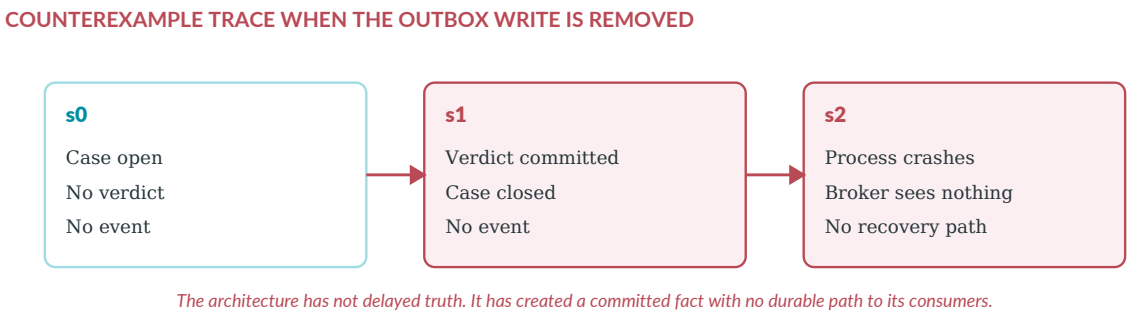

Figure 1.4 - The real architectural failure is not that the broker was unavailable. It is that the architecture allowed a committed verdict with no durable publication obligation.

This trace is a counterexample to the safety property. The bad state is reachable. The proof-oriented response is not to add more retries after the fact; there is nothing durable to retry. The architecture must be repaired at the transaction boundary.

```
// Broken shape: the process can crash between the two acts.
await rootCauseRepository.SaveAsync(rootCauseCase, cancellationToken);
await unitOfWork.CommitAsync(cancellationToken);

await broker.PublishAsync(
    RootCauseVerdictIssuedV1.From(rootCauseCase),
    cancellationToken);
```

```
// Safer shape: domain change and publication obligation commit together.
rootCauseCase.CloseWithVerdict(verdict);

await outbox.AddAsync(
    RootCauseVerdictIssuedV1.From(rootCauseCase),
    cancellationToken);

await unitOfWork.CommitAsync(cancellationToken);
```

The second fragment is still not a proof of production behavior. It is executable evidence that the code shape follows the modelled atomic decision, assuming the repository and outbox share the same transaction. The assumption must be verified in integration tests and persistence configuration.

Why one theory is not enough

The first decision already creates several different obligations. No single lens expresses all of them equally well.

Lens	Contribution to this one decision
State machine	Names the lifecycle and the forbidden committed-without-outbox state.
Hoare contract	States what must hold before and after CloseRootCauseCase succeeds.
Temporal logic / TLA+	Checks safety, eventual progress, retry, and failure traces.
Petri net	Later examines parallel Compliance, Audit, and Reporting reactions.
Alloy	Constrains which service owns and may write verdict, review, projection, and audit data.
Meaning-preserving translation	Checks that the public event does not widen verdict into actuation authority.
Refinement	Asks whether the distributed implementation preserves the observable promise of the specification.

The theories are therefore used together as one method, but not forced into one notation. The state machine is the common substrate. The proof obligations determine the lens. Refinement gathers the result at the runtime boundary.

The first Architecture Proof Case

Field	Chapter 1 entry
Decision	Close RootCauseCase and notify downstream services through an outbox.
Boundary	RootCause-owned transaction to public integration-event flow.
Property	Verdict commit never exists without a durable publication obligation.
Assumptions	Aggregate and outbox share one database transaction; records are durable after commit.
Counterexample	Commit succeeds, process crashes before any durable event exists.
Repair	Transactional outbox.
Later obligations	Fair publication, idempotent consumption, ownership, meaning preservation, projection lag.
Evidence now	Code shape, unit-of-work boundary, negative trace.
Not proved	Broker configuration, production recovery time, real C# conformance, operational certification.

What this chapter has established

Formal reasoning enters advanced architecture at the point where a design makes promises across time, failure, concurrency, ownership, or translation. A diagram remains necessary, but it is no longer sufficient. The team must name properties, construct a model that contains the load-bearing behavior, search for a counterexample, repair the architecture, and later ask whether the implementation refines the specification.

HONEST BOUNDARY

This chapter has not proved the NEXUS-1 microservice architecture. It has exposed one precise obligation and one counterexample in a deliberately small model. The symbols □ and ⇝ have been used as signposts, not yet given their full semantics. Later chapters will define them, execute specifications, and print actual checker output.

Chapter checkpoint

Question	A strong answer should include
Why is a diagram insufficient after distribution?	It shows structure but not all allowed behavior through time, failure, duplication, and translation.
What is the difference between delayed and lost truth?	Delayed truth retains a durable path to completion; lost truth has no recoverable obligation.
Why does the outbox serve a property rather than merely a pattern?	It makes commit-without-publication-obligation unreachable when used atomically.
What is a counterexample?	A reachable behavior that violates the named property.
Do the lenses prove the whole system by themselves?	No. Each proves or checks a bounded obligation in its own model; refinement and implementation evidence connect the results.
What must every proof claim name?	Model, property, assumptions, method, boundary, and residual risk.

NEXT STEP

Chapter 2 will define the common formal object beneath an aggregate, a process manager, a service, and the whole platform: M = <S, s0, ->, L>. States, transitions, runs, traces, and labels will become the grammar of the rest of the book.