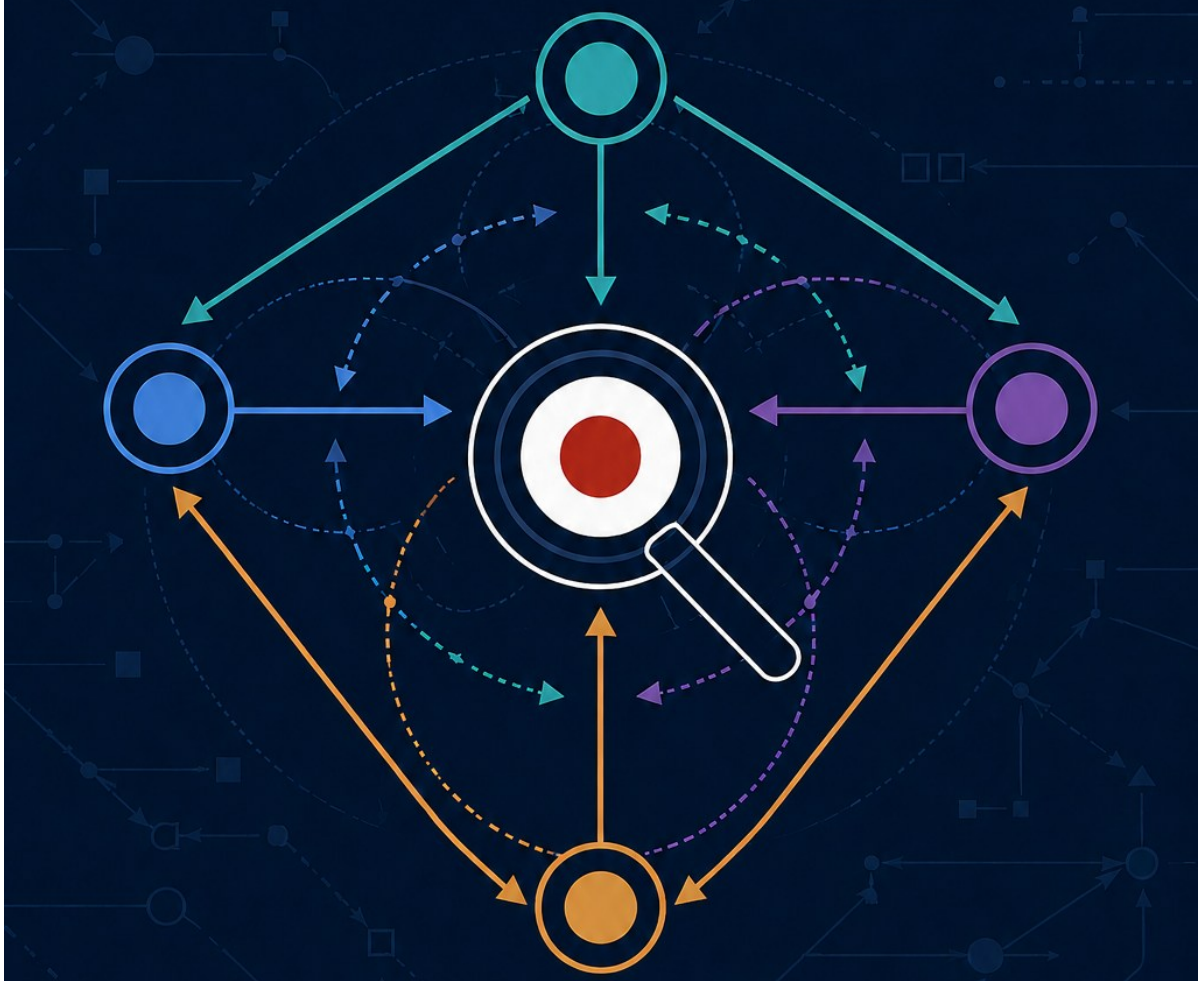
Category Theory

Software Diagnostics, Observability, and Debugging



Category Theory for Software Diagnostics, Observability, and Debugging

Applied Category Theory for Runtime Evidence, Invariants, and Root-Cause Analysis

Concept and editorial direction: Dmitry Vostokov, DumpAnalysis.org

Drafting and revision assistance: GPT-6 Astra Max

Independent editorial review: Fable 5.1 Extra

Manuscript Version 22 · September 2026

STRUCTURE · EVIDENCE · INVARIANCE

Copyright and manuscript status

Copyright © 2026 Dmitry Vostokov. All rights reserved.

Copyright © 2026 OpenTask. All rights reserved.

ISBN-13: 978-1-919135-01-4

Manuscript Version 22, September 2026.

In this book, we develop an applied category theory framework for software diagnostics, observability, and debugging. We synthesize and substantially extend earlier category-theory articles and trace-analysis patterns in *Theoretical Software Diagnostics*, Fourth Edition; *Trace, Log, Text, Narrative, Data*, Fifth Edition; *Accelerated Windows API for Software Diagnostics*, Second Edition; and *Memory Dump Analysis Anthology*, Volume 17 (Vostokov 2024b, 2026, 2024a, 2025). We cite those books as sources rather than silently mining them. We label new constructions introduced here as proposals and separate them from standard mathematical results.

Product and company names may be trademarks of their respective owners. The examples are illustrative, but they do not describe any confidential system or incident.

No part of this work may be reproduced, stored, or transmitted in any form or by any means without prior written permission from the copyright holder, except for brief quotations used in review, scholarship, or teaching as permitted by law.

Preface

Software diagnostics usually involves several execution artifacts. When we investigate a request, a trace reports one path, a metric an aggregate, a log selected statements, a memory dump one frozen state, and a profiler a sampled distribution. We then move among these artifacts, code, architecture, deployment history, and user reports. The difficult part is not simply to collect more data: we must decide which transformations preserve meaning, which views we can compare, and what kind of inconsistency counts as diagnostic evidence.

Category theory is a mathematics of objects, transformations, composition, and invariance, and these words already belong to diagnostic work. We transform artifacts, compose investigative operations, map runtime events into telemetry, compare views, and expect some routes from evidence to conclusion to agree. Therefore, we use category theory as more than an exotic vocabulary: it can specify what must be preserved and turn structural disagreement into a testable signal.

We can state the central idea of this book simply:

> ***We call category theory an anomaly-detector generator.*** *It does not detect anomalies by itself. It helps us state the identities, compositions, naturality conditions, gluing conditions, and universal properties that a correct observation process should preserve. An anomaly becomes a failed specified test: a diagram that should commute, but does not.*

This is applied category theory in a strict, operational sense. Its objects and arrows come from executions, artifacts, schemas, observations, transformations, and claims; its composition laws are translated into evidence checks; and its abstractions remain answerable to real diagnostic decisions. A categorical model earns its place here only when it clarifies what can be composed or compared, exposes what should be invariant, and states how contrary evidence would revise the model. Appendix H makes that methodological position explicit (Spivak 2014a; Fong and Spivak 2019).

This formulation grew from several lines of earlier work. *Theoretical Software Diagnostics* introduced categories of concrete execution artifacts and problem patterns, analysis-pattern functors, General Analysis Patterns as natural transformations, software codiagnostics, diagnostic operads, debugging as a presheaf, and traces and logs as possible 2-categories (Vostokov 2024b, 271–80, 319–30, 361–62). The trace-pattern reference introduced such applied concepts as Adjoint Thread, Sheaf of Activities, Trace Field, Trace Presheaf, Trace Viewpoints, Trace Join, and Visibility Limit (Vostokov 2026, 43–52, 266–67, 314, 321, 330, 344–45, 360). The Windows API volume explored API calls and glue code, cross-layer mappings, stack transformations, higher categories, operads, properads, and monoidal parallelism (Vostokov 2024a, 251–68). Volume 17 added Observation Spaces, the Diagnostics–Observability Square, Declarative Memory, Trace Plan, Trace Polynomial, and Pattern Category Theory as proposals for further development (Vostokov 2025, 17:38–49, 69–74, 96–97). *Visual Category Theory Brick by Brick* supplied an accessible diagrammatic route into the subject (Vostokov 2021). In this book, we unify these threads around observable runtime evidence and give them a stricter operational test.

Some Volume 17 passages intentionally report AI-assisted exploratory ideas and explicitly warn that individual associations or references may be strange or hallucinated (Vostokov 2025, 17:49, 96–97). In this book, we treat

those passages as attributed proposals, not established results. We retain promising constructions only after typing their objects and arrows, distinguishing analogy from definition, and identifying laws that could be tested.

We wrote this book for developers, support engineers, SREs, observability engineers, security analysts, performance engineers, architects, and researchers. We assume no university mathematics. In the main text, we introduce every construction through a diagnostic question. Appendix A begins with school-level sets, functions, logic, and graphs, and Appendix B gives a compact category-theory reference. We carry claim status in the surrounding prose rather than in separate formal panels.

We follow this discipline throughout: name the objects, name the arrows, state the composition, identify the preserved structure, and say what observation could refute the model. If any of these is missing, the categorical language remains a metaphor, and although metaphors can inspire, they cannot validate a diagnosis.

How to read this book

In each chapter, we use four recurring moves:

1. **Diagnostic question.** What operational uncertainty are we trying to reduce?
2. **Categorical construction.** Which objects, arrows, diagrams, or universal property represent the relevant structure?
3. **Failure mode.** What would non-preservation look like in concrete evidence?
4. **Practical recipe.** How can an engineer instrument, query, or test the construction?

We use three claim statuses to keep the level of claim visible:

Standard mathematics denotes a conventional definition or theorem from category theory. **Diagnostic construction** denotes a formal model proposed for a specified kind of software evidence. **Guiding analogy** denotes a useful resemblance that has not yet been supplied with all categorical data or laws. These statuses are expressed in nearby prose rather than repeated as panel headings: Appendix B presents standard mathematics, the book's typed software models are diagnostic constructions, and passages that withhold laws remain guiding analogies.

The distinction matters. A real mathematical category cannot "break associativity"; if it did, it would not be a category. What can fail is the claim that a system, telemetry mapping, or implementation realizes an intended categorical model. We therefore speak of an invariant violation as a failure of the *candidate implementation or observation map* to satisfy a specified law, not as a failure inside category theory itself.

Contents

Preface ... 3
How to read this book ... 5
Novelty, prior work, and scope ... 7
Part I — From Runtime Evidence to Categorical Structure ... 10
Chapter 1 — Why diagnostics needs a language of relations ... 11
Chapter 2 — The system, the observation, and the claim ... 14
Chapter 3 — Building categories from software behavior ... 16
Chapter 4 — Functors as instrumentation and representation ... 21
Chapter 5 — Commuting diagrams as diagnostic invariants ... 24
Part II — Comparing and Combining Evidence ... 29
Chapter 6 — Natural transformations and cross-view consistency ... 30
Chapter 7 — Products, pullbacks, and evidence joins ... 33
Chapter 8 — Limits, colimits, and evidence fusion ... 37
Chapter 9 — Adjunctions, views, and reversibility ... 40
Chapter 10 — Software codiagnostics and functorial data transformation ... 43
Part III — Time, Behavior, Context, and Higher Structure ... 46
Chapter 11 — Debugging as a presheaf ... 47
Chapter 12 — Sheaves and local-to-global diagnosis ... 50
Chapter 13 — Coalgebra, behavior, and observational indistinguishability ... 54
Chapter 14 — Traces and logs as 2-categories ... 56
Chapter 15 — Diagnostic operads and compositional workflows ... 59
Part IV — A Categorical Theory of Observability ... 62
Chapter 16 — Signals as a family of observation functors ... 63
Chapter 17 — Causality, context propagation, and distributed traces ... 67
Chapter 18 — Visibility limits, information loss, and observability drift ... 70
Chapter 19 — Trace viewpoints, ontologies, and world models ... 73
Chapter 20 — Instrumentation design through categorical contracts ... 77
Part V — Diagrammatic Debugging in Practice ... 80
Chapter 21 — The Categorical Invariant Violation pattern ... 81
Chapter 22 — A diagrammatic debugging protocol ... 84
Chapter 23 — Case study: a distributed timeout ... 87
Chapter 24 — Case study: a process memory leak ... 89
Chapter 25 — Case study: duplicate completion under concurrency ... 91
Chapter 26 — Case study: semantic-convention drift ... 93
Chapter 27 — Case study: an agentic AI tool chain ... 94
Part VI — Engineering, Evaluation, and Research ... 96
Chapter 28 — Automating categorical diagnostic checks ... 97
Chapter 29 — A category-aware diagnostics architecture ... 100
Chapter 30 — Limits, evaluation, and a research agenda ... 102
Chapter 31 — Case study: a real HTTP log artifact ... 104
Appendix A — Prerequisites from school mathematics ... 107
Appendix B — Category theory: a diagnostic quick course ... 113
Appendix C — Diagnostic invariant cards ... 127
Appendix D — Where category theory is established in computing ... 132
Appendix E — Source concordance and conceptual provenance ... 137
Appendix F — Notation and glossary ... 141
Appendix G — The conceptual emblem ... 148
Appendix H — Applied category theory: the diagnostic program ... 150
References ... 155
Index ... 158

Novelty, prior work, and scope

Category theory has a substantial history in computing. Foundational expositions cover categories, functors, natural transformations, limits, colimits, adjunctions, monads, and related constructions (Mac Lane 1998; Riehl 2016; Leinster 2014). Computer-science introductions connect the theory to programming languages and semantics (Pierce 1991), while Goguen described a broad program for categorical methods in computing (Goguen 1991). Fiadeiro applied categorical tools systematically to formal software development, architectures, specification, and concurrency (Fiadeiro 2005). Diskin and Maibaum used categorical structures for model transformation, merging, and synchronization (Diskin and Maibaum 2012).

Applied category theory has also developed mature accounts of compositional systems, databases, knowledge representation, and behavior. Examples include functorial data migration (Spivak 2012), ologs (Spivak and Kent 2012), open systems and structured cospans (Baez and Courser 2020), dynamical systems and sheaves (Schultz, Spivak, and Vasilakopoulou 2020), operadic systems engineering (Foley et al. 2021), and coalgebraic models of state and observation (Jacobs 2017). Sheaf-based data integration supplies quantitative consistency measures for heterogeneous local observations (Robinson 2017, 2020).

Topos theory places presheaves and sheaves inside categories that also carry an internal logic. This is established mathematical and computing territory: standard references develop Grothendieck toposes, subobject classifiers, and internal logic, while the topos of trees supplies a concrete model for guarded recursion and programming-language semantics (Mac Lane and Moerdijk 1994; Borceux 1994; Birkedal et al. 2012).

Separately, software diagnosis has developed distributed tracing, causal propagation, request-flow comparison, program slicing, spectrum-based fault localization, and delta debugging (Sigelman et al. 2010; Fonseca et al. 2007; Mace, Roelke, and Fonseca 2015; Sambasivan et al. 2011; Weiser 1984; Zeller and Hildebrandt 2002; Abreu et al. 2009). OpenTelemetry and W3C Trace Context standardize signal structure and cross-process context propagation (OpenTelemetry Authors 2026b, 2026a; W3C Distributed Tracing Working Group 2021).

Diagnosis from discrepancies between a system model and observations has a substantial history. Reiter's model-based diagnosis identifies sets of components for which allowing abnormal behavior restores consistency between a system description and its observations (Reiter 1987). Our failed diagrams do not replace that diagnostic theory or determine a unique faulty component. They specify typed comparisons whose residuals may be supplied to a separate diagnosis procedure.

Testing relations between several executions or transformations is also established. Metamorphic testing uses expected relations among related inputs and outputs when individual expected results are difficult to provide (Chen et al. 2018). Property-based testing, exemplified by QuickCheck, generates inputs for executable properties (Claessen and Hughes 2000). Our contribution is the proposed organization of diagnostic comparisons through declared categories, interfaces, and compositional laws. The test-generation and cross-checking ideas themselves are not new.

The four source books form an unusual bridge between these literatures. They apply or propose categories, functors, natural transformations, presheaves, sheaves, 2-categories, coalgebra, operads, API categories, Observation Spaces, and Pattern Category Theory for software diagnostics, trace analysis, and internals (Vostokov

2024b, 2026, 2024a, 2025). A targeted literature search updated on 4 September 2026 found adjacent work in all the areas listed above, but did not locate any peer-reviewed monograph whose organizing subject is category theory for *software diagnostics, observability, and debugging across logs, traces, metrics, dumps, profiles, APIs, and runtime evidence*. A search result is not a proof of absence, so this is a bounded literature claim rather than an absolute priority claim.

The narrower topos claim is equally bounded. We found established work on toposes in logic and software semantics and on sheaves in distributed computation, but no peer-reviewed treatment combining a diagnostic scope category, an observational-coverage topology, telemetry sheaves, and contextual hypothesis classification inside an executable software-diagnostics program. We therefore present that combination as our proposed diagnostic-topos synthesis, not as new topos theory and not as an absolute priority claim.

We make the following book-level contributions:

1. A unified **diagnostic observation system**: an execution category, modality-specific evidence categories, and explicit observation or normalization functors.
2. A **cross-view naturality test** in which logs, traces, metrics, dumps, and derived views are compared only after mapping them into a declared common evidence category.
3. A **Categorical Invariant Violation** pattern: an expected diagram plus a commutativity test, scope, residual, and localization procedure.
4. A treatment of **observability drift** as failed preservation across software versions, instrumentation versions, or semantic-convention mappings.
5. A compositional account of diagnostic work through **diagnostic operads** and typed pattern pipelines.
6. A practical route from categorical diagrams to telemetry contracts, automated consistency checks, and falsifiable case analyses.
7. A **diagnostic-topos proposal** in which a Grothendieck topology records sufficient observational coverage and a subobject classifier records the scopes under which a diagnostic hypothesis is supported.
8. A typed extension of the newer source proposals: the Diagnostics–Observability Square becomes a commutativity program, Declarative Memory becomes a chain of partially faithful representations, and Pattern Category Theory becomes a testable category of pattern specifications, refinements, detections, and explanations.

In this book, we do not claim that category theory replaces statistical anomaly detection, causal inference, formal verification, program analysis, or human expertise. We supply a coordination layer: a way to state how different models and tools ought to fit together and where their agreements can be tested.

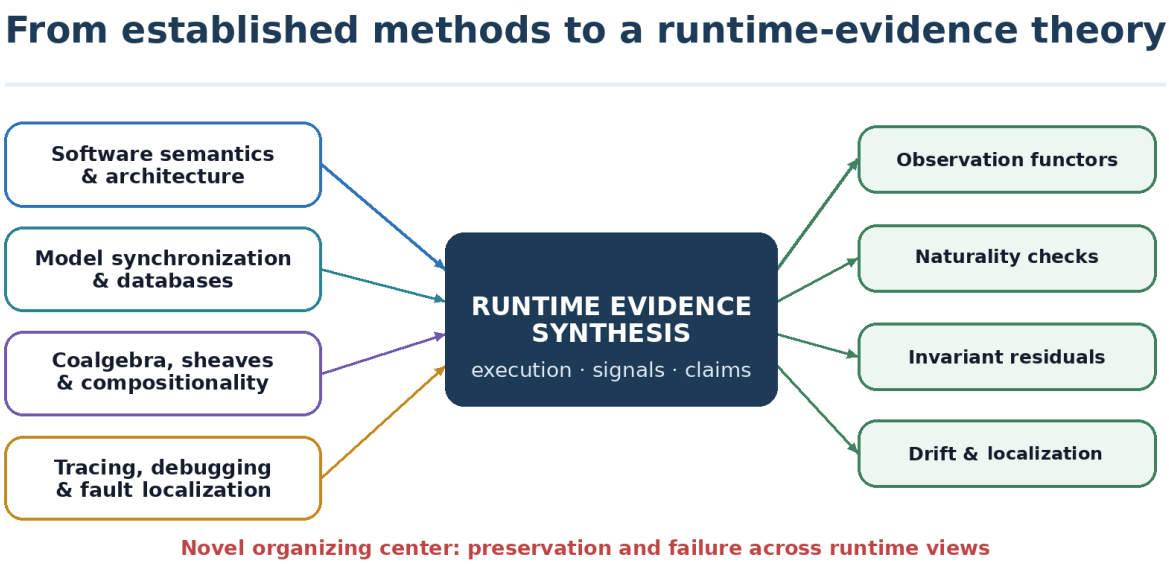


Figure 1. Prior work supplies several mature strands; this book composes them around runtime evidence and diagnostic invariants.

Part I — From Runtime Evidence to Categorical Structure

Chapter 1 — Why diagnostics needs a language of relations

> ***Diagnostic question:*** *How can evidence from different tools be combined without assuming that similar labels mean the same thing?*

Evidence is transformed, not merely collected

When we begin investigating a production problem, the first command has not yet been run, but transformations have already begun. A user report has transformed a lived failure into words, an alert has transformed measurements into a threshold crossing, a trace collector has transformed execution into spans, a symbol server has transformed addresses into names, and a stack walker has transformed memory into a call sequence. Every useful artifact is therefore the result of one or more mappings.

Diagnostics therefore has two simultaneous subjects:

- the software system and its behavior;
- the transformations through which that behavior becomes evidence.

If we model only the system, we may mistake a telemetry defect for an application defect. If we model only the evidence, we may forget what the evidence is supposed to preserve. A useful theory must connect both sides.

The earlier pattern-oriented formulation treats diagnostics as the extraction of indicators from execution artifacts through analysis patterns (Vostokov 2024b, 216–18, 264–80). The trace reference makes the transformation layer concrete: filtering produces Adjoint Threads, joining produces Trace Joins, quotienting produces Quotient Traces, and working-set selection reduces the active evidence (Vostokov 2026, 48–52, 250, 321, 367–68). Category theory adds one disciplined question to every such operation: **what structure is preserved?**

Three kinds of sameness

Engineers often say two events are “the same” when they share a timestamp, trace identifier, message template, stack frame, request key, or semantic role. These are different equivalence relations. Conflating them produces false joins and false causality.

Category theory does not choose the correct sameness for us: we must put that choice into the model. We say that two objects are isomorphic when they have mutually inverse structure-preserving arrows, equal when they are literally the same object in the chosen formalism, and observationally equivalent when a declared family of observations cannot distinguish them. These notions may coincide in one model and diverge in another.

For example, two retry spans may have the same operation name and attributes but be different attempts. A metric exporter may intentionally merge them into one count. The trace category distinguishes them, while the metric category may not. The observation map is many-to-one. That loss is not a defect by itself, but it becomes a defect if an investigation later assumes that the metric identifies the attempt responsible for a timeout.

Relations first

We begin the categorical viewpoint with relationships rather than fields. We may use a state, event, component, artifact, hypothesis, or query result as an object, and a transition, call, message send, extraction, filtering operation, schema mapping, or evidential implication as an arrow. The question determines the useful choice.

We can use the same concrete data to support several categories. As an ordered sequence, a trace gives a path category. Grouped by thread, the same rows give a category of activities, while causal context gives a causal category of the same messages. The same artifact transformed by filters gives a category of views. Asking "what is the category of this log?" is incomplete; ask "which objects and arrows make the diagnostic claim testable?"

A first operating rule

Before correlating two artifacts, we write down:

1. the identity of an object in each artifact;
2. the arrows that relate objects within each artifact;
3. the mapping between the artifacts;
4. the property the mapping is expected to preserve;
5. the observation that would reveal non-preservation.

If the only mapping is "same timestamp within five seconds," the model should say so. A weak, explicit relation is safer than a strong, implicit one.

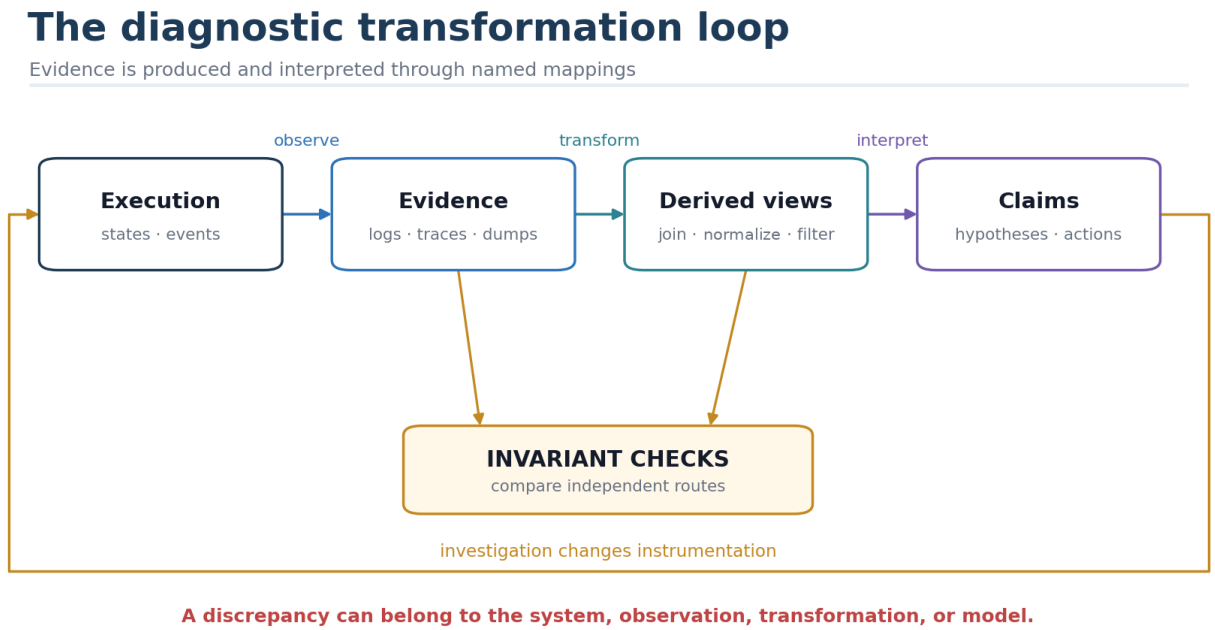


Figure 2. Diagnostics connects execution, evidence, derived views, and explanations through named transformations.

Practical checkpoint

We choose one dashboard panel used in an incident and identify its raw source, aggregation, grouping keys, missing-value policy, and update cadence. Then we identify one conclusion that people routinely draw from it. The gap between the transformation actually performed and the conclusion assumed is our first candidate categorical invariant to test.

Chapter 2 — The system, the observation, and the claim

> ***Diagnostic question:*** *Which part of a diagnostic statement belongs to the system, which part belongs to the observer, and which part belongs to interpretation?*

A three-layer model

Consider the statement "service B caused the request to exceed its latency objective." It contains at least three layers:

Execution layer: Processes, requests, queues, locks, packets, memory cells, and state transitions occurred.

Observation layer: Instrumentation selected events, attached context, sampled spans, aggregated metrics, wrote logs, and perhaps captured a dump.

Interpretation layer: An engineer or algorithm related observations to architecture and concluded that service B was causal rather than merely correlated.

Confusion between these layers is a recurrent source of diagnostic error: a missing span is an observation-layer fact. It may be caused by a missing execution, sampling, exporter loss, context propagation failure, or schema mismatch. Only the first possibility directly says that the operation did not happen.

Categories for each layer

We use three provisional categories:

- $\mathcal{E}$ — an **execution category**, whose objects and arrows represent a chosen granularity of system behavior;
- $\mathcal{A}$ — an **artifact category**, whose objects are evidence items or evidence scopes and whose arrows are derivations or relations;
- $\mathcal{H}$ — a **hypothesis category**, whose objects are diagnostic claims and whose arrows represent refinement, implication, or evidential support.

These categories are not universal. A memory-safety investigation and a latency investigation may use different execution objects. Even within one incident, a request-level category may be refined into a thread-schedule category. Granularity is part of the model.

An instrumentation functor $O: \mathcal{E} \to \mathcal{A}$ maps selected execution structure into evidence. An interpretation process $I: \mathcal{A} \to \mathcal{H}$ maps evidence into hypotheses. The composite $I \circ O$ represents one route from runtime behavior to diagnostic claim.

We call this mapping a functor only if it preserves identities and composition. If two execution transitions compose but their emitted events cannot be composed in the declared artifact model, O is not a functor on that domain. We can repair the model by restricting the domain, enriching the artifact category, or representing observation as a partial or lax construction. We should not simply keep the word functor.

Claims are not observations

An alert named database_latency_root_cause is still an observation produced by code; its name does not make its conclusion true. Similarly, a log line saying "authentication failed" may report a local return code, not the causal explanation for a user-visible failure. The hypothesis layer makes this distinction explicit.

A useful hypothesis category can be a preorder: $h_1 \to h_2$ means that h_1 is at least as specific as, and therefore entails under the declared diagnostic semantics, h_2. A concrete claim such as "connection pool P was exhausted between 10:02:14 and 10:02:31" may imply the broader claim "database access was resource-constrained." The reverse implication does not hold. Evidential support is represented by a separate typed relation and is not folded into this refinement preorder.

Diagnostic provenance as paths

Every reportable conclusion should have at least one provenance path:

$$e \xrightarrow{O} a_0 \xrightarrow{t_1} a_1 \xrightarrow{t_2} \cdots \xrightarrow{I} h.$$

Here e is the modeled execution, a_0 raw evidence, t_i transformations, and h the hypothesis. The path does not prove the hypothesis, but it exposes where information was filtered, joined, inferred, or supplied by background knowledge.

Here we extend the idea of a Software Diagnostic Space as a graph of narratives and transformations (Vostokov 2024b, 258–62). In our categorical refinement, we treat composable routes as first-class and ask when different routes to the same claim agree.

Practical recipe: the claim ledger

For each significant diagnostic claim, we record:

- the claim and its scope;
- source artifacts;
- transformations and parameters;
- identity and correlation rules;
- assumptions supplied from architecture or code;
- alternative paths to the same claim;
- disconfirming evidence.

This ledger later becomes a diagram, and it also prevents an interpretive conclusion from masquerading as raw telemetry.

Chapter 3 — Building categories from software behavior

> ***Diagnostic question:*** *How do we turn an event graph or state-transition model into an actual category without smuggling in invalid compositions?*

Categories need identities and composition

We use the standard definition: a category consists of objects, arrows between objects, an identity arrow for every object, and an associative composition operation. The definition is compact, but it places real obligations on a software model.

Suppose objects are runtime states and arrows are individual transitions. If state x moves to y by f and y moves to z by g, the category must contain a composite $g \circ f : x \to z$. The composite may represent the path rather than a single machine instruction. If we insist that arrows are only atomic transitions, composition is not closed and we do not yet have a category.

The free category on an event graph

The standard repair is the **free category** on a directed graph. Vertices become objects. Finite directed paths become arrows. A zero-length path is the identity. Composition concatenates paths. Associativity follows from associativity of concatenation.

We find this construction diagnostically useful because most traces begin as graphs or sequences, and it gives us a valid category before we impose domain-specific equations. Later we may identify paths that should be equivalent, for example, two retry implementations with the same declared external effect. Such a quotient requires evidence and a stated congruence: similar endpoints do not grant it.

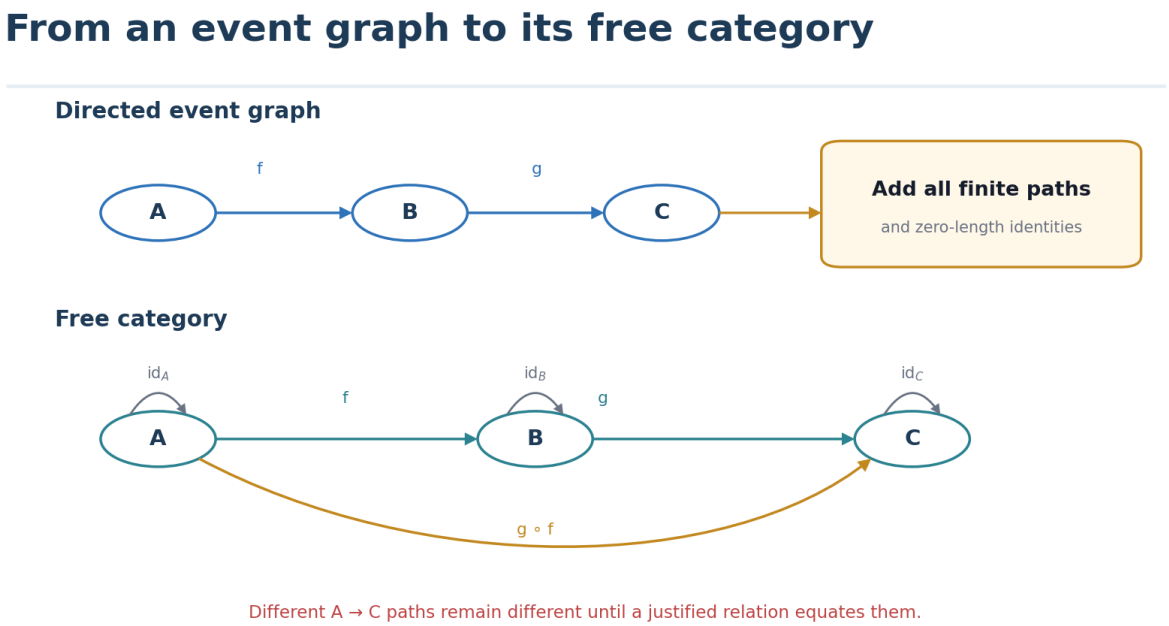


Figure 3. An event graph becomes a free category by adding identity paths and all finite path composites.

Several useful execution categories

State-transition category: Objects are states at the chosen resolution, and arrows are execution paths.

Happens-before category: Objects are events; there is at most one arrow $a \to b$ when a happens before b. This is a preorder category based on Lamport's partial order (Lamport 1978). Concurrent events may have no arrow between them.

Call category: Objects are interfaces or call sites, and arrows are call paths. This model supports questions about composition across services but may hide state.

Artifact-transformation category: Objects are artifacts and derived views, and arrows are filters, joins, symbolization, aggregation, slicing, or enrichment steps.

Pattern category: Objects are diagnostic situations or pattern states; arrows are pattern refinements or applications. The earlier categorical foundations distinguish concrete execution artifacts, concrete problem patterns, and analysis-pattern functors (Vostokov 2024b, 271–72).

We do not expect one choice to dominate: a model is useful when its arrows match the question and its required laws are testable.

API categories: two valid constructions

Accelerated Windows API for Software Diagnostics proposes an informal category in which API functions are objects and the glue code that connects them supplies arrows. It also considers layered API categories, stack traces as functorial views, transformations between API sequences, higher categories for diagnostics, and monoidal composition for independent calls (Vostokov 2024a, 251–68). The proposal becomes formal in more than one way.

In the **call-as-object model**, we first construct a directed graph. Its vertices are typed API invocation contracts, including function, version, precondition, result shape, and resource state. An edge records an admissible glue program from the post-state of one call to the pre-state of another. We then take the free category of this graph. Its arrows are finite paths of compatibility edges, its identities are empty paths, and composition is path concatenation. Such paths describe candidate API sequences. Executing them requires the intermediate API calls as well as the glue programs; the categorical composition here is not ordinary composition of glue programs alone.

In the **call-as-arrow model**, objects are resource or state interfaces and an API operation is an arrow between them:

$$\text{Uninitialized} \xrightarrow{\text{Initialize}} \text{Ready} \xrightarrow{\text{Create}} \text{HandleOpen} \xrightarrow{\text{Close}} \text{Ready}.$$

This second construction is valid when source and target states make API composition meaningful. It is invalid only if "API call" is left untyped so that arbitrary calls are assumed to compose. The two constructions answer different questions and may be related by a functor; neither is a universal category of an operating system.

Cross-platform unification then uses a semantic category $\mathcal{S}$ of operations such as allocate, wait, map, send, and close, with platform mappings

$$W: \mathcal{W} \to \mathcal{S}, \qquad L: \mathcal{L} \to \mathcal{S}, \qquad M: \mathcal{M} \to \mathcal{S}$$

for Windows, Linux, and macOS categories. The Volume 17 "OS Internals and Category Theory" note proposes this cross-OS direction (Vostokov 2025, 17:49). A translation is trustworthy only for the subcategory whose ownership, lifetime, concurrency, and error semantics it preserves. Matching function names is not enough.

Software internals as representation layers

Volume 17 introduces **Declarative Memory**: source-level values may be reordered, merged, split, eliminated, or mapped many-to-many into virtual memory, especially under optimization (Vostokov 2025, 17:43–44). This yields a diagnostic representation chain:

$$\mathcal{D}_{src} \xrightarrow{C} \mathcal{D}_{layout} \xrightarrow{V} \mathcal{D}_{virtual} \xrightarrow{P} \mathcal{D}_{physical} \xrightarrow{O} \mathcal{D}_{dump}.$$

The objects may be scoped value instances, storage regions, pages, and captured records. Arrows record lifetime, containment, aliasing, translation, or reachability at the relevant layer. Compiler layout C, address translation P, and capture O are not automatically ordinary functors on every chosen category. Optimization can eliminate a value; one source value can occupy several locations over time; several source values can reuse one location. Relations, spans, profunctors, partial maps, or time-indexed categories may be more honest.

Chapter 7 develops one of these alternatives as a categorical span. It retains the correlation witnesses for a source-value-to-location association and composes them with the witnesses connecting locations to dump artifacts.

A debugger creates an independent path through symbols and module metadata. The diagnostically useful test compares "source value → compiled layout → captured address" with "captured address → symbolized source value." A failed square localizes wrong symbols, build mismatch, stale module state, optimization ambiguity, corrupt capture, or an overconfident source-level model.

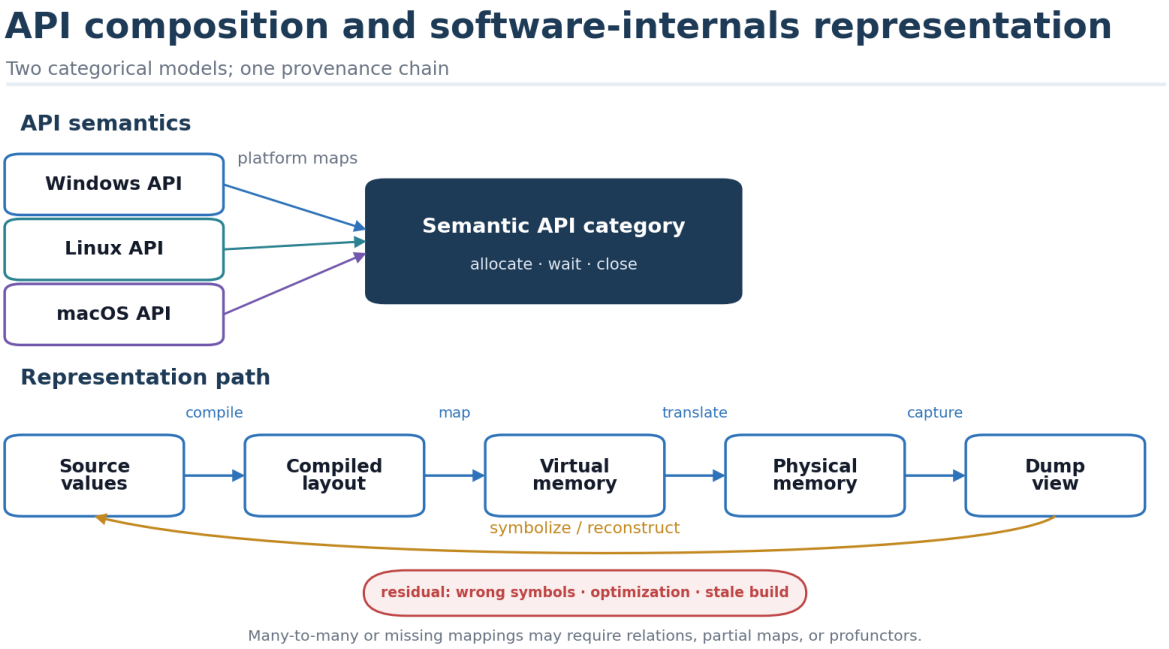


Figure 3A. API and software-internals models connect call composition, platform semantics, and source-to-memory representation without collapsing their different notions of identity.

Equations between paths

We can present a category by generators and relations: generators are the primitive arrows, and relations state which composite paths are equal. We can use the same idea for a telemetry contract.

For example, let enqueue, dequeue, direct, and finish generate alternative request paths from a common start state. Suppose a correlation adapter maps both the direct route and the queued route to the same declared request completion:

$$\text{finish} \circ \text{direct} = \text{finish} \circ \text{dequeue} \circ \text{enqueue}.$$

The equation is not necessarily true of latency, resource consumption, or internal messages. It may be true only of an external state transition such as “one order accepted.” The codomain determines what equality means.

Granularity and false composition

When two log rows appear next to each other, we cannot yet compose them because they may belong to different requests or threads. Likewise, we cannot compose two spans merely because their names match when context has been lost. In the model, we require the target of the first arrow to match the source of the second.

We get a practical test for accidental correlation: if the join rule cannot establish a shared intermediate object, the proposed composite is ill-typed. The category-theoretic language turns “these events look related” into “show the object through which the arrows compose.”

Construction checklist

1. Choose the diagnostic question and granularity.

2. Name the objects.
3. Name primitive arrows and their source and target.
4. Close under identities and composition, often by taking a free category.
5. Add only justified equations between paths.
6. State which raw fields realize each object and arrow.
7. Test whether new evidence violates typing or a declared path equation.

Chapter 4 — Functors as instrumentation and representation

Diagnostic question: *When does telemetry preserve enough execution structure to support the conclusions drawn from it?*

What a functor promises

A functor $F: \mathcal{C} \to \mathcal{D}$ maps objects to objects and arrows to arrows while preserving identities and composition:

$$F(1_x) = 1_{F(x)}, \qquad F(g \circ f) = F(g) \circ F(f).$$

In diagnostics, we may use an execution model as the source and an evidence model as the target. We do not promise that every detail survives. We promise that the structure declared relevant survives consistently.

An instrumentation library might map a request start to a span start, a child call to a child span, and a completion to a span end. If the span graph preserves the parent-child composition of the modeled calls, the mapping is functorial for that structure. Sampling can remove objects and arrows, so the full trace may instead be a functor on a selected subcategory or a partial observation. In the model, we should say which.

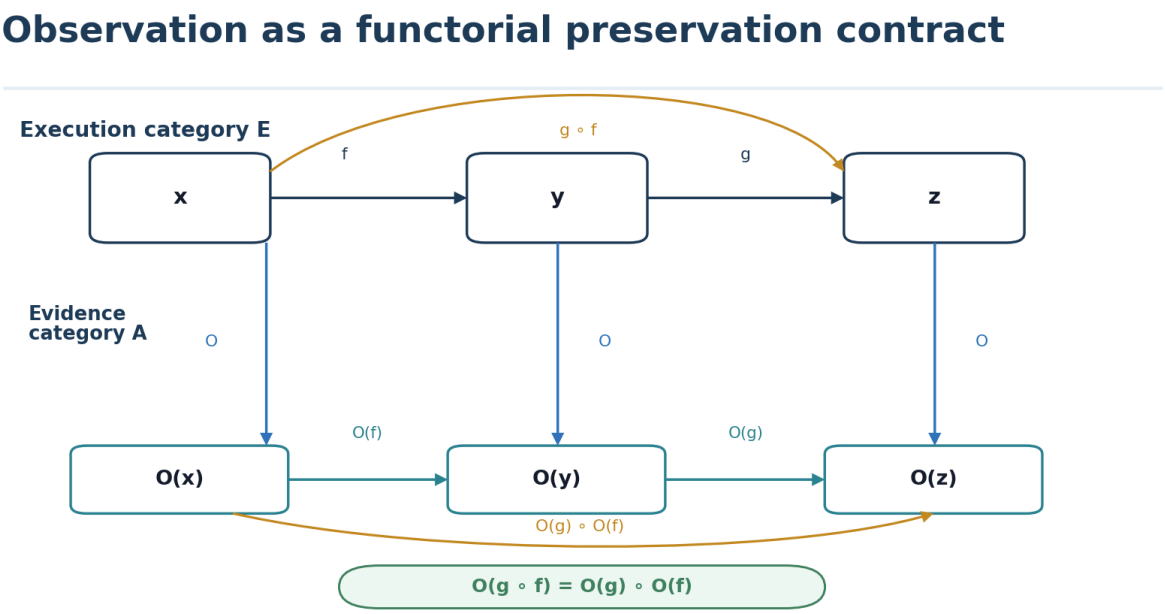


Figure 4. An observation functor preserves identities and the composition of the execution paths included in its scope.

Faithful, full, and essentially surjective—operational meanings

We call a functor **faithful** if it does not merge distinct parallel arrows. In an observability model, faithfulness concerns the execution relationships represented by those arrows. Metric aggregation can collapse distinct execution values or paths. To call this non-faithfulness, we must specify distinct parallel execution arrows that the chosen functor maps to the same evidence arrow.

We call a functor **full** if every arrow between mapped objects in the target comes from an arrow in the source. If a dashboard permits apparent transitions that cannot occur in the system, the visualization map may not be full with respect to the intended semantics or, more commonly, the target category contains analytical relations beyond execution relations and fullness is not desired.

We call a functor **essentially surjective on objects** if every target object is isomorphic to one in the image. Operationally, we ask whether every kind of evidence object represented in the target is accounted for by the source model, up to the declared notion of equivalence.

These properties are not grades on a universal scale. Aggregation should often be non-faithful, and we ask whether the loss profile is compatible with the intended use.

Observation loss profiles

For each telemetry modality, we record which distinctions it preserves:

- event identity;
- causal or parent-child order;
- wall-clock order;
- multiplicity;
- duration;
- payload or state;
- component identity;
- semantic type;
- absence.

Metrics preserve aggregates and often discard identity. Logs may preserve payload but omit explicit causal edges. Traces preserve request context but may be sampled. Dumps preserve a rich state at one instant but usually omit the complete preceding path. Profiles preserve distributions but not individual executions. The trace reference calls the boundary created by unavailable or invisible information a Visibility Limit (Vostokov 2026, 360). A categorical loss profile makes the boundary explicit.

Partial observation without pretending

Real instrumentation is partial. Three honest formalizations are common:

1. Restrict $\mathcal{E}$ to the subcategory actually observed.
2. Map into a category whose objects explicitly include “missing” or “unknown.”
3. Use a formalism for partial maps, spans, profunctors, or lax functors when the application warrants it.

The first option is easiest and often sufficient, and the danger is forgetting the restriction when interpreting the result.

Functoriality test

For every composable pair f, g in a test corpus:

```
expected = observe(compose(g, f))
actual   = compose(observe(g), observe(f))
assert equivalent(expected, actual)
```

The equivalence predicate belongs to the target model. It may compare trace structure, normalized event sequences, a multiset of semantic events, or an aggregate. A failed test may indicate broken context propagation, exporter reordering, duplicate emission, schema mismatch, or an incorrect source model.

Chapter 5 — Commuting diagrams as diagnostic invariants

Diagnostic question: *How can a structural expectation become an executable anomaly test?*

Two routes, one expected result

We say that a diagram commutes when all directed paths with the same start and end have equal composites. In the simplest square, objects A, B, C, D and arrows f, g, h, k satisfy

$$k \circ f = h \circ g.$$

For diagnostics, we use the two paths to represent two independently meaningful routes from a source to a result: runtime-to-log then normalize, or runtime-to-trace then normalize; configuration-to-deployment then inspect, or configuration-to-expected-state then compare; request-to-metric exemplar, or request-to-span then aggregate.

The square is valuable only when the routes are typed and equality in the codomain is defined. "The dashboard and logs should agree" is not yet a diagram. "For every completed request with propagated context, the normalized status derived from the root span equals the normalized status derived from the terminal application log" is testable.

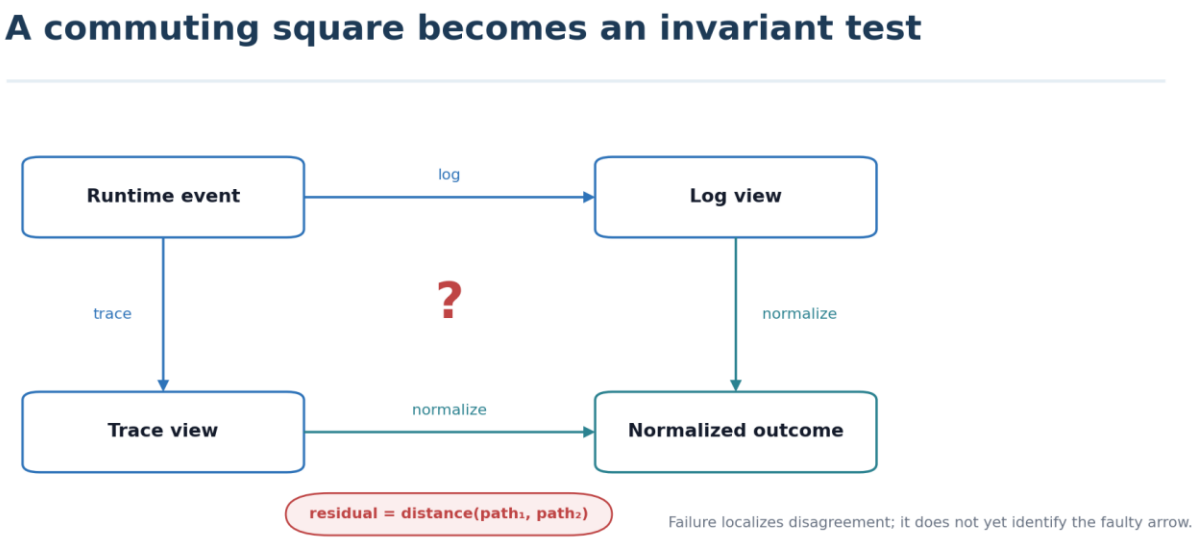


Figure 5. A categorical invariant is an explicitly expected equality of parallel composites. Noncommutativity is a localized discrepancy, not a mystical failure.

From equality to a residual

Exact equality is often too strong for timestamps, sampled data, floating-point measurements, or probabilistic models. If the target category carries a distance or preorder, define a residual between parallel results:

$$r(f, g, h, k) = d(k \circ f,\ h \circ g).$$

The distance d is additional structure; ordinary category theory does not provide it. A threshold τ yields an operational test $r(f, g, h, k) \leq \tau$. A sheaf consistency radius, an edit distance between normalized event sequences, or an interval overlap test may supply the comparison (Robinson 2020).

Categorical invariants

In this book, we use the term *categorical invariant* as an operational umbrella for a specified structural law that an implementation or observation map is expected to realize. Examples include:

- preservation of identity and composition by an observation map;
- naturality of a cross-view transformation;
- agreement on overlaps for a sheaf-like evidence model;
- satisfaction of a universal-property contract;
- closure of a typed diagnostic operation under composition;
- a monad, comonad, or algebra law in an explicitly chosen effect model.

We use the law as the detector template, and data and tolerances instantiate it.

Failure does not immediately identify cause

A noncommuting square localizes a mismatch among a small set of objects, arrows, and transformations. It does not tell us which arrow is wrong. The cause may be:

- a system defect;
- an instrumentation defect;
- loss or reordering in transport;
- a schema or semantic-convention mismatch;
- a clock or correlation problem;
- an invalid equality assumption;
- an implementation error in the checker.

We still gain something important: the problem has moved from an unbounded artifact space to the boundary of a declared diagram.

The invariant card

Every executable diagnostic diagram should have a card:

Name: A stable identifier.

Question: The uncertainty it reduces.

Objects and arrows: Typed, scoped, and tied to concrete fields or operations.

Expected law: The parallel composites or universal property.

Comparator: Exact equality, equivalence, preorder, or distance.

Tolerance and missingness: What counts as acceptable and how absence is treated.

Failure interpretation: Candidate causes and known confounders.

Localization: How to decompose a failed large diagram into smaller squares.

Disconfirmation: Evidence that would show the model, rather than the system, is wrong.

We use this card as the bridge from diagram to engineering practice.

A small example

Suppose an API gateway writes a terminal log and a tracer exports a root span. Both are normalized to a common status category $\mathcal{K} = \{\text{ok,client-error,server-error,unknown}\}$. For each request q:

$$N_L\big(L(q)\big) \stackrel{?}{=} N_T\big(T(q)\big).$$

A mismatch is a categorical invariant violation for the declared contract. It may expose a span closed before an asynchronous failure, a logging statement that records a downstream status, or an incorrect normalization table. The diagram does not prejudge which.

Worked example: pasting two diagnostic squares

A single commuting square compares two routes, and a real diagnostic pipeline usually has several stages. Pasting places squares edge to edge along a shared boundary. If each local square commutes, the route around the outside also commutes. We get two levels of evidence: the outer rectangle tests the end-to-end contract, while the local cells show where a disagreement first appears.

The typed pipeline

Let L be raw log lines, R parsed records, and N normalized records. Let F be extracted field strings, T typed fields, and D diagnostic outcomes. The top route parses and normalizes, and the bottom route extracts, types, and classifies. The vertical arrows compare the representation at each stage. Every arrow has a declared source and target, so an accidental comparison of unlike values is a type error rather than a mysterious mismatch.

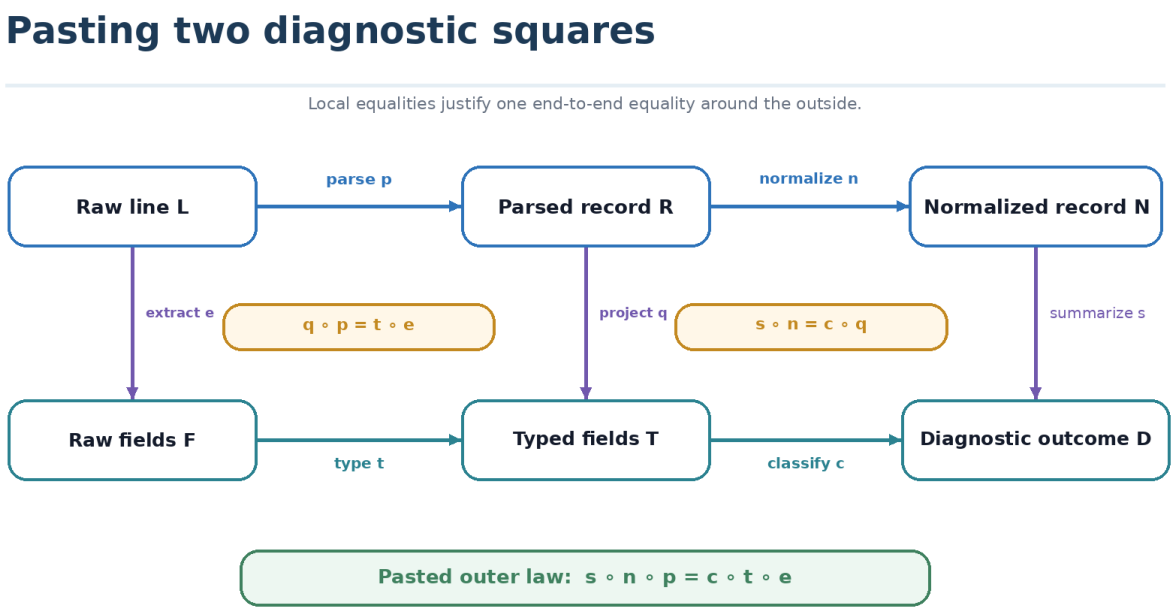


Figure 5A. Two adjacent diagnostic squares share the middle comparison q. Their local laws paste into one outer end-to-end law.

The left cell starts at the same raw line and ends at the same typed fields. Parsing followed by projection must agree with direct extraction followed by typing:

$$q \circ p = t \circ e \quad \text{(two routes } L \to T\text{)}$$

The right cell starts at the parsed record and ends at the same diagnostic outcome. Normalization followed by summarization must agree with projection followed by classification:

$$s \circ n = c \circ q \quad \text{(two routes } R \to D\text{)}$$

Why the outer rectangle commutes

The proof is only substitution plus the rule that parentheses may be moved in a composition, and we start on the upper route. The right-cell law replaces s ∘ n by c ∘ q, and the left-cell law then replaces q ∘ p by t ∘ e:

$$s \circ n \circ p = c \circ q \circ p = c \circ t \circ e$$

Therefore the outer routes L → D agree: this is the pasting principle in its smallest useful diagnostic form. We neither check the shared edge q twice nor guess it away: it is the interface that lets the two local contracts compose.

One concrete record

H4 - - [01/Jul/1995:00:00:11 -0400] "GET /shuttle/countdown/liftoff.html HTTP/1.0" 304 0

For this line, both routes produce the outcome (3xx, zero-payload), and a second real line in Chapter 31 has status 200 and zero bytes, so it produces (2xx, zero-payload). The examples have the same byte count but different status classes. A shortcut that classifies from bytes alone cannot replace the pasted route: it has discarded a distinction required by D.

Diagnostic procedure

1. Declare the six object types and type every transformation before comparing values.
2. Run the left-cell test on each raw line; a residual points to extraction, parsing, projection, or field typing.
3. Run the right-cell test on each parsed record; a residual points to normalization, projection, classification, or summarization.
4. Run the outer test on the same record identity and equality policy; it checks the composed contract seen by downstream diagnostics.
5. If both local tests pass but an independently computed outer test fails, inspect record alignment, fixtures, versioning, and equality policy before changing the mathematics.

The worked artifact check in Chapter 31 applies this pasted diagram to public log records and, equally important, records which causal arrows the artifact cannot justify.

Approximate pasting and residual bounds

The preceding example uses exact agreement. In other investigations, two routes may differ because of rounding, clock calibration, or numeric conversion. We then need more than a statement that each local discrepancy is small. We need to know how a later transformation changes an earlier discrepancy.

Consider the arrangement of Figure 5A in a separate timing model, and let the final values be reported durations in milliseconds. The comparison at the shared boundary uses seconds. Suppose the left comparison differs by at most 0.002 seconds. The transformation from that boundary to the final result converts seconds to milliseconds, and therefore multiplies that discrepancy by 1000. Suppose the right comparison contributes at most another 1 millisecond. The outer comparison may then differ by as much as 3 milliseconds.

This calculation requires the same record identity, compatible units, and a bound on the amplification introduced by the shared downstream transformation. Adding 0.002 and 1 without converting the units would not give a meaningful residual. Applying an arbitrary tolerance independently to each square would not give the required outer bound either.

We record the local residual limits and the downstream amplification bound with the invariant card. Appendix B.22 derives the outer bound by inserting the shared intermediate result and using the triangle inequality. For a metric comparison, zero local residuals recover exact agreement; a pseudometric comparison gives only the declared zero-distance equivalence.

A threshold classifier requires additional care. Two durations can be numerically close and still lie on opposite sides of the threshold. In that case, a small timing residual does not guarantee the same status. We either compare the numeric values before classification, require a justified margin from the threshold, or specify a separate relation on the resulting statuses.

Lawvere's work connects distance with categorical composition (Lawvere 1973). Robinson's sheaf-consistency work also studies quantitative disagreement (Robinson 2020). Our two-square calculation is an elementary diagnostic application of error propagation; it is not a new result about enriched categories or a proof that every tolerance policy composes.

Part II — Comparing and Combining Evidence

Chapter 6 — Natural transformations and cross-view consistency

> ***Diagnostic question:*** *How can two complete observation methods be compared point by point without confusing their different representations?*

Transforming one functor into another

Let $F, G: \mathcal{C} \to \mathcal{D}$ be functors. A natural transformation $\eta: F \Rightarrow G$ assigns an arrow $\eta_x: F(x) \to G(x)$ to every object x of $\mathcal{C}$ so that, for every arrow $f: x \to y$, the naturality square commutes:

$$G(f) \circ \eta_x = \eta_y \circ F(f).$$

In plain language, we may first transform the view at x and then follow the G-view of the transition, or first follow the F-view and then transform at y. The result should agree.

The concept was introduced as "natural equivalence" in the paper that helped found category theory (Eilenberg and Mac Lane 1945). Its diagnostic value is that it compares *systems of mappings*, not isolated values.

Logs and traces usually have different codomains

We may be tempted to write a natural transformation directly from a log functor to a trace functor. Usually that is ill-typed: logs and traces inhabit different evidence categories. We first need declared adapters into a common category $\mathcal{K}$:

$$K_L = N_L \circ L, \qquad K_T = N_T \circ T,$$

where $L: \mathcal{E} \to \mathcal{L}$, $T: \mathcal{E} \to \mathcal{T}$, $N_L: \mathcal{L} \to \mathcal{K}$, and $N_T: \mathcal{T} \to \mathcal{K}$. A candidate natural transformation is then

$$\eta: K_L \Rightarrow K_T.$$

The common category might represent normalized request outcomes, causal milestones, resource identities, or a diagnostic ontology. Mapping both modalities to strings is not sufficient; the arrows and composition in $\mathcal{K}$ must carry the semantics being compared.

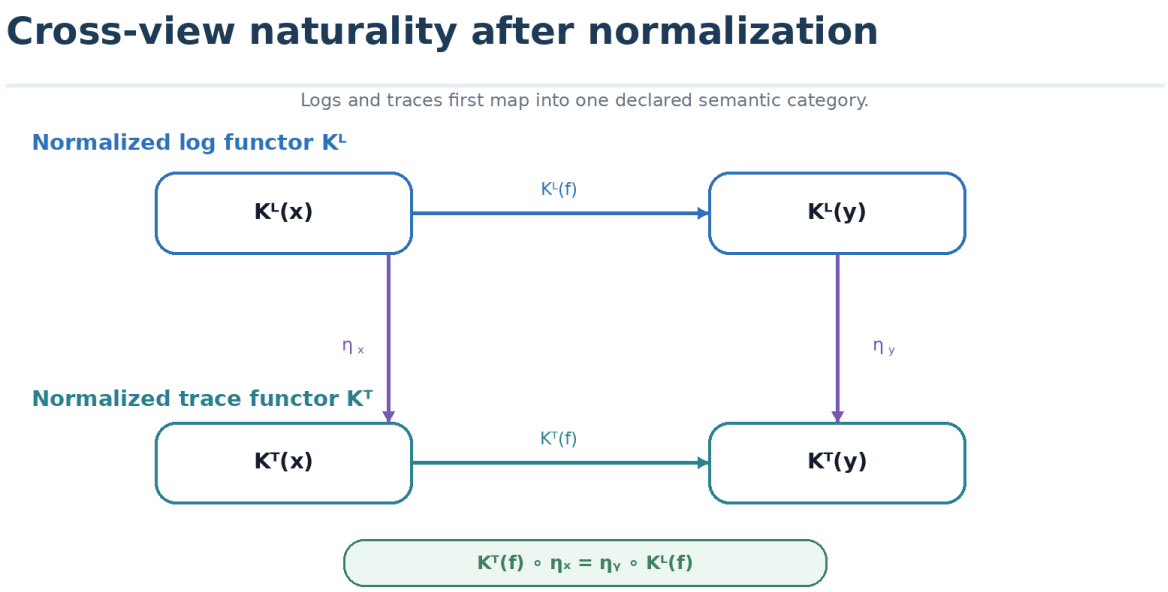


Figure 6. Cross-view comparison becomes well typed only after modality-specific evidence is mapped into a declared common category.

Naturality as an observability contract

Suppose $f: x \to y$ is a request transition. The log view records semantic events; the trace view records spans. Components η_x and η_y translate normalized log states into normalized trace states. Naturality says that translating before or after the transition gives the same result.

A failure may reveal:

- a missing or duplicated event;
- a span boundary inconsistent with the logged operation boundary;
- context attached to the wrong request;
- different semantic versions in the adapters;
- a transition visible to one modality but outside the other's declared domain.

The last case may be correct partiality, not a defect. In the contract, we must state the shared domain.

Naturality residuals

When $\mathcal{K}$ supports a distance on parallel arrows, define

$$r_\eta(f) = d\left(K_T(f) \circ \eta_x,\ \eta_y \circ K_L(f)\right).$$

This residual can be aggregated by service, operation, build, or semantic-convention version. A rise after deployment is **cross-view drift**. Local residuals identify which execution arrows no longer transform consistently.

One practical implementation compares canonical event sequences:

```
def naturality_residual(edge, log_view, trace_view, eta, distance):
  left  = compose(trace_view(edge), eta(edge.source))
  right = compose(eta(edge.target), log_view(edge))
  return distance(left, right)
```

The function names hide important design work. compose must be typed, eta must be versioned, and distance must be meaningful for the codomain.

Multi-view consistency

More than two views can be organized as a cone into a shared diagnostic model. Logs, traces, metrics, dumps, and profiles each have an adapter to a category of diagnostic entities and relations. Pairwise naturality tests are useful, but a shared cone prevents a quadratic maze of ad hoc converters.

This formalizes the practical role of Trace Viewpoints, which distinguish error, use-case, architectural, implementation, non-functional, and signal/noise perspectives (Vostokov 2026, 344–45). Different viewpoints need not contain the same objects; comparison proceeds on declared overlap and through declared transformations.

Practical recipe

1. Select one invariant shared by two modalities, such as request outcome or resource identity.
2. Define each modality's category and the common category.
3. Implement and version the two normalization functors.
4. Define the transformation components at each object type.
5. Generate naturality squares for a representative edge set.
6. Measure failures and decompose by component, version, and missingness cause.

Chapter 7 — Products, pullbacks, and evidence joins

> ***Diagnostic question:*** *When two artifacts share a key or entity, what exactly is the combined object, and what does the join exclude?*

Product: combine without a matching constraint

The product $A \times B$ comes with projections to A and B and is universal among objects that map to both. In sets, it is the set of all pairs. In diagnostics, we use a plain product to represent all combinations of two evidence collections, but it is rarely the final correlation because it does not require relatedness.

A dashboard that displays CPU and latency side by side is product-like at the presentation level. It does not by itself assert that a particular CPU sample caused or even belongs to a particular request.

The trace reference's **Cartesian Trace** is this unconstrained product: messages in a long trace are paired with independent configuration traces, without a matching condition (Vostokov 2026, 69). The source contrasts it with dependent constructions such as Trace Presheaf and Trace Extension. The distinction is operational: use a product to enumerate combinations, and introduce a pullback or another fibered construction only after the dependency map is declared.

Pullback: combine over a shared object

Given maps $a: A \to Q$ and $b: B \to Q$, their pullback $A \times_Q B$ contains compatible pairs that map to the same object in Q. In sets:

$$A \times_Q B = \{(x, y) \mid a(x) = b(y)\}.$$

We can read this as the categorical shape of an equijoin. Q might be a category or set of trace contexts, process identities, resource identities, build versions, or normalized semantic events.

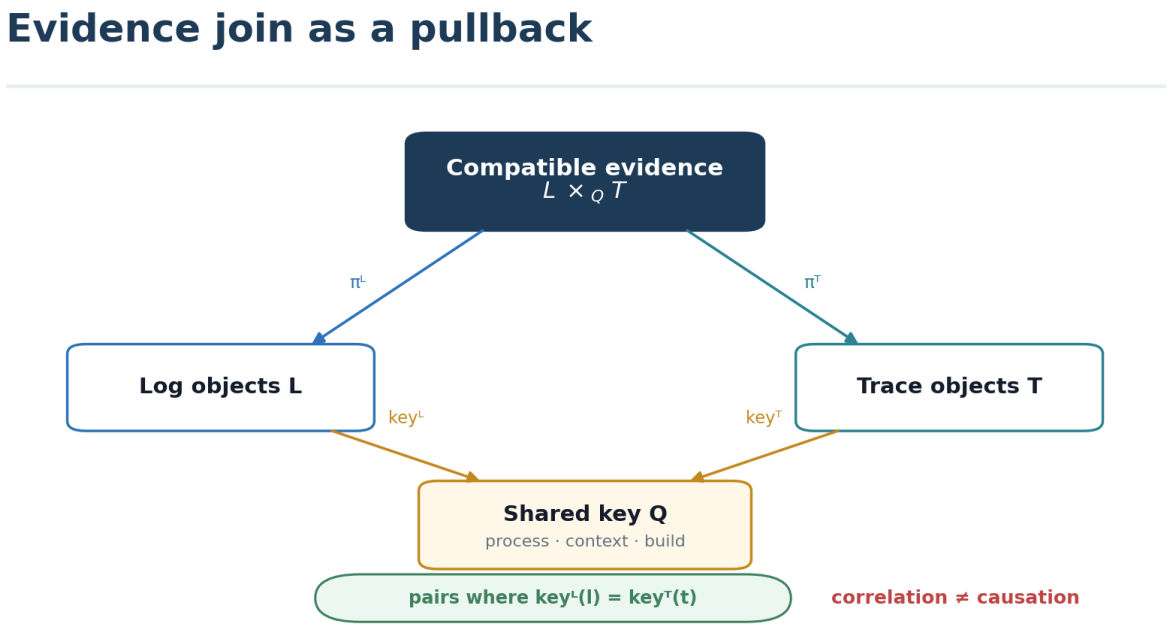


Figure 7. A pullback correlates evidence through a declared shared key; it is not a claim of causality.

The pullback clarifies three things that an informal "join the logs and traces" instruction hides:

- the two source mappings into the key space;
- the equality or equivalence used by the key space;
- the universal combined object created by compatible pairs.

The Trace Join pattern similarly selects messages from different traces according to a condition such as a shared activity identifier or feature (Vostokov 2026, 321). The categorical account makes the join's mediating object explicit.

Pullbacks do not create causality

Two rows with the same trace_id are correlated through that identifier. The pullback says they are compatible under the key maps, but it does not establish that one caused the other. Causality requires additional arrows, temporal constraints, code semantics, interventions, or a causal model (Pearl 2009).

With this distinction, we prevent a common failure: treating a correlation key as a causal edge. The key says "same declared context," not "producer of."

Weak and approximate keys

Real joins use fallbacks: timestamp windows, host names, message templates, or inferred request flows. We can model these with a relation rather than equality, an enriched category, or a span through an uncertainty object. The output should carry the strength of the match.

A reused process identifier can make separate process lifetimes look like one activity—the trace reference's **Glued Activity** pattern (Vostokov 2026, 146). We therefore prefer a process-lifetime key that includes host or workload

identity, start or boot identity, generation, and interval. When both artifacts are calibrated total orders, the Galois Trace construction in Chapter 9 supplies a different operation: a principled next/previous alignment, not a claim that timestamp proximity establishes causality.

A good correlation result is not merely a row pair. It is a pair plus provenance:

```
left_object
right_object
mediating_key
key_version
match_rule
confidence_or_tolerance
unmatched_reason
```

Correlation witnesses and categorical spans

When a source value has several possible storage locations, choosing one location too early may hide the reason for a later mismatch. The same problem appears in trace correlation when a weak key connects one message with several possible activity instances. In both cases, a relation can record the possible pairs, but the pairs alone may not explain how the association was obtained.

We therefore retain a correlation witness for each supported association. Such a witness may include the symbol record, build identity, capture identity, address range, time interval, and the rule used to connect the records. The fields depend on the question. Their purpose is to let us inspect the association without reconstructing it from an unexplained endpoint pair.

Consider a source-value view, a storage-location view, and a dump-artifact view. One collection of witnesses connects source values with locations. Another connects locations with dump artifacts. We compose these collections by matching their location keys and retaining both witnesses in every matched pair. A location key must include the required address-space, lifetime, and capture context. Matching only a numeric address may combine unrelated observations.

This is a span construction in sets. The witness set maps to the two endpoint sets, and the composition uses the pullback already introduced in this chapter. If we subsequently keep only the endpoint pairs, we obtain a relation and forget which intermediate witnesses produced those pairs. That loss may be acceptable for a summary, but not for an analysis that must explain a symbolization or correlation failure.

For example, two location paths may connect the same source value with the same dump artifact. The endpoint relation contains one pair, whereas the retained construction contains two witness pairs. They are two recorded ways to obtain the association, not automatically two independent confirmations. Duplicate records and alternative explanations should remain distinguishable.

For discrete source and target categories, the witness sets can also be arranged as a set-valued profunctor. Appendix B.24 explains this restricted case as a table of sets. If we add nonidentity arrows to the endpoint categories, we must also specify how the witnesses transform along those arrows. A many-to-many table by itself does not provide those additional laws.

This construction gives a precise continuation of our earlier discussion of Software Codiagnostics and the Trace Join analysis pattern (Vostokov 2024b, 275–76; 2026, 321). The categorical background is standard (Fong and

Spivak 2019; Riehl 2016). The diagnostic choice is to retain the evidence for the association instead of only its endpoints.

Equalizers for agreement

If two transformations $f, g: A \to B$ should agree on some inputs, the equalizer selects the subobject where $f(x) = g(x)$. For example, one function derives an HTTP outcome from a terminal log and another from a root span, and the equalizer is the set of requests where the derivations agree. Its complement is a direct worklist for diagnostic investigation.

Unlike a global mismatch count, the equalizer retains the input objects and supports further slicing by service, version, or path.

Practical recipe: correlation before interpretation

1. Identify the shared object Q.
2. Specify both maps into Q.
3. Test key uniqueness, stability, and propagation.
4. Materialize matched and unmatched sides separately.
5. Preserve the match rule in the result.
6. Add causal or semantic arrows only in a subsequent, justified step.

Chapter 8 — Limits, colimits, and evidence fusion

> ***Diagnostic question:*** *How can several partial views be combined while preserving the constraints among them?*

Limits as compatible families

A diagram is a category-shaped collection of objects and arrows, and a cone to that diagram is an object that maps compatibly to every object in it. A limit is a universal such cone: every other compatible cone factors uniquely through it.

Products, pullbacks, and equalizers are special limits. The general concept matters when evidence has more than one compatibility condition. Suppose logs, traces, configuration, and deployment metadata all map to resource identities and version identities. The limit consists of tuples that satisfy all declared relationships simultaneously.

In set-like settings, the limit is a constrained join. In a richer category, it preserves the structure represented by the diagram. It is therefore a useful model for a **diagnostic evidence assembly**.

Missing pieces and empty limits

An empty result may mean no compatible family exists. That can be diagnostic: trace context says build 42, deployment metadata says build 43, and the log schema is valid only for build 41. It can also mean the mappings or data are incomplete. In the model, we must distinguish inconsistency from missingness.

One approach adds explicit unknown objects and arrows, and another calculates limits only over the observed subdiagram. The choice affects whether absence becomes a contradiction, an unknown, or a restricted scope.

Colimits as assembly

Colimits reverse the direction, and they assemble a universal object from pieces under identifications. Coproducts combine alternatives, and pushouts glue two objects along a shared part.

A pushout can model schema or ontology merging. Two evidence schemas share a common core K; the pushout forms a combined schema in which corresponding elements of K are identified. Functorial data migration and categorical database work provide formal mechanisms for schema mapping, projection, union, and join (Spivak 2012, 2014b).

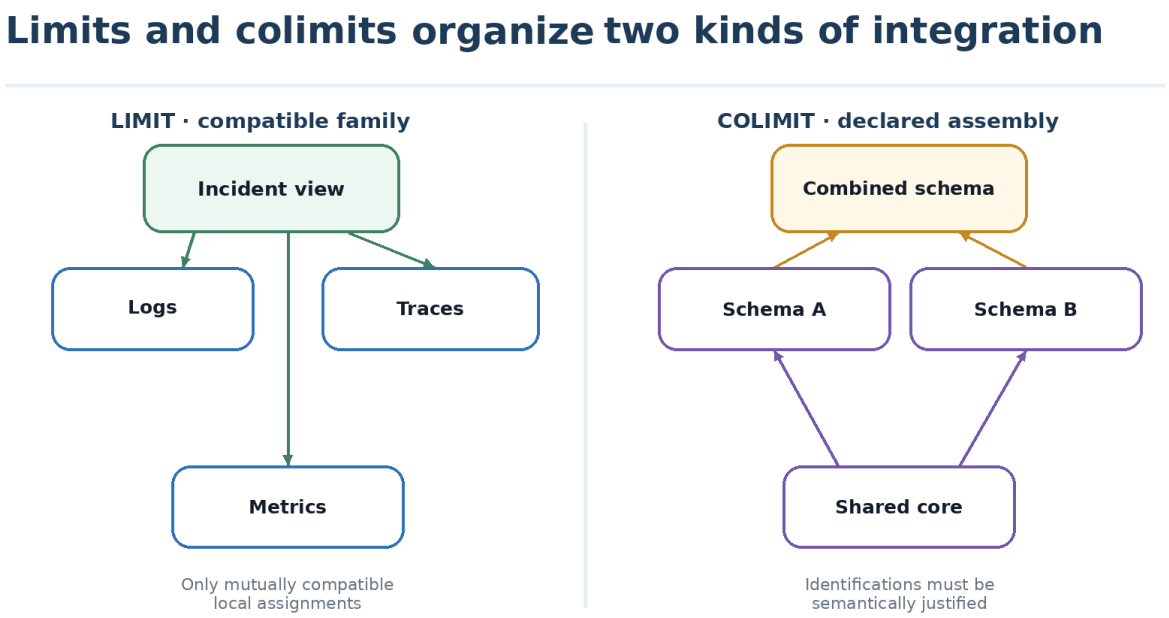


Figure 8. Limits assemble compatible evidence; colimits assemble representations by declared identifications.

A warning about semantic conflict

A pushout performs the identifications specified by the maps, but it does not decide that two similarly named fields mean the same thing. If `status` means transport status in one schema and business outcome in another, identifying them creates a bad combined model with mathematical precision.

The semantic work must happen in the common schema K and in the arrows from K. This is where ologs and explicit ontologies help: objects and arrows are labeled with meaningful phrases and commutative facts (Spivak and Kent 2012).

Root cause is not automatically a limit

Universal constructions are attractive, and we can easily call the "best explanation" a terminal object or limit. Such a claim requires a category of explanations and arrows that encode a precise comparison, such as logical implication, factorization, or minimal sufficient support. Without that category, the phrase is only an analogy.

A safer use is operational: build a category whose objects are explanation candidates and whose arrows are verified refinement steps. Then ask whether a universal object exists. In many incidents it will not, and several incomparable explanations may remain.

Practical recipe: build the constraint diagram first

For a multi-artifact query:

1. Draw every evidence source as an object.
2. Draw only verified mappings and constraints.
3. Mark required and optional subdiagrams.

4. Compute compatible tuples for the required limit.
5. Record which constraint removes each rejected tuple.
6. Treat the rejection explanation as diagnostic output.

The last step is important: a failed join is often more informative than the joined rows.

Chapter 9 — Adjunctions, views, and reversibility

> ***Diagnostic question:*** *When does a "forward" diagnostic transformation have a principled best reverse operation?*

Adjunction is stronger than opposition

Two activities may feel dual (instrumenting and interpreting, expanding and summarizing, collecting and querying) without being adjoint. An adjunction $F \dashv G$ requires a natural correspondence

$$\mathrm{Hom}_{\mathcal{D}}(F(c), d) \cong \mathrm{Hom}_{\mathcal{C}}\big(c, G(d)\big).$$

We require the correspondence to be natural in both variables: this is a precise universal relationship, not merely a trade-off or reverse arrow.

A genuine example: free paths and underlying graphs

The free-category functor turns a directed graph into a category of paths, and the forgetful functor sends a category to its underlying directed graph. The free functor is left adjoint to the forgetful functor: graph mappings from G to the underlying graph of C correspond to functors from the free category on G to C.

Diagnostically, we use this to explain why an event graph can be lifted into a compositional path model without inventing extra equations. Any mapping of primitive events into a target category extends uniquely to paths.

Data migration adjunctions

For a schema functor $F: S \to T$, categorical database theory provides a pullback-like migration Δ_F and left and right adjoints $\Sigma_F \dashv \Delta_F \dashv \Pi_F$ under appropriate conditions (Spivak 2012). These correspond, roughly, to different ways of moving instances along a schema mapping.

Here we have a direct connection to observability pipelines. A semantic-convention migration is not just a field rename: it moves structured instances between schemas. Adjoints explain canonical ways to aggregate, extend, or constrain data around the mapping.

Views and bidirectional transformations

Model-driven engineering studies mappings between source models and views, including update propagation and synchronization (Diskin and Maibaum 2012). Diagnostics uses similar structures:

- a raw trace and a filtered view;
- a memory dump and a decoded object graph;
- a log schema and a normalized event schema;
- a runtime configuration and a human-edited desired-state view.

Reversing a view is generally underdetermined. Many raw traces have the same summary. A lawful update operation needs complement information, constraints, or an explicit policy. Calling filtering "reversible" because the original file still exists confuses external storage with invertibility of the transformation.

Adjoint Thread and categorical adjunction

The trace-pattern term **Adjoint Thread of Activity** names a trace view obtained by restricting on an attribute other than the conventional thread identifier—module, file, function, operation, or process identifier (Vostokov 2024b, 371–75; 2026, 48–52). Let T be the set of trace messages and let $A: T \to V$ be one total attribute column. That school-level function induces three monotone maps between the subsets of T and the subsets of V:

$$\exists_A \dashv A^* \dashv \forall_A.$$

Here A^* takes selected values back to their messages, $\exists_A$ reports which values occur in a selected message set, and $\forall_A$ reports the values whose entire fibers lie in that set. The adjunctions are standard mathematics (Davey and Priestley 2002). Identifying source patterns with these maps is our **diagnostic construction**.

$$\exists_A(S) \subseteq X \;\Leftrightarrow\; S \subseteq A^*(X), \qquad A^*(X) \subseteq S \;\Leftrightarrow\; X \subseteq \forall_A(S).$$

A Thread of Activity is the fiber $\mathrm{TID}^*(\{t\})$; an Adjoint Thread is $A^*(\{v\})$ for another column. The nonempty fibers partition T, supplying candidate subtraces for Sheaf of Activities and Message Cover. Equality of attribute values is an equivalence relation on messages; its quotient and image factorization are what Quotient Trace and Adjoint Message read off. The source's compact formula "(Thread A, B) = [A, Thread B]" (Vostokov 2024b, 98) therefore receives a literal order-theoretic reading—not because the name guarantees an adjunction, but because the two subset inclusions above are equivalent. Appendix B derives the result using only functions, sets, and inclusion.

Galois Trace after clock alignment

The trace reference's **Galois Trace** moves between two traces in opposite directions and relates the move to monotone maps; Calibrating Trace supplies the prior clock alignment (Vostokov 2026, 68, 144). Under one explicit finite model, the analogy becomes a theorem. Let T_1 and T_2 be nonempty traces ordered by calibrated time, with ties resolved by each trace's stable sequence number. Form $P = T_1 \cup \{\bot\}$ and $Q = T_2 \cup \{\top\}$.

Define $F(\bot) = \min T_2$. For $a \in T_1$, let $F(a)$ be the earliest message of T_2 whose calibrated time is not earlier than $t(a)$, or $\top$ when no such message exists. Define $G(\top) = \max T_1$. For $b \in T_2$, let $G(b)$ be the latest message of T_1 whose calibrated time is not later than $t(b)$, or $\bot$ when no such message exists. Then

$$F(p) \le q \;\Leftrightarrow\; p \le G(q), \qquad \text{so} \qquad F \dashv G.$$

The unit $p \le G\big(F(p)\big)$ says that a round trip never lands before its starting message; $G \circ F$ is therefore a closure operator on P. The counit $F\big(G(q)\big) \le q$ gives the complementary boundary check on Q. This is **standard mathematics** after the orders and alignment are fixed (Davey and Priestley 2002). Choosing the clock calibration, tie policy, and sentinel meanings is a **diagnostic construction**. Clock skew is a residual, and the adjunction aligns observations; it does not establish causality. Appendix B verifies every boundary case and works a small example.

Practical test for an adjunction claim

Before describing F and G as adjoint, answer:

1. What are the two categories?
2. What are the two functors?
3. What are the hom-sets on each side?
4. What is the bijection between them?
5. Why is the bijection natural?

If the answer is only "the operations go in opposite directions," use paired transformations or a dual workflow instead. The attribute-map and Galois Trace constructions pass because their categories, functors, order-hom correspondences, and monotonicity are explicit.

Chapter 10 — Software codiagnostics and functorial data transformation

> ***Diagnostic question:*** *How do transformations of evidence participate in diagnosis without being mistaken for neutral preprocessing?*

Codiagnostics

Software diagnostics often reveals indicators only after the artifact is transformed. For example, a large log is filtered, sorted, normalized, windowed, joined, quotiented, or enriched. *Theoretical Software Diagnostics* calls these cooperative and transformation-oriented activities **software codiagnostics** or **data codiagnostics** (Vostokov 2024b, 275–76).

The prefix is useful because the transformation does not merely prepare data for a later "real" diagnosis. It changes which patterns become visible. Sorting by thread reveals one structure; grouping by function reveals an Adjoint Thread; taking a quotient by message template reveals multiplicities and shape; joining with deployment metadata reveals version boundaries.

A category of artifact versions

Let objects be artifact states:

$$A_0 \xrightarrow{\text{parse}} A_1 \xrightarrow{\text{normalize}} A_2 \xrightarrow{\text{filter}} A_3 \xrightarrow{\text{join}} A_4.$$

We define arrows as reproducible transformations with parameters and versions. Composition is pipeline composition, and the identity leaves the artifact unchanged. In this way, we preserve provenance for every derived view.

Analysis patterns may then be functors from artifact categories to indicator categories. This develops the earlier categorical foundation in which concrete analysis patterns act as functors between execution artifacts and problem patterns (Vostokov 2024b, 271–72).

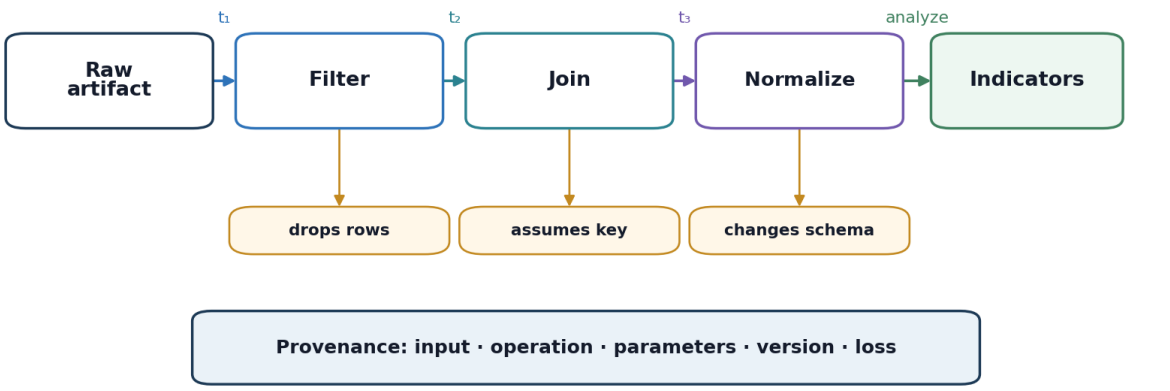


Figure 9. Codiagnostic transformations form a provenance path; each step can reveal indicators and can also lose distinctions.

Every transformation has a loss contract

For each arrow, we record:

- preserved identities and order;
- merged or discarded fields;
- newly inferred fields;
- determinism and dependence on external state;
- error and missingness behavior;
- inverse, complement, or provenance availability.

A quotient by message template intentionally discards variable payload, and a sort changes adjacency. A window truncates context, and a join may duplicate rows. A symbolization step depends on binary and symbol versions. None is neutral.

Functorial pipeline laws

If a direct converter $C_{02}: A_0 \to A_2$ is advertised as equivalent to normalize(parse(A0)), test the commuting triangle:

$$C_{02} \stackrel{?}{=} C_{12} \circ C_{01}.$$

This detects toolchain drift. Similar tests compare an online stream processor with an offline batch reconstruction, or a vendor exporter with a reference normalizer.

Pattern composition

Pattern-oriented diagnostics becomes more powerful when patterns compose. Sequential composition chains transformations. Branch-and-merge applies several patterns to a common input and then correlates their outputs. Refactoring replaces a composite with an equivalent pattern. Evidence fusion combines pattern outputs through limits or colimits.

The first question is typing: does the output artifact type of one operation match the input type of the next? The second is preservation: which invariants survive the composite? The third is equivalence: does a replacement commute with the rest of the workflow?

These questions prepare the operadic account in Chapter 15.

Practical recipe: provenance-first analysis

We treat every derived artifact as an object with:

```
artifact_id
source_ids
transformation_name
transformation_version
parameters
schema_version
time_created
preservation_profile
content_hash
```

When two analyses disagree, compare their paths before comparing their conclusions. Many “diagnostic disagreements” are codiagnostic path differences.

Part III — Time, Behavior, Context, and Higher Structure

Chapter 11 — Debugging as a presheaf

> ***Diagnostic question:*** *How can a debugger move from broad evidence to local scopes while preserving the restriction relationships among views?*

From "going backwards" to contravariance

Debugging often moves against the direction of execution. From a failure, we walk back through logs, stack frames, causal parents, configuration changes, and earlier states. *Theoretical Software Diagnostics* proposes that debugging can be viewed as a presheaf: a contravariant functor that assigns sets of artifacts to execution or time objects (Vostokov 2024b, 319–30).

To make the idea formal, we need more than the phrase "go backwards." A presheaf on a category $\mathcal{B}$ is a functor

$$P: \mathcal{B}^{op} \to \mathbf{Set}.$$

The objects of $\mathcal{B}$ can be diagnostic scopes: time intervals, trace regions, component sets, address regions, or combinations. An arrow $U \to V$ may mean that U is included in V. Contravariance then supplies a restriction map

$$\rho_U^V: P(V) \to P(U),$$

which takes evidence valid on the larger scope and restricts it to the smaller one.

Why scopes are better than isolated instants

If the base category contains only time instants, the restriction operation may be unclear: what exactly is restricted from one instant to another? Intervals or causally closed regions usually provide a better base. A log window can be restricted to a subwindow; a memory-region interpretation can be restricted to a subregion; a configuration model can be restricted to a service.

The base category is part of the diagnosis, and a temporal base supports back-tracing. A component base supports subsystem isolation, and a product base supports simultaneous time-and-component zooming.

The trace reference already gives this base two operational names. A **Trace Window** is a horizontal time scope or a vertical attribute scope, while a **Flag** is a chain of message sets or activity regions ordered by inclusion (Vostokov 2026, 140, 346). Inclusion is standard order theory; reading the chain as a path along which a presheaf restricts is our diagnostic construction.

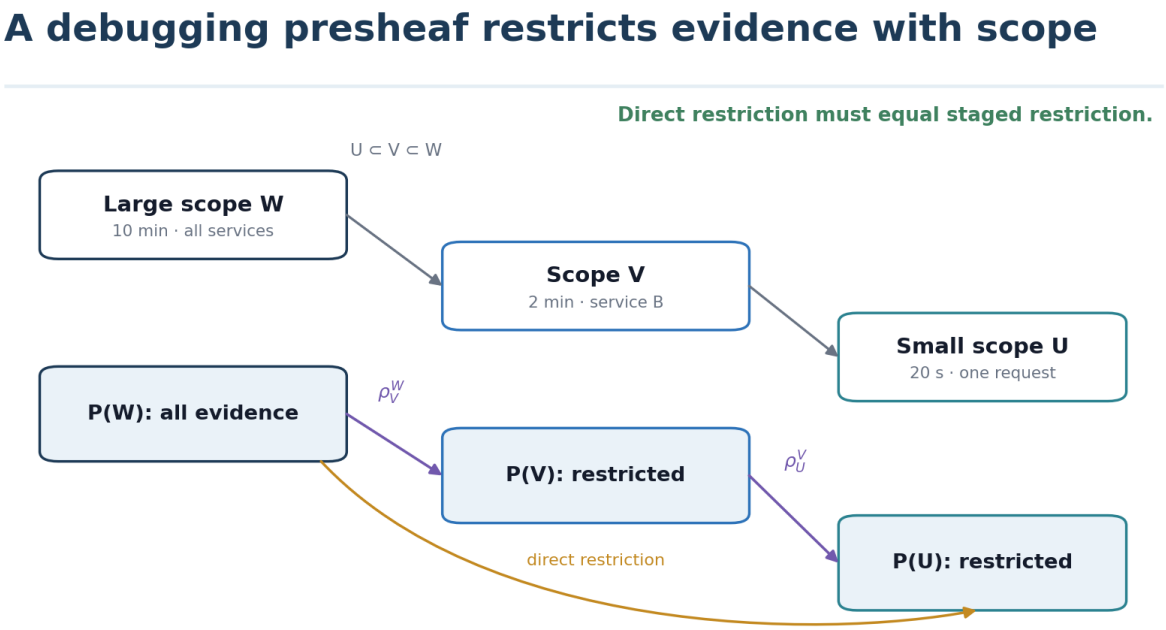


Figure 10. A presheaf assigns evidence to diagnostic scopes and supplies restriction maps when the scope narrows.

Presheaves of observations, hypotheses, and commands

Several presheaves can share the same base:

- $P_{obs}(U)$ — observations available in scope U;
- $P_{hyp}(U)$ — hypotheses meaningful in U;
- $P_{cmd}(U)$ — diagnostic commands applicable in U;
- $P_{code}(U)$ — code or configuration fragments associated with U.

A natural transformation between P_{obs} and P_{code} can model systematic navigation from evidence to source locations, provided the transformation respects restriction. This captures the practical idea in the earlier article: moving backward through a log while simultaneously browsing the associated source or stack information (Vostokov 2024b, 328–30).

Restriction laws as tests

Presheaf laws require that restricting to the same scope changes nothing and that a two-step restriction equals direct restriction. If $W \subseteq U \subseteq V$:

$$\rho_W^V = \rho_W^U \circ \rho_U^V.$$

A trace viewer can test this. Filtering a full trace directly to service B and interval W should equal filtering first to interval U and then to service B and W, if the filters are claimed to be pure restrictions. A mismatch may reveal stateful query behavior, boundary semantics, inconsistent time normalization, or a filter that is not actually a restriction.

Extension is not restriction

Debuggers also extend context by loading symbols, joining configuration, or retrieving a parent trace. Extension is not supplied automatically by a presheaf. It may be an additional operation, a left or right adjoint under suitable conditions, or a separate functor. Keeping the directions separate prevents a convenient enrichment from being misdescribed as “going back.”

Practical recipe: restriction-stable views

1. Define a partial order of scopes.
2. Assign evidence sets or structured evidence to each scope.
3. Implement restriction maps.
4. Test identity and composition of restrictions.
5. Record enrichments separately from restrictions.
6. Use failed restriction laws as viewer or query-engine defects.

Chapter 12 — Sheaves and local-to-global diagnosis

> ***Diagnostic question:*** *When can locally consistent evidence be assembled into one global account, and where does that assembly fail?*

From presheaf to sheaf

By a sheaf, we mean a presheaf with gluing and uniqueness conditions. Informally, compatible local sections over a cover can be glued into a unique global section: this is a precise local-to-global principle.

For diagnostics, we may use the following local sections:

- per-service request narratives;
- per-thread activity sequences;
- per-host configuration views;
- per-region memory interpretations;
- per-window signal estimates;
- per-team domain explanations.

Overlaps contain shared request context, messages, resources, or state. If local accounts agree on overlaps, a sheaf model permits a global account. If they do not, the obstruction is diagnostic.

A small distributed example

Service A records request `r` as sent to B. Service B records `r` as received and sent to C. Service C records `r` as received and failed. Pairwise overlaps are the A–B and B–C boundaries. The sections glue when request identity, ordering, semantic operation, and relevant status agree on each overlap.

If B uses a reused identifier or an incompatible semantic version, local narratives may each look valid while failing to glue. The failure is more specific than “the trace is incomplete”: it identifies the overlap and restriction maps at which consistency breaks.

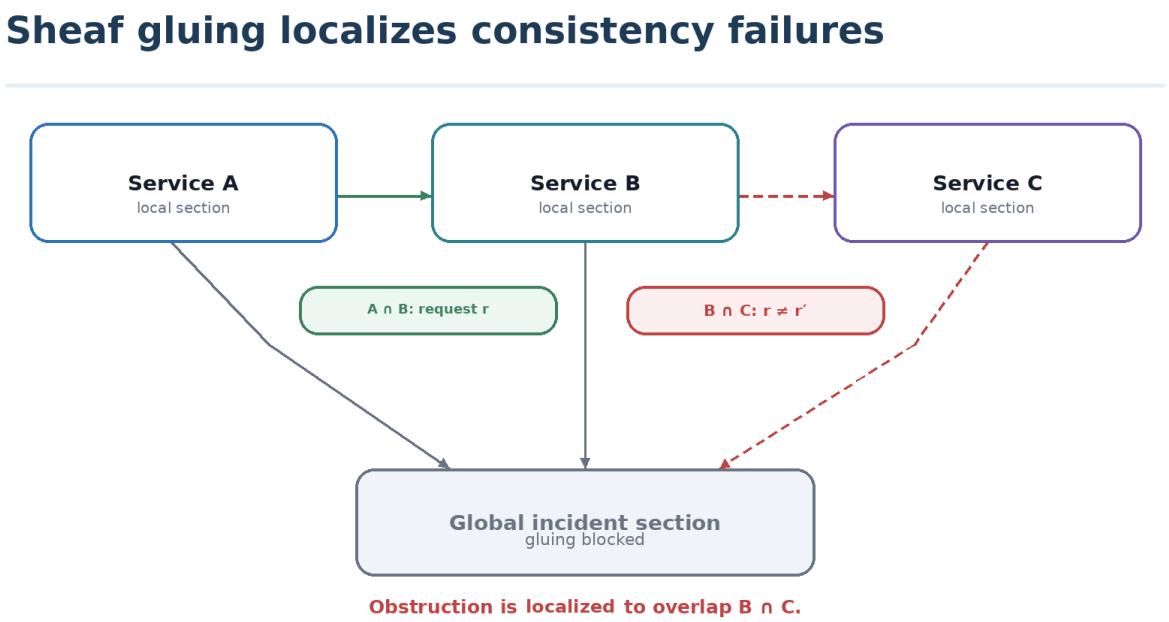


Figure 11. Compatible local evidence glues to a global section; disagreement on an overlap localizes the obstruction.

Sheaf of Activities and Trace Presheaf

The **Sheaf of Activities** pattern constructs subtraces from message types such as operation or function name, compares the corresponding fibers in normal and problematic logs, and uses their local bifurcation points in the full traces (Vostokov 2026, 266–67). In the language of Chapter 9, these subtraces are fibers $A^*(\{v\})$ of an attribute column. The fiber partition is therefore an explicit candidate cover. The **Trace Presheaf** maps trace messages or message sets to associated memory objects or state information (2026, 330).

The partition supplies a cover, not the sheaf axioms. Because the nonempty fibers of one column are disjoint, they create no nontrivial overlap-agreement conditions; for a set-valued candidate presheaf S, the sheaf condition reduces to asking whether the canonical map $S(T) \to \prod_{v \in \mathrm{im}(A)} S\left(A^*(\{v\})\right)$ is a bijection. Nontrivial overlaps arise when fibers of different columns, or windows, are combined, as in Message Cover. A formalization must still define the sections, restriction maps, and gluing operation, and must test separation and existence. The source pattern is thus concrete input to a diagnostic sheaf construction rather than evidence that every collection of subtraces is already a sheaf.

Approximate consistency

Telemetry rarely agrees exactly. Clocks differ, values are sampled, and schemas round or aggregate. Robinson's sheaf-based sensor integration models quantify consistency across heterogeneous sources, and assignments to sheaves of pseudometric spaces support a continuous consistency radius robust to noisy data (Robinson 2017, 2020).

A simpler software diagnostic measure, inspired by the consistency-radius idea, compares pairwise disagreement among normalized local sections. For a cover $\{U_i\}$ with observations s_i, compare restrictions on every overlap $U_i \cap U_j$:

$$r = \max_{i,j} d\left(\rho^{U_i}_{U_i \cap U_j}(s_i), \rho^{U_j}_{U_i \cap U_j}(s_j)\right).$$

This pairwise statistic is not Robinson's consistency radius, which is defined against an assignment along restriction maps. The maximum is one operational choice, and a robust percentile or weighted aggregation may be more appropriate. The metric and aggregation are domain decisions.

Missing overlap and false globality

If two services propagate no shared context, their local sections may have no observable overlap. A global narrative produced by timestamp proximity is then an inferred extension, not a sheaf gluing based on agreement. The result should carry that weaker provenance.

This distinction is central to observability. A system can have abundant local telemetry and still lack the overlap data required for global reconstruction.

From sheaves to a diagnostic topos

A presheaf and a sheaf do not stand alone. Let $\mathcal{B}$ be a small category of diagnostic scopes and scope inclusions. The category of all set-valued presheaves on $\mathcal{B}$ is written

$$\widehat{\mathcal{B}} = [\mathcal{B}^{\mathrm{op}}, \mathbf{Set}].$$

This presheaf category is a **Grothendieck topos**. When J declares which families of arrows cover each scope, $(\mathcal{B}, J)$ is a **site**, and the sheaves satisfying that cover policy form another Grothendieck topos:

$$\mathbf{Sh}(\mathcal{B}, J).$$

Calling J an **observational-coverage policy** is our diagnostic interpretation. For example, the whole request scope may count as adequately covered only when caller evidence, downstream-service evidence, and their shared boundary are present. The usual cover laws require that full coverage covers, that coverage remains valid after restriction to a smaller scope, and that locally adequate covers compose. These laws test the policy; they do not let us call an arbitrary group of artifacts a cover.

Naming the topos adds a precise account of hypotheses. A subobject $H \hookrightarrow E$ is a restriction-stable family of evidence states inside an evidence presheaf or sheaf E. Its characteristic map $\chi_H : E \to \Omega$ returns a contextual truth value: the scopes and refinements under which an evidence state lies in H. This is standard topos theory (Mac Lane and Moerdijk 1994; Borceux 1994). Appendix B explains the construction from subsets and functions.

Contextual truth is generally intuitionistic. Failure to establish a hypothesis globally is not automatically evidence for its negation. This is well matched to partial telemetry, but it is not a probability, fuzzy confidence, causal score, or root-cause result. Any such quantity needs additional declared structure. Appendix B explains sheafification and why it cannot recover missing telemetry.

Worked example: contextual support for a timeout hypothesis

Let U be the scope of one distributed request, A its caller region, B its downstream-service region, and $O = A \cap B$ their observed boundary. Arrows point from a narrower scope into a wider one. Let $E(V)$ be the normalized evidence states available on scope V, with restriction maps that discard evidence outside a smaller scope.

For each scope V, let $H(V) \subseteq E(V)$ contain the evidence states compatible with the hypothesis, "the downstream service exhausted the remaining deadline." We require compatibility to survive restriction; otherwise H is not a subpresheaf and the topos claim fails. For $e \in E(U)$, the characteristic map returns

$$\chi_{H,U}(e) = \{\, f : V \to U \mid E(f)(e) \in H(V) \,\}.$$

Suppose the restrictions to B and O are compatible with the hypothesis, while A contains an earlier local queue delay that is not. Then $\chi_{H,U}(e)$ contains the arrows from B, O, and every further refinement of those scopes, but it is not the maximal truth value at U. The result is a **sieve**: once a scope is included, every permitted narrower view is included as well.

The conclusion is deliberately limited: the hypothesis survives in the downstream branch but not over the whole request. It has contextual support, not a proven causal status. If the boundary scope O is missing and no section e over U can be formed, the earlier gluing test fails first; we record an evidence obstruction rather than forcing true or false.

The presheaf topos, sheaf topos, subobject classifier, and characteristic map are **standard mathematics**. Selecting $\mathcal{B}$, J, E, H, and the compatibility predicate is a **diagnostic construction**. Comparing two instrumentation regimes by a geometric morphism remains a research direction until both toposes, both functors, and their preservation laws are specified.

Practical recipe: an evidence consistency map

1. Define the scope category and the coverage policy: services, hosts, trace regions, or time windows.
2. Specify local section schemas.
3. Define restriction to each overlap.
4. Choose exact or approximate agreement.
5. Calculate per-overlap residuals.
6. Visualize the cover as a graph weighted by residual.
7. Investigate the largest obstruction before constructing a global story.
8. For a restriction-stable hypothesis, report its supporting sieve rather than collapsing contextual support to one Boolean value.

Chapter 13 — Coalgebra, behavior, and observational indistinguishability

> ***Diagnostic question:*** *When do two internal states count as the same because every declared observation gives the same behavior?*

Algebra builds; coalgebra unfolds

An algebra often combines or evaluates structure, and a coalgebra describes how a state unfolds into observations and successor structure. For an endofunctor F, an F-coalgebra is a map

$$c: X \to F(X).$$

The set or object X contains states. The functor F describes one observable step: output plus next state, branching successors, probability distribution, or another behavior shape. Coalgebra is a standard framework for transition systems, automata, streams, and dynamical behavior (Jacobs 2017).

The source describes a "behavior functor" from execution artifacts to diagnostic indicators (Vostokov 2024b, 271–72). That is a useful cross-category observation or analysis functor, but it is not by itself a coalgebra $c: X \to F(X)$: in a coalgebra, F is an endofunctor, so its behavioral shape is built in the same ambient category as X. We therefore separate the **system coalgebra**, which unfolds state and successor behavior, from the observation and analysis functors that map that behavior into evidence and indicators. The separation is the repair that makes both constructions well typed.

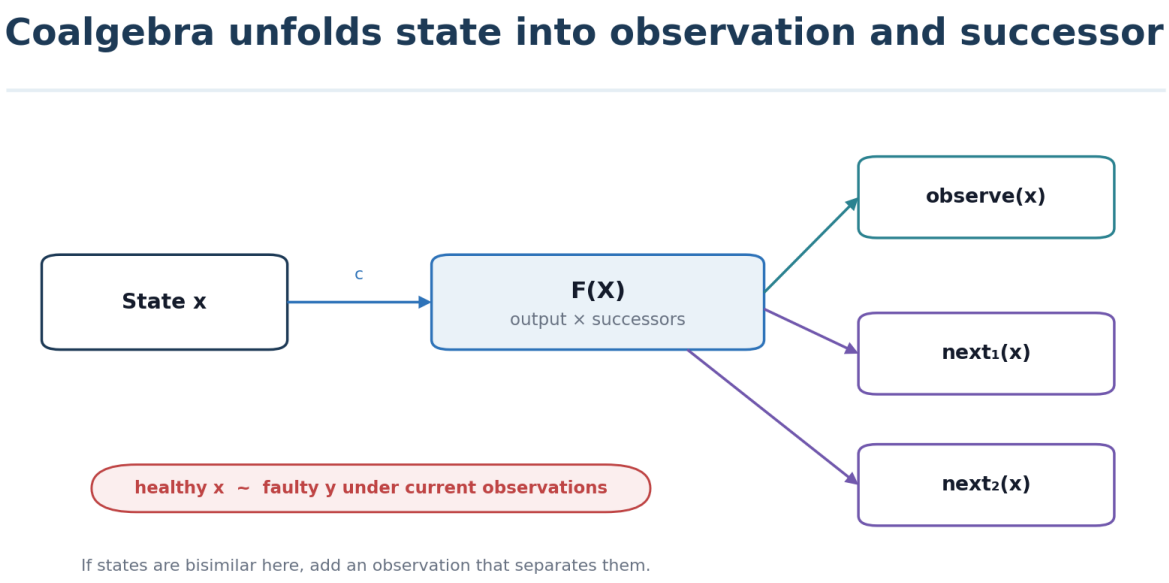


Figure 12. A coalgebra exposes one step of observable behavior and successor structure from a state.

Bisimulation and diagnostic blindness

Two states are behaviorally equivalent when the coalgebraic observations cannot distinguish them, often formalized through bisimulation. We find this both useful and dangerous.

If a monitoring model regards two internal states as bisimilar, an alert based on that model cannot distinguish them. If one state contains a latent resource leak and the other does not, the observation functor is too coarse for leak diagnosis even if it is adequate for current request success.

Observability is therefore relative to a behavior functor and a question. "The system is observable" without specifying the state distinctions and observations is incomplete.

Refinement by new observations

Adding a heap-size metric, allocation stack database, or periodic dump can refine behavioral equivalence. States previously merged by the observation model become distinguishable. This gives us a mathematical perspective on instrumentation requests: we are not merely adding data but are refining the quotient of states induced by observation.

Partially observable dynamical systems can be studied categorically, including functorial measures of information loss (Horstmeyer and Rezagholi 2020). For software, a similar program can quantify which state distinctions a telemetry configuration collapses.

Final behavior and explanations

Final coalgebras, when they exist, provide canonical representations of behavior. They are powerful in formal models, but production systems rarely arrive as a neat endofunctor with a computable final coalgebra. The practical use is local: specify a subsystem, observation step, and equivalence relation, and then test whether the model distinguishes the failure states of interest.

Practical recipe: an observability discrimination test

1. Select state pairs that must be distinguished for a diagnostic goal.
2. Define the current observation behavior F.
3. Run or simulate the observation from each state.
4. Determine whether the observations are equivalent under the production comparator.
5. If they are, propose the smallest additional observation that separates them.
6. Re-test overhead and perturbation as part of the new model.

In this way, we turn "we need more logs" into a targeted separation problem.

Chapter 14 — Traces and logs as 2-categories

> ***Diagnostic question:*** *How can we represent transformations and relations between trace messages, not only sequences of messages?*

One level is sometimes insufficient

For an ordinary category, we have objects and arrows. A 2-category also has 2-morphisms between parallel 1-morphisms, and these 2-morphisms compose vertically and horizontally and obey compatibility laws.

The trace-pattern work first considered a trace or log as a monoid (a one-object category whose morphisms are messages) and later proposed viewing the traced system as an object, messages as 1-morphisms, and relations between messages as 2-morphisms (Vostokov 2026, 380–81; 2024b, 361–62). This supplies a conceptual basis for vertical and horizontal composition and whiskering.

The Windows API source gives a complementary hierarchy: ordinary API usage at one categorical level, diagnostics and debugging at a level of transformations between executions, and meta-diagnostics at a further level (Vostokov 2024a, 264). This is a guiding analogy until the parallel morphisms and composition laws at each level are supplied. Its practical message is sound: evidence about a call path and evidence about the process that interpreted that path belong to different logical levels.

What can a 2-morphism mean?

Depending on the model, a 2-morphism may represent:

- a refinement from a generic event to a concrete event;
- an explanation relating two alternative paths;
- a transformation between message schemas;
- an equivalence or rewrite between trace fragments;
- an annotation, correlation, or causal justification between parallel activities.

The two 1-morphisms must be parallel: they have the same source and target objects in the chosen category. A similarity edge between arbitrary log rows is not automatically a 2-morphism.

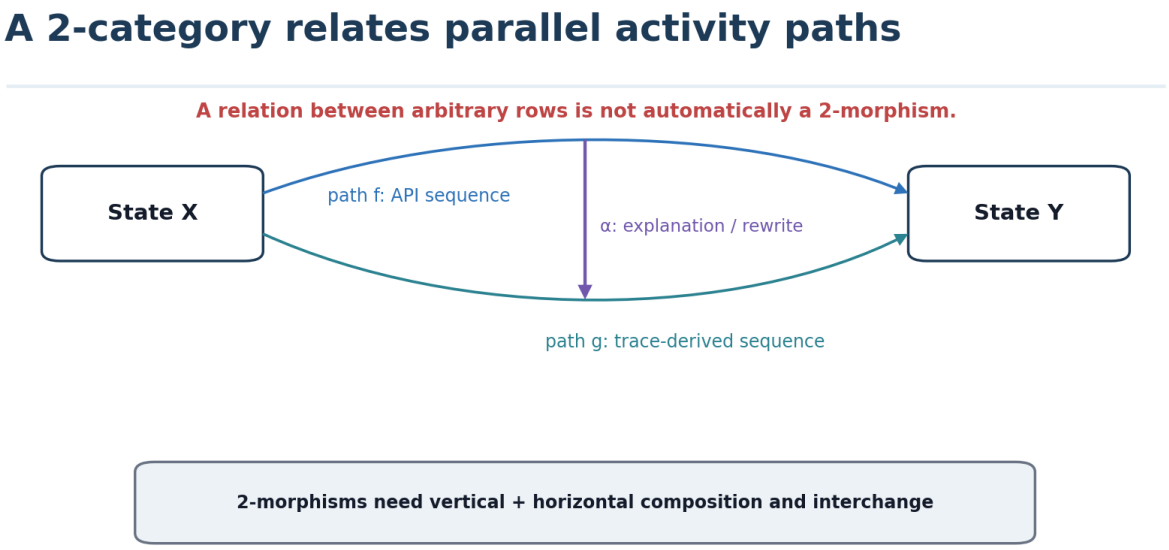


Figure 13. A 2-category adds transformations between parallel activity paths and supports vertical and horizontal composition.

Vertical and horizontal composition

Vertical composition combines successive transformations between the same pair of paths. A raw message is normalized, then semantically classified.

Horizontal composition combines transformations alongside composition of the underlying activities. If a transformation relates two implementations of step A and another relates two implementations of step B, horizontal composition relates the A-then-B paths.

Whiskering composes a 2-morphism with an adjacent 1-morphism on one side. Operationally, we extend a local path equivalence by a shared prefix or suffix.

These operations can support compositional trace comparison, and a local equivalence proof can be embedded in a larger request path instead of recomputed from scratch.

Trace Polynomial as a two-composition proposal

Volume 17 proposes a **Trace Polynomial** in which message concatenation forms monomial-like trace fragments, while a second concatenation assembles fragments; both operations are noncommutative, and no distributive or interchange law is assumed (Vostokov 2025, 17:38–39). For example, a stream may be segmented as

$$ABC + 3(AC) + AC^2D.$$

This is not a polynomial over an ordinary commutative semiring. A precise version begins with two typed associative composition operations (within-fragment and between-fragment) plus a canonical segmentation rule. If no interchange law connects them, their nesting cannot be rearranged freely.

The diagnostic value is motif preservation. A normalizer, sampler, or summarizer may preserve counts while destroying fragment boundaries; another may preserve local order while merging repetitions. A Trace Polynomial adapter should therefore state which of the two compositions it preserves. A discrepancy between raw and reconstructed forms can detect segmentation drift, lost repetitions, or an invalid canonicalization. Until those operations and laws are defined for a concrete trace model, "polynomial" and "2-semigroupal" remain disciplined proposals rather than established properties of all logs.

Software trace versus categorical trace

A traced monoidal category is an established categorical structure for feedback and iteration (Joyal, Street, and Verity 1996). Its word *trace* does not mean an execution log. The concepts may interact (for example, feedback in a process model can be represented by a categorical trace) but we must not infer that every software trace is a traced monoidal category.

This terminological separation prevents one of the easiest category-theory errors in observability writing.

Strictness and weak structure

Real transformations often associate only up to coherent isomorphism rather than literal equality. Bicategories and weak 2-categories model this. The main text uses strict 2-categorical diagrams for clarity. Any implementation that depends on reassociation or schema adapters should state the coherence maps rather than assume strict equality.

Practical recipe: promote only useful relations

1. Define system or state objects.
2. Define parallel activity paths as 1-morphisms.
3. Select one well-typed relation as a 2-morphism.
4. Define vertical and horizontal composition.
5. Test identity, associativity, interchange, and typing laws.
6. Use the model only if composition reduces repeated trace-analysis work.

Chapter 15 — Diagnostic operads and compositional workflows

> ***Diagnostic question:*** *How can multi-input analysis operations be assembled into lawful, reusable workflows?*

Why categories alone may be awkward

For a category arrow, we have one source object and one target object. Diagnostic operations often accept several inputs: a trace, symbol set, deployment manifest, and baseline, and they may also use parameters. Operads are designed to describe operations with multiple inputs and one output and how those operations compose.

Theoretical Software Diagnostics introduced the **Diagnostic Operad** as a sequence of diagnostic operations required to extract indicators in a process described by analysis patterns (Vostokov 2024b, 279–81). The present formulation makes artifact types explicit through a colored operad.

Colors and operations

Colors are types such as:

```
RawLog, NormalizedLog, Trace, MetricSeries,
MemoryDump, Symbols, Config, WorkingSet,
IndicatorSet, HypothesisSet, Report
```

For an operation, we have an input profile and output color:

$$\text{symbolize: } (\text{MemoryDump},\text{Symbols}) \to \text{MemoryDump},$$

$$\text{correlate: } (\text{Trace},\text{NormalizedLog}) \to \text{WorkingSet}.$$

Composition substitutes the output of one operation into a matching input of another, and the operad laws say that substitution is associative and has identity operations. Symmetric operads also permit controlled reordering of inputs.

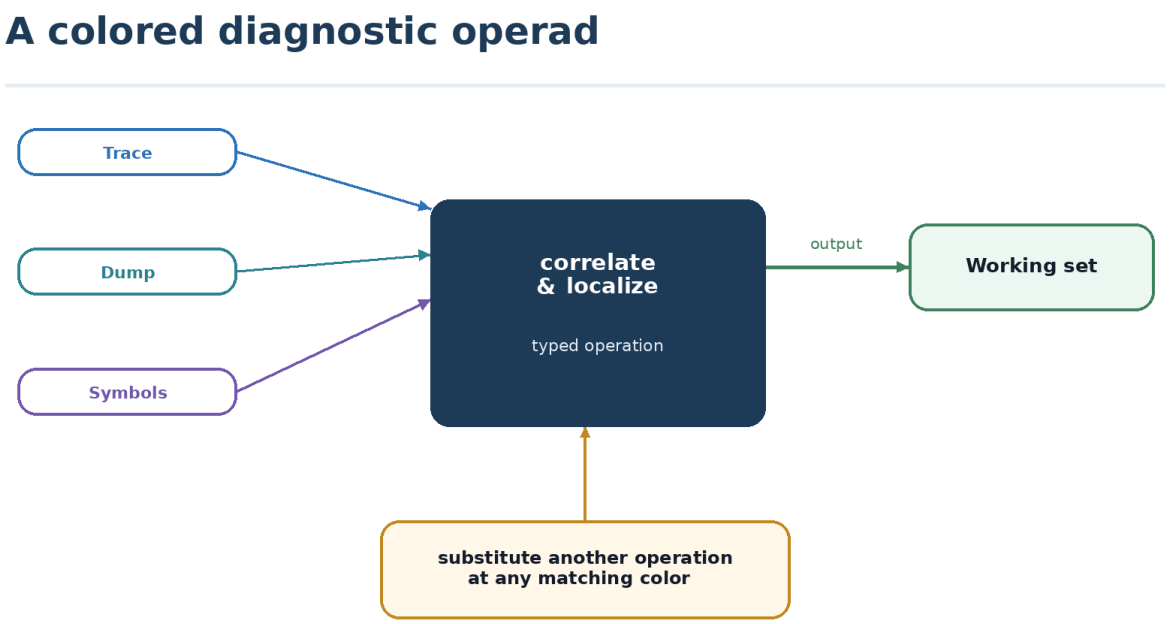


Figure 14. A colored diagnostic operad types multi-input analysis operations and composes them by substitution.

Operad versus pipeline engine

The operad specifies syntax: which operations can compose. An **algebra** of the operad gives semantics or implementation: concrete data types and functions that realize the operations. Different tools can implement the same diagnostic operad.

This separation supports interoperability. One team may implement normalize in a stream processor and another in a notebook; if both satisfy the same algebraic and invariant contracts, workflows can be compared.

Operads have been applied to complex-system design and automated reasoning, where decomposition and reconstitution must remain explicit (Foley et al. 2021). Diagnostic workflows share that need.

The Windows API source further distinguishes many-input/one-output operations from many-input/many-output operations and points to properads for the latter, with monoidal products for independent parallel calls (Vostokov 2024a, 266–67). This extension matters when an analysis both consumes and produces several artifacts (for example, one capture operation produces a dump, a manifest, and a timeline) or when independent subanalyses execute in parallel. The richer structure is justified only when its wiring and composition laws are used; a list of function parameters is not by itself an operad or properad.

Branch, merge, and feedback

Operadic substitution covers tree-shaped multi-input composition. Workflows with sharing, feedback, or state may require PROPs, monoidal categories, traced structures, or other richer formalisms. Do not force every workflow into a plain operad.

A practical representation can still begin with an operadic core:

```
operation: leak_analysis
inputs: [MetricSeries, MemoryDump, Symbols]
output: HypothesisSet
composition:
  - trend: detect_monotone_growth(MetricSeries)
  - heaps: symbolize_heaps(MemoryDump, Symbols)
  - suspects: pullback(trend, heaps, ProcessIdentity)
invariants:
  - build_identity_preserved
  - process_identity_preserved
  - symbol_version_matches_binary
```

Practical recipe: a typed pattern registry

For every analysis pattern, add:

- input and output colors;
- required parameters and external state;
- preservation profile;
- identity operation;
- allowed substitutions;
- equivalence tests for alternative implementations;
- cost, perturbation, and confidence metadata.

Pattern composition then becomes mechanically checkable before data is processed.

Part IV — A Categorical Theory of Observability

Chapter 16 — Signals as a family of observation functors

Diagnostic question: *What does each observability signal preserve, and how can the signals form one system without being collapsed into one format?*

Beyond a bag of signals

Observability platforms commonly organize data into traces, metrics, logs, context, and related signals. OpenTelemetry treats signals as specialized parts of one client architecture that share context propagation while retaining distinct models (OpenTelemetry Authors 2026b). The categorical viewpoint represents that distinction explicitly.

Let $\mathcal{E}$ be an execution category. For each modality i, define an observation construction

$$O_i : \mathcal{E}_i \to \mathcal{A}_i,$$

where $\mathcal{E}_i$ is the domain actually observed and $\mathcal{A}_i$ is the modality's evidence category. Typical modalities include:

- L for logs and structured events;
- T for traces and spans;
- M for metrics and exemplars;
- D for memory or state dumps;
- P for profiles;
- N for native kernel, network, or security events.

We do not require the codomains to be identical, and their difference is valuable.

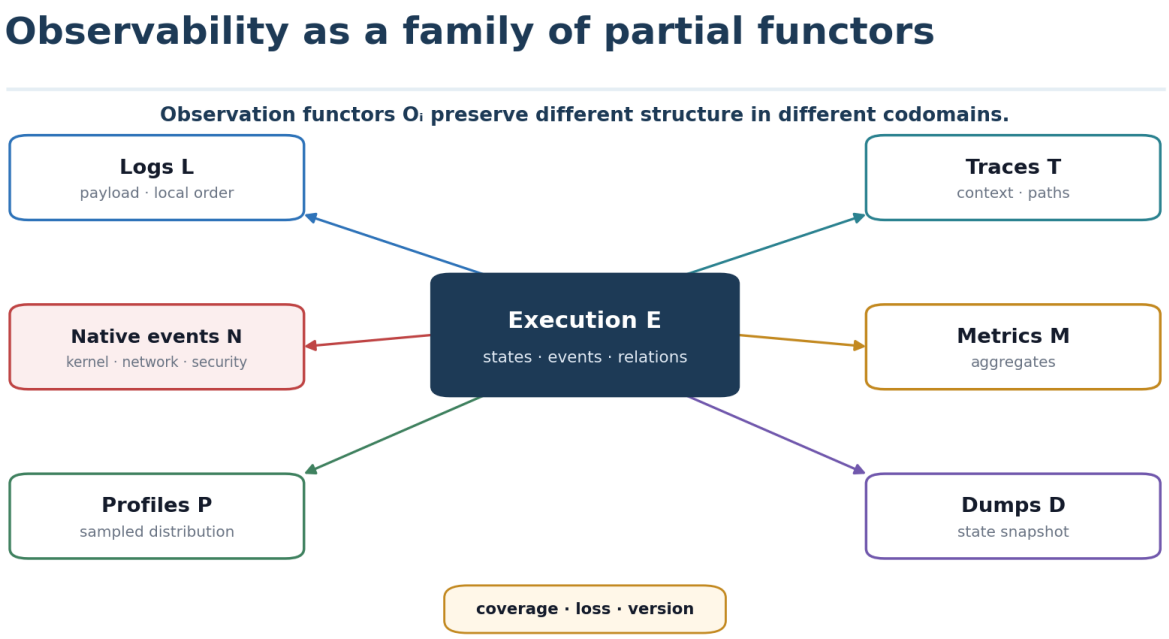


Figure 15. Observability is a family of partial, loss-bearing observation functors rather than one undifferentiated stream.

Observation Spaces

Volume 17 catalogs a plurality of **Observation Spaces**: memory, traces and logs, metrics, diagnostic indicators, narratives, presentation, software internals, code, state, data, hardware, and token spaces (Vostokov 2025, 17:69–71). Categorically, we treat each space as a candidate category or family of categories, not one universal coordinate system. It needs its own objects, arrows, comparison rules, and visibility limits.

The important new step is to connect spaces without flattening them. For example, a presentation view may be a functor from a metric category into a visual category. Likewise, a symbolizer may relate a memory-address category to a source-location category, while a trace-field functor may map messages into latency values. Cross-space comparison then uses natural transformations, pullbacks, spans, or a common semantic codomain according to the type of relation.

Calling an analysis pattern a "metric" on an Observation Space is mathematically literal only if it supplies a distance satisfying the required metric laws. Otherwise it is a diagnostic comparator or pattern detector. The wider spatial vocabulary remains useful, but the formal label should track the actual structure.

The Diagnostics–Observability Square

Volume 17 separates **observability** and **diagnosability** as properties from **observation** and **diagnostics** as processes, arranging them in a Diagnostics–Observability Square (Vostokov 2025, 17:72–74). This distinction resolves a common ambiguity: a team may perform observation even when the system is poorly observable, and may perform diagnostics even when the available evidence poorly distinguishes faults.

The square can be turned into a categorical engineering contract in a category of specifications and implementations. Let:

- **Observability specification** — state distinctions that evidence should reveal;
- **Diagnosability specification** — fault or pattern distinctions that analysis should decide;
- **Observation implementation** — the implemented observation process;
- **Diagnostics implementation** — the implemented diagnostic process.

We have an enablement arrow e from the observability specification to the diagnosability specification; realization arrows r_O and r_D from each specification to its implementation; and a feed arrow f from the observation implementation to the diagnostics implementation. The contract is

$$r_D \circ e \approx f \circ r_O.$$

One route derives and realizes diagnostic requirements from the desired observability, and the other realizes observation and feeds its products into the diagnostic implementation. Noncommutation reveals a design–operation gap: the telemetry may not support a promised diagnosis, the diagnostic procedure may depend on undeclared observations, or the property specifications may be wrong.

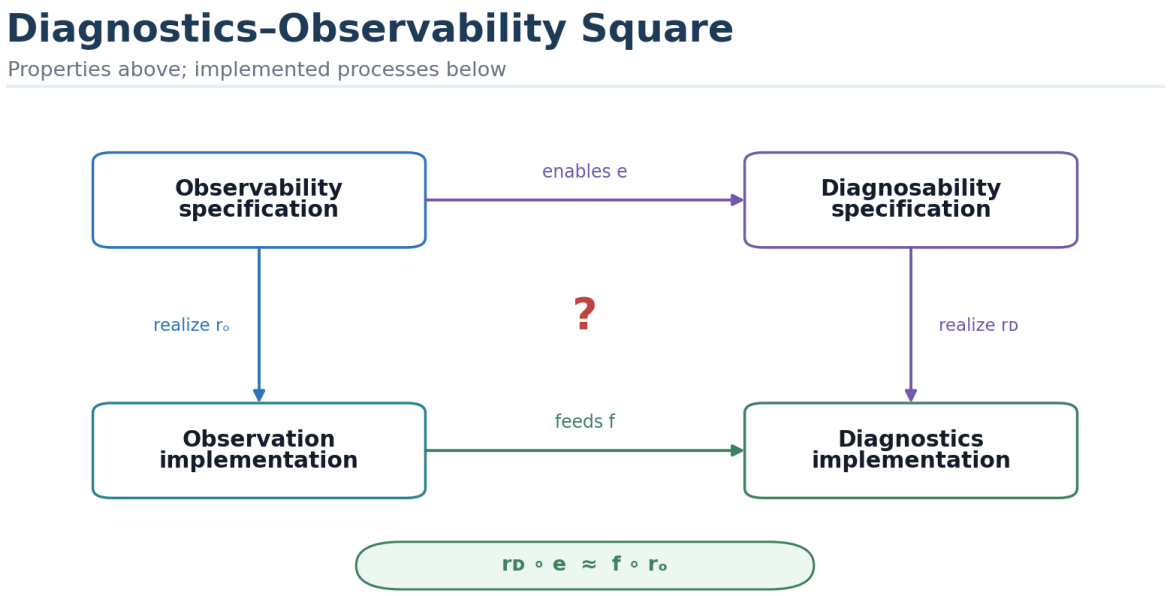


Figure 15A. The Diagnostics–Observability Square becomes a testable contract between desired properties and implemented processes.

The feedback arrows proposed in the source (diagnostics requesting more observations and diagnostic experience improving observability) form an iterative design loop around the square. They should be modeled separately from the one-way commutativity equation so that a feedback cycle is not mistaken for an inverse or adjunction.

Modality contracts

Each O_i needs a contract:

Coverage: Which execution objects and arrows lie in $\mathcal{E}_i$?

Preservation: Which identities, orders, multiplicities, attributes, and compositions survive?

Loss: Which distinctions are intentionally merged or sampled?

Perturbation: How can observation change execution?

Semantics: Which schema and convention define the meaning of the result?

Failure behavior: How are exporter loss, backpressure, clock uncertainty, and missing context represented?

This transforms a signal taxonomy into a diagnostic model.

Logs

Logs often preserve human-oriented semantics and local payload. Their order may be only per process or per sink, and textual adjacency need not be causal adjacency. Templates induce equivalence classes of messages, while parameters distinguish instances. A log observation functor should state whether it preserves occurrence identity, local order, causal parenthood, or only a sequence of writes.

Traces

Distributed traces represent requests as spans connected by parent-child or link relations. OpenTelemetry describes a trace as a directed acyclic graph of spans (OpenTelemetry Authors 2026b). The source execution may contain cycles through retries or message loops, while one individual trace DAG unfolds particular occurrences. The observation is therefore an execution instance representation, not necessarily the architecture graph.

Metrics

Metrics map many events to aggregate values over attributes and time. Algebraically, counters often use monoidal aggregation. This aggregation can discard event-level distinctions; non-faithfulness is the more specific claim that distinct parallel arrows are merged by the chosen functor. Exemplars link aggregates to selected traces and supply some instance identity. They do not invert the aggregation or recover every contributing event.

Dumps and profiles

A memory dump is rich in state but narrow in time. A profile is rich in distributional behavior but sampled and often weak in individual identity. Treating either as a "more detailed log" obscures its preservation profile.

The earlier **Adjoint Space** and **Trace Presheaf** patterns connect traces to memory state (Vostokov 2024b, 223–25; 2026, 45–47, 330). Categorically, we need explicit association maps and scope.

Trace Fields

The **Trace Field** pattern maps each trace message to another domain and also proposes a functor between categories (Vostokov 2026, 314). Every column is a set function $F: \mathcal{T} \to \mathcal{V}$. It is automatically functorial when $\mathcal{T}$ is discrete, because only identities must be preserved.

If $\mathcal{T}$ contains time-order arrows and $\mathcal{V}$ is ordered, the field is functorial only when monotone. Most diagnostic fields rise and fall. We therefore keep $\mathcal{T}$ discrete or declare target transitions that the field actually preserves. "Functor" remains a claim to test.

A family of fields $F_j: \mathcal{T} \to \mathcal{V}_j$ supports multiphysics-style analysis of the same message sequence. Couplings between fields are additional transformations or relations; they should not be inferred solely because values change together.

Practical recipe: signal inventory as mathematics

We create one row per modality with domain, codomain, object identity, arrow meaning, composition, preservation, loss, and version. Then we add only those cross-signal adapters whose semantics have tests, and we obtain an observability architecture that can be reasoned about compositionally.

Chapter 17 — Causality, context propagation, and distributed traces

> ***Diagnostic question:*** *What must context propagation preserve for a distributed trace to support causal reasoning?*

Time order is not causal order

Wall-clock timestamps provide an order after normalization, but clock order alone does not establish causality. Lamport's happens-before relation is generated by local program order, message send-to-receive edges, and transitivity (Lamport 1978). It is a partial order: concurrent events may be incomparable.

A happens-before preorder becomes a category with events as objects and at most one arrow $a \to b$ when a happens before b. A distributed tracing system attempts to observe part of this structure through propagated context and span relations.

Context propagation as structure preservation

W3C Trace Context standardizes headers that identify requests and propagate vendor-specific context across services (W3C Distributed Tracing Working Group 2021). OpenTelemetry uses context propagation to correlate signals across process and network boundaries (OpenTelemetry Authors 2026b).

In categorical terms, a propagation adapter maps a communication edge in the execution category to a relation in the trace category. The key invariant is not merely "the identifier was copied." It is that composition across service boundaries preserves the intended parent, link, or causal relationship.

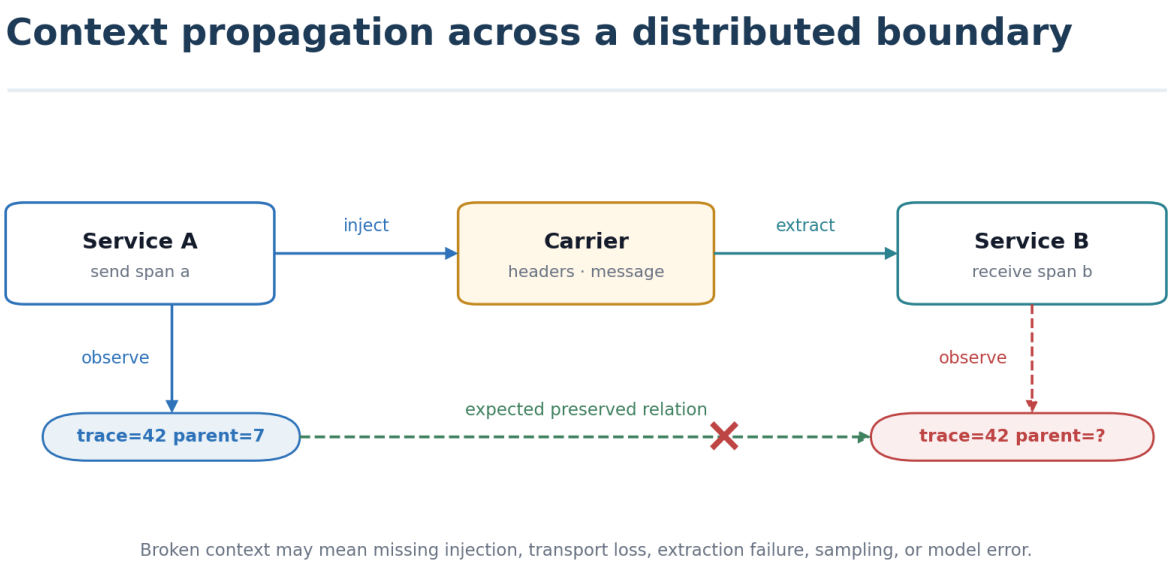


Figure 16. Context propagation maps cross-process communication into trace relations; a broken edge creates a visibility gap.

The propagation square

Let s be a send, r the matching receive, E the execution mapping, and T the trace mapping. A simplified invariant compares:

1. compose the execution edge across the boundary and then observe it;
2. observe each side and compose through the propagated trace context.

If these routes disagree, candidate causes include header removal, incompatible propagators, asynchronous work detached from context, message fan-out represented with the wrong relationship, or reuse of an identifier.

Tracing systems as prior work

X-Trace proposed pervasive cross-layer tracing through propagated task metadata (Fonseca et al. 2007). Dapper demonstrated a large-scale, low-overhead distributed tracing infrastructure and its evolution into a monitoring platform (Sigelman et al. 2010). Pivot Tracing combined dynamic instrumentation with a happened-before join for cross-tier causal monitoring (Mace, Roelke, and Fonseca 2015). Request-flow comparison has been used to diagnose performance changes between executions (Sambasivan et al. 2011).

The categorical framework does not replace these techniques. It supplies a common language for their preserved relations, transformations, and comparison laws.

Fan-out, fan-in, and links

Parent-child trees are insufficient for queues, batching, retries, and shared work. A message may contribute to several consumers, and several messages may contribute to one batch. Representing every relation as parent-child invents structure.

Use a category or higher structure whose arrows reflect the actual semantics. Trace links, correlation sets, or hypergraph-like relations may be necessary. Structured-cospan approaches to open networks demonstrate how interfaces and composition can be made explicit (Baez and Courser 2020).

Absence and silent messages

An expected receive with no observed span may mean no receive occurred, the event was uninstrumented, context was lost, sampling removed it, or export failed. Absence is not one value. The trace-pattern idea of Silent Messages provides a way to represent meaningful positions where no ordinary message is visible (Vostokov 2026, 272–73). A categorical model can add typed absence objects or partial arrows rather than treat every gap as the same null.

The boundary sentinels ⊥ and ⊤ in Chapter 9's Galois Trace are another typed absence: no earlier or no later partner exists under the declared alignment. They are not ordinary messages, and they are not the Silent Messages used to mark minimal-resolution gaps. Keeping the types separate prevents an alignment boundary from being read as an observed event.

Practical recipe: boundary invariants

For every protocol boundary:

1. declare the execution communication relation;
2. declare the propagated context fields and version;
3. declare how fan-out, fan-in, retries, and cancellation map to span relations;
4. inject test requests with known topology;
5. compare observed and expected composites;
6. preserve unmatched sends and receives as first-class residuals.

Chapter 18 — Visibility limits, information loss, and observability drift

> ***Diagnostic question:*** *How can we tell whether a change in telemetry reflects a change in the system or a change in the observation mapping?*

Observability is a relation, not a substance

A system is observable only relative to state distinctions, admissible observations, and a diagnostic goal. In control theory, observability concerns recovery of internal state from outputs. In software practice, the term is broader, but the relational structure remains: an observer maps system behavior into evidence.

The trace-pattern **Visibility Limit** names the boundary beyond which data is unavailable or not visible (Vostokov 2026, 360). A categorical model refines the boundary into a loss profile of the observation functor.

Non-faithfulness and ambiguity

If distinct execution arrows map to the same evidence arrow, the observation is non-faithful on those arrows. The resulting ambiguity can be measured by equivalence classes of executions that observations cannot distinguish.

This gives us an **observability kernel** in an informal but useful sense: the distinctions collapsed by the map. In concrete algebraic categories, kernels have precise definitions; in general categories, use the appropriate kernel pair or equivalence relation rather than assuming an ordinary kernel exists.

Versioned observation squares

Let E_0 and E_1 be execution models before and after a software change, and A_0 and A_1 evidence models before and after an instrumentation or schema change. Let $U: E_0 \to E_1$ represent the system evolution and $V: A_0 \to A_1$ the evidence migration. Observation functors O_0 and O_1 form a square:

$$V \circ O_0 \overset{?}{\cong} O_1 \circ U.$$

If the square commutes up to the declared equivalence, evidence changed exactly as expected from the two migrations. If it does not, we have **observability drift**.

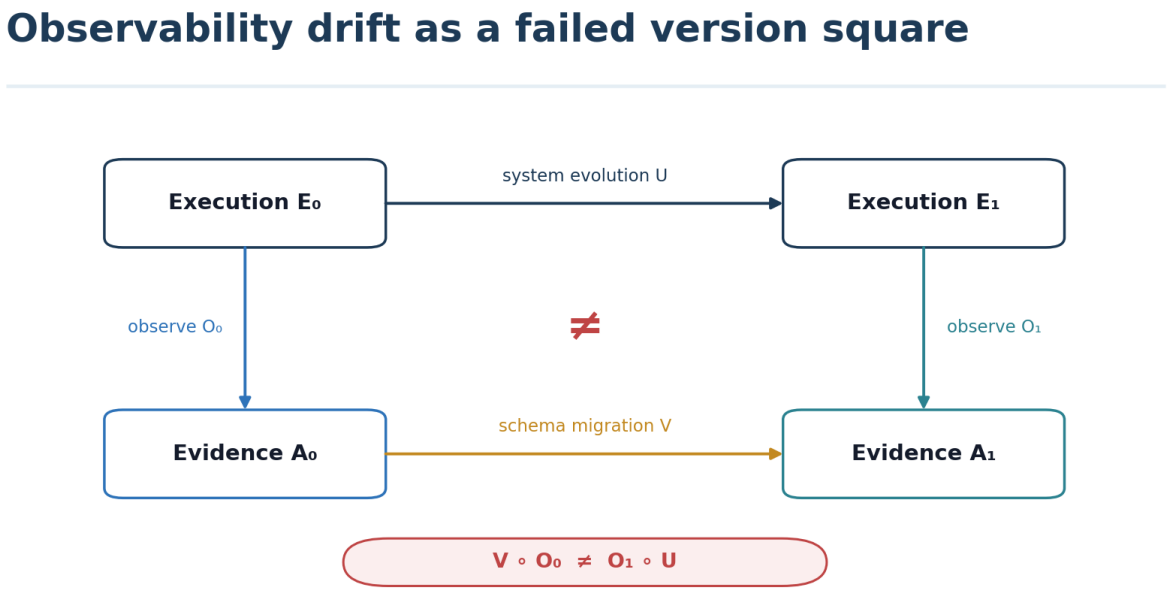


Figure 17. Observability drift is failed compatibility between system evolution and evidence evolution.

Sources of drift

- instrumentation points moved across semantic boundaries;
- attribute names remained but meanings changed;
- aggregation windows or temporality changed;
- sampling policy changed;
- context propagation coverage changed;
- a normalizer uses an old semantic convention;
- system behavior changed in an unmodeled way.

The square localizes the ambiguity to four mappings instead of attributing every dashboard change to the application.

Semantic conventions as contracts

OpenTelemetry semantic conventions define common names and meanings across signals (OpenTelemetry Authors 2026a). A convention migration should be treated as a schema functor with tests for preservation of resource identity, operation semantics, status, and units. Renaming a field is the easy part; preserving the arrows among entities is the real contract.

Counterfactual instrumentation

Before adding telemetry, we ask which currently indistinguishable execution states it will separate. Before removing telemetry, we ask which distinctions it will collapse. Together, these questions give us a counterfactual analysis of the observation functor.

A high-volume event that never changes an equivalence class relevant to any diagnostic goal may be redundant. A low-volume correlation edge that separates otherwise identical failure paths may be extremely valuable.

Diagnostic sufficiency for a declared question

Consider two requests with the same retained trace summary. One completed normally, while the other failed after a retry. A message count alone may not distinguish them. Repeating the analysis cannot recover a distinction absent from that summary.

Before deciding which artifact fields to retain, we ask whether two executions can have the same observation but require different diagnostic answers. If they can, the observation is insufficient for that question. Otherwise, we can determine the required answer from it even though other execution details have been discarded.

For example, Chapter 31 includes three records with zero response bytes, in two HTTP status families. A byte-only observation cannot determine the status family. Retaining the status family is sufficient for that classification, but it still does not identify why a request received that status. Diagnostic sufficiency depends on the declared question and on the execution cases admitted by the model.

For a finite collection of modeled cases, we group the cases by their retained observation and compare the required answers within each group. Different answers give a counterexample and a reason to add an observation that separates those cases. Agreement in the test collection alone does not establish sufficiency outside it.

Please note that we need not require an injective observation mapping or a faithful functor. We may discard distinctions that do not affect the diagnostic answer. Appendix B.21 gives the factorization condition. Here diagnostic sufficiency is deterministic and relative to a declared goal; it is not statistical sufficiency.

Sound approximation of diagnostic states

Abstract interpretation provides an established approach to reasoning with approximations of program behavior (Cousot and Cousot 1977). In diagnostics, a sound overapproximation may similarly retain a set containing all states consistent with the evidence. If that set contains no fault state of the specified kind, the fault is excluded under the model. If it contains a fault state, the approximation does not exclude the fault; it does not establish that it occurred. A proposed abstraction must justify its soundness condition. Giving an artifact summary a categorical name does not supply that justification. Appendix B.23 gives a finite interval example.

Practical recipe: drift gates in CI

1. Version the execution schema, telemetry schema, and adapters.
2. Maintain golden execution scenarios.
3. Produce evidence under old and new versions.
4. Migrate both into a common comparison category.
5. Calculate commutativity residuals.
6. Block or review changes that exceed declared tolerances.

Chapter 19 — Trace viewpoints, ontologies, and world models

> ***Diagnostic question:*** *How can specialists maintain different views of the same incident while still referring to one shared diagnostic world?*

Viewpoints are structured partial views

A support engineer, developer, security engineer, SRE, architect, database engineer, and ML engineer attend to different objects and arrows. The trace reference formalizes this practical plurality with Trace Viewpoints (Vostokov 2026, 344–45). A viewpoint is not merely a filter; it may supply a different vocabulary, granularity, and relevance relation.

We can model each viewpoint as a functor

$$V_i : \mathcal{W} \to \mathcal{P}_i,$$

from a shared world category $\mathcal{W}$ to a perspective category $\mathcal{P}_i$. In practice, no team possesses the full world category. $\mathcal{W}$ is a maintained diagnostic model assembled from evidence, architecture, and validated relations.

Shared ontology versus shared labels

A shared ontology declares entity types and arrows: a process *runs in* a container; a span *represents* an operation; an allocation *belongs to* a heap; a deployment *installs* a build. Ologs provide a categorical framework in which meaningful noun phrases are objects and meaningful verb phrases are arrows (Spivak and Kent 2012).

Sharing the string service.name does not guarantee shared ontology. One source may mean logical workload, another deployment unit, and another billing label. The arrows around the object reveal the mismatch.

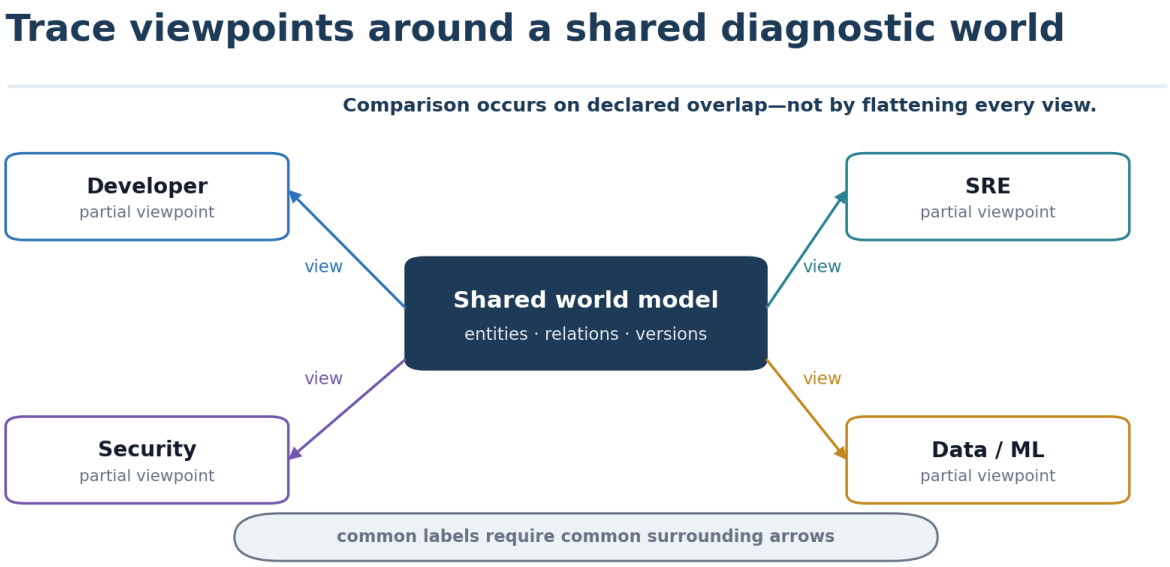


Figure 18. Specialist viewpoints map a shared diagnostic world into purpose-specific categories while comparison adapters preserve common relations.

Multiple traces and multiple discourses

An incident may have several traces of the same underlying activity and several discourses about them. A runtime trace, audit trace, network trace, and agent transcript are different observation modalities. A developer explanation, security interpretation, and customer narrative are different discourses. Category theory can keep these axes separate:

- observation functors map the world to artifacts;
- interpretation functors map artifacts to specialist hypothesis categories;
- natural transformations compare systematic views;
- a shared ontology provides common mediating objects.

Here we extend the software-narratological distinction among story, plot, and presentation developed in the source work (Vostokov 2024b, 121).

Trace multiphysics as a field family

Several Trace Fields can be applied to the same message category: latency, CPU, memory, disk, information content, uncertainty, control state, and semantic outcome. Couplings may be same-message maps

$$C_{ij}(m) = g\left(F_i(m), F_j(m), R_{ij}(m)\right)$$

Here $R_{ij}(m)$ denotes explicitly declared shared-context or interaction data for fields i and j at message m; it is not inferred from co-variation alone. A coupling may instead use delayed relations between messages m_a and m_b. This gives us a structured way to study cascades such as latency → retries → context growth → uncertainty → outcome degradation.

The field functors are formal when their domains, codomains, and arrow mappings are specified. The word *multiphysics* remains a guiding analogy unless the coupling laws and conserved quantities are supplied.

Pattern Category Theory as diagnostic knowledge engineering

Volume 17 proposes **Pattern Category Theory** (PCT): patterns as objects and pattern relations, transformations, applications, inheritance, and composition as morphisms in a "living category of meaning" (Vostokov 2025, 17:96–97). The source presents PCT as exploratory and explicitly cautions that some generated details may be hallucinated. The strongest version for diagnostics therefore separates several structures instead of placing every pattern-related thing in one category.

Let $\mathcal{P}$ be a category of **versioned pattern specifications**. Objects contain intent, context, evidence schema, invariant, confounders, and consequences. A morphism is a declared refinement or translation that preserves the meaning required by its target. Identities are unchanged specifications; composition chains refinements. "Is similar to" is not a morphism unless its direction and preservation law are defined.

Let $\mathcal{A}$ be a category of artifacts and transformations, and $\mathcal{I}$ a category of pattern instances and evidential refinements. A detector implementation may be a functor on a supported artifact subcategory,

$$D: \mathcal{A}_{supported} \to \mathcal{I},$$

while pattern composition is represented by the diagnostic operad of Chapter 15. Natural transformations can compare detector versions; a schema-migration square can test whether instances remain semantically compatible; a provenance functor maps each instance to its evidence path.

This natural-transformation layer has explicit source provenance. The earlier Categorical Foundations calls cross-platform transformations between analysis-pattern functors **General Analysis Patterns** and calls the generalizing arrows between problem patterns **General Problem Patterns** (Vostokov 2024b, 271–72). We retain the proposal and supply the typing that makes it testable.

$$U_W: C \to C_W, \quad U_L: C \to C_L, \quad D_W: C_W \to P, \quad D_L: C_L \to P, \qquad \eta: D_W \circ U_W \Rightarrow D_L \circ U_L.$$

The comparison functors U_W and U_L interpret one common abstract artifact category C in the two platform categories; both detectors then land in the same semantic pattern category P. Each component η_c maps a Windows pattern instance to its declared Linux counterpart, and the naturality square tests whether artifact transformation and pattern translation commute. Without the common C, the common P, or an explicit comparison functor, cross-platform similarity is not a natural transformation. General Problem Patterns are typed refinements in P or in the specification category above; the word "general" does not create a 2-cell by itself.

This typed separation gives "pattern metabolism" an operational interpretation: creation, refinement, composition, detection, rejection, and retirement are morphisms or functorial transformations only in categories where their laws are declared. It also prevents three common conflations:

- a pattern specification is not a detected instance;
- a detector is not the pattern it implements;
- a narrative similarity is not automatically a semantics-preserving refinement.

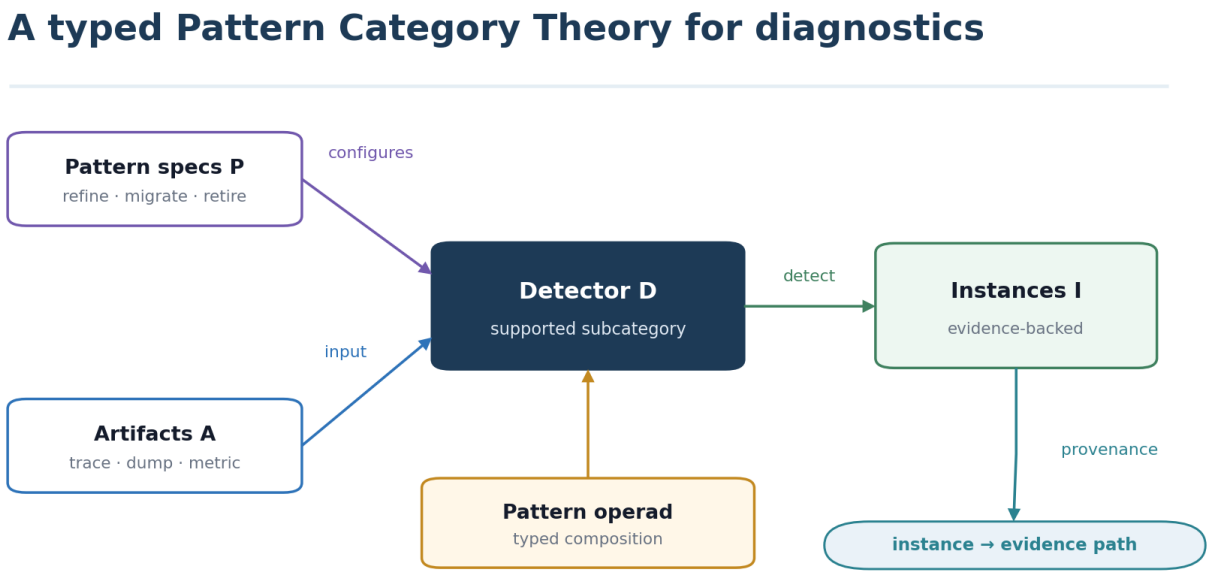


Figure 18A. Pattern Category Theory becomes testable when specifications, artifact transformations, instances, detectors, and provenance occupy explicitly related structures.

Our Categorical Invariant Violation pattern is one object in such a specification category. In Appendix C, we supply instances of the common card schema, and Chapter 28 shows how their diagram laws can compile into checks.

World-model governance

Every ontology mapping should have:

- an owner and version;
- source and target schemas;
- identity and equivalence policy;
- mapping tests;
- known non-preserved distinctions;
- examples from each specialist viewpoint;
- deprecation rules.

We do not treat the world model as a static truth store: it is a versioned categorical interface among evidence and interpretations.

Chapter 20 — Instrumentation design through categorical contracts

> ***Diagnostic question:*** *How can instrumentation be designed from future diagnostic questions rather than from available logging APIs?*

Start with a diagram

An instrumentation requirement should begin with an expected diagram and a possible failure, not with "add more fields." For example:

> *For every accepted order, the business outcome derived from the terminal span and the outcome derived from the audit event must agree after normalization.*

This statement identifies two observation paths, a common codomain, and an equality test. It implies concrete fields only after the mappings are designed.

Categorical telemetry contract

A contract contains:

Question: What distinction or invariant matters?

Execution model: Objects, arrows, and granularity.

Observation domains: Which subcategories are instrumented?

Signal functors: Object and arrow mappings.

Common comparison category: Normalized entities and relations.

Expected diagrams: Naturality, functoriality, gluing, or universal-property tests.

Loss and perturbation: Sampling, aggregation, overhead, and observer effect.

Versioning: Schema, semantic convention, and code compatibility.

Test fixtures: Executions that exercise identities, composition, concurrency, retries, failure, and cancellation.

Instrumentation and the observer effect

Adding debug logging can alter timing, contention, memory layout, or failure probability. The observation mapping is not always external to the system: it may change the source category itself.

Represent instrumentation configuration as an intervention $J: E \to E_J$ and observe $O_J: E_J \to A$. Comparing O and $O_J \circ J$ requires accounting for the intervention. A mismatch may be perturbation rather than better visibility.

Boundary-first design

The highest-value instrumentation often lies on compositional boundaries:

- send/receive;
- enqueue/dequeue;
- acquire/release;
- allocate/free;
- start/finish;
- parse/serialize;
- model/tool request and result;
- deployment desired/actual state.

At each boundary, record identity, relationship, semantic type, version, and outcome. These fields create the intermediate objects through which diagnostic arrows compose.

Trace Plans and trace context

Volume 17 introduces **Trace Plan** as the requirement side of trace engineering and **Trace Context** as the issue, acquisition, system, prior-artifact, and prior-analysis information needed to interpret a trace (Vostokov 2025, 17:40–42). Categorically, we use a Trace Plan to specify a desired observation construction O_{plan} and its preservation profile. The realized instrumentation O_{real} is compared through versioned fixtures and, where well typed, a natural transformation.

A plan should declare:

- the execution objects and arrows that must be observable;
- mandatory identities, context edges, and semantic attributes;
- sampling, retention, resolution, and export behavior;
- the diagrams that downstream diagnostics will test;
- the context record needed to interpret every capture.

The context record is not administrative decoration. It scopes the categories: build, host, clock model, configuration, acquisition method, schema version, missingness, and known observer effects. Without it, two identical trace rows may denote different objects.

Precision and recall of an observation plan

Volume 17 defines **Trace Precision** as the fraction of selected messages used in successful diagnostics and **Trace Recall** as the fraction of diagnostically relevant messages that were selected, including the possibility that relevant messages were never captured (Vostokov 2025, 17:36–37). For selected messages S and relevant messages R in a declared universe U:

$$\text{precision} = \frac{|S \cap R|}{|S|}, \qquad \text{recall} = \frac{|S \cap R|}{|R|}.$$

Precision and recall exchange the denominators S and R, but this numerical symmetry does not by itself make them categorical duals. A duality would require categories and an arrow-reversing correspondence that converts the complete construction and its laws. Operationally, we treat them as complementary adequacy measures for a selection or observation functor.

Recall is especially subtle. If R includes uncaptured messages, its denominator cannot be read directly from the artifact being evaluated. A controlled execution, independent signal, model, or capture with higher coverage must estimate it. Category theory helps expose this visibility boundary; it does not manufacture the missing population.

Test generation from diagrams

A commutative diagram can generate a test skeleton:

```
given: execution fixture and instrumentation versions
collect: all paths in the expected diagram
normalize: each path into the common category
compare: parallel composites
report: residual, missing arrow, and smallest failed subdiagram
```

We can see this as the beginning of diagrammatic observability testing. It complements unit, integration, and end-to-end tests by testing the *relationship between execution and evidence*.

The diagnostic sufficiency check in Chapter 18 supplies a complementary test before instrumentation is removed. For the admitted execution cases, no two observations that remain equal should require different answers to the declared diagnostic question (Appendix B.21).

Practical recipe: the minimum separating instrumentation

1. List the failure states that operations must distinguish.
2. Identify which pairs current signals merge.
3. Propose one new object, arrow, or attribute that separates each critical pair.
4. Add cross-view invariants for the new evidence.
5. Measure perturbation and cost.
6. Remove fields that do not support a declared distinction, relation, or invariant.

Part V — Diagrammatic Debugging in Practice

Chapter 21 — The Categorical Invariant Violation pattern

> ***Diagnostic question:*** *How can a failed structural law be captured as a reusable diagnostic pattern rather than a one-off discrepancy?*

Intent

We call this reusable diagnostic pattern **Categorical Invariant Violation (CIV)**. It detects and localizes a mismatch between an intended categorical model of system observation and the behavior realized by a system, telemetry pipeline, analysis transformation, or interpretation adapter.

Context

We use the pattern when:

- the same fact or relation is obtainable through two or more routes;
- instrumentation is expected to preserve composition or identity;
- two systematic views are related by a natural transformation;
- local evidence should glue into a global account;
- a workflow implementation should realize typed compositional laws;
- an aggregate or schema migration has a stated universal property.

Problem

Evidence is often checked value by value. Structural failures can remain latent because every local value appears plausible while the relationships among values are inconsistent. A trace may contain valid spans with an invalid parent relation, and logs and metrics may each be internally reasonable yet disagree on request outcome. Two filter sequences may return different working sets despite being advertised as equivalent.

Solution

We specify the categorical structure and test its law:

1. Declare categories, objects, arrows, and scope.
2. Declare the expected diagram or algebraic law.
3. Define equality, equivalence, preorder, or distance in the comparison codomain.
4. Evaluate all relevant parallel composites.
5. Calculate an exact mismatch or residual.
6. Decompose a failed diagram into smaller subdiagrams.
7. Distinguish system, instrumentation, transport, schema, analysis, and model causes.
8. Preserve the failed diagram as a diagnostic artifact.

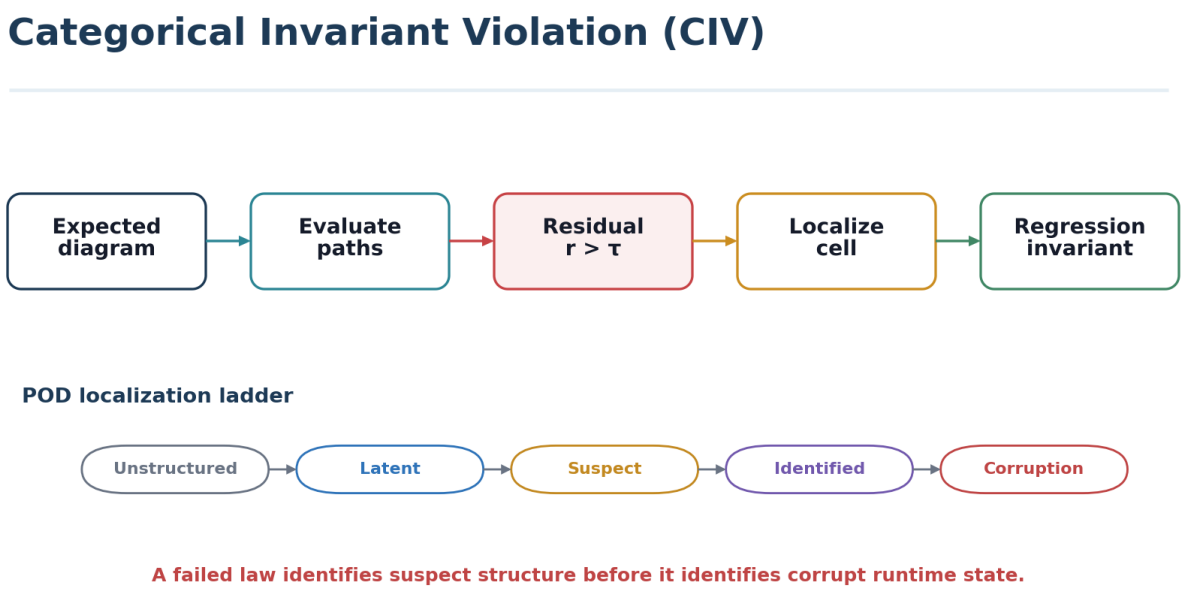


Figure 19. The Categorical Invariant Violation pattern converts an expected structural law into a residual and a localization path.

Types of violation

Typing or identity violation: An event, relation, or no-op does not map to the declared object, arrow, or identity. For example, a retry result is attached to the original attempt object.

Composition or functoriality violation: Observing a composite differs from composing the observations. For example, the end-to-end status differs from the status composed from verified suboperations.

Naturality violation: A systematic conversion between views depends on the route taken. For example, runtime-to-log-to-ontology disagrees with runtime-to-trace-to-ontology.

Gluing violation: Locally valid sections disagree on overlaps or fail to produce a global account. For example, adjacent services disagree on request identity or semantic operation.

Universal-property violation: A purported join, merge, or summary does not satisfy the claimed factorization or uniqueness. For example, a "canonical" normalized representation depends on pipeline order.

Effect-law violation: An implementation explicitly modeled by a monad, comonad, operad algebra, or other structure fails its laws. For example, composing diagnostic operations changes result solely because of reassociation, contrary to the declared model.

The Pattern-Oriented Diagnostics (POD) localization ladder

The pattern supports a progressive diagnostic ladder:

Unstructured: A symptom exists, but no structural model relates the evidence.

Latent: For a large diagram or cross-view contract, we have a residual, but the failed region is unknown.

Suspect: Decomposition isolates a small set of objects, arrows, or adapters.

Identified: A particular mapping, law, version, or boundary is shown to produce the mismatch.

Chapter 28 gives a request-identifier renaming test and a deliberately faulty status classifier. Their counterexamples illustrate how to retain both comparison routes before assigning the mismatch to a particular component.

Corruption: The responsible state or evidence is demonstrated to violate the intended system contract, rather than merely the observation model.

Movement through the ladder requires evidence. A noncommuting diagram establishes *suspect structure*; it does not automatically establish corrupt runtime state.

Consequences

Benefits include precise cross-tool contracts, localized mismatches, reusable tests, and preservation of diagnostic provenance. Costs include modeling effort, version governance, and the risk of dignifying weak analogies with formal language.

Known uses

Candidate uses include context-propagation tests, log/trace status agreement, exporter equivalence, online/offline aggregation comparison, symbol/binary version compatibility, desired/actual deployment state, audit/runtime consistency, and agent tool-call accountability.

Pattern card

```
Name: Categorical Invariant Violation
Arena/category:
Objects:
Morphisms:
Observation and normalization functors:
Expected diagram or law:
Comparator and tolerance:
Preserved distinctions:
Known information loss:
Smallest localization units:
Alternative explanations:
Disconfirming evidence:
Owner and version:
```

Chapter 22 — A diagrammatic debugging protocol

> ***Diagnostic question:*** *What sequence turns an incident from a pile of artifacts into a set of falsifiable structural checks?*

Step 1: State the user-visible failure

Begin with a scoped outcome, not an internal label. “Checkout request r returned 504 between 10:02:14 and 10:02:45” is better than “database issue.” Record the time basis, identity, environment, and what would count as normal.

Step 2: Build the minimal execution graph

We represent only the objects and arrows needed for the question, and we use a free category on this graph unless justified path equations are already known. We mark concurrency and alternative branches rather than forcing a total order.

Step 3: Add observation functors

For every artifact, we draw what it observes and what it loses. We do not draw an arrow for a correlation that has not been established. We mark sampling, aggregation, and unknown regions.

Step 4: Add normalization and interpretation

We map heterogeneous evidence into common categories only through versioned adapters, and we separate raw observation from diagnostic inference.

Step 5: Generate expected diagrams

Create squares or higher diagrams from:

- boundary contracts;
- duplicated representations;
- online/offline implementations;
- old/new versions;
- local/global consistency;
- pattern-pipeline equivalence.

Step 6: Evaluate residuals

We preserve both routes, not only the difference. A residual without its composites is hard to audit. We classify missing arrows separately from unequal arrows.

Step 7: Localize

We decompose the largest failed diagram. Binary decomposition works well for long composites: compare prefixes and suffixes, then recurse on the failing half. For sheaf-like models, rank overlaps by consistency residual.

Step 8: Test alternative models

A failed diagram may invalidate the assumed mapping. We build at least one rival explanation: clock uncertainty, sampling, reused identifier, asynchronous completion, schema drift, or incorrect granularity. We seek evidence that differentiates the models.

Step 9: Identify intervention and consequence

When possible, we change one arrow: restore context propagation, correct a normalizer, add a missing release, or revert a schema. We predict which diagrams should begin to commute. Then we test the prediction.

Step 10: Capture a reusable invariant

We convert the incident-specific diagram into a versioned contract and regression test, and we add it to the diagnostic operad or pattern registry.

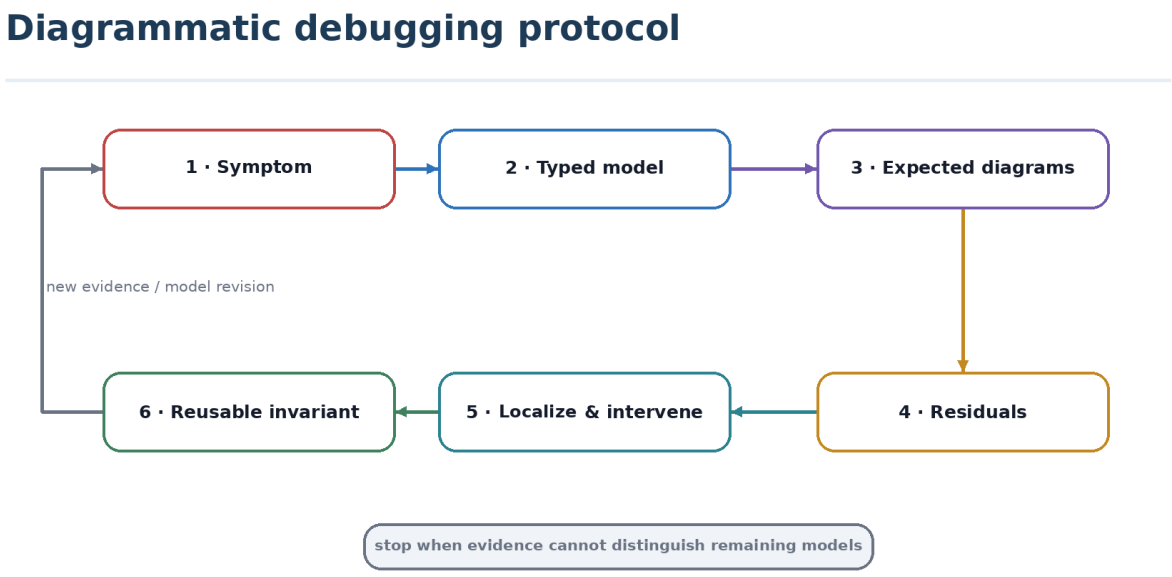


Figure 20. The protocol moves from symptom to typed model, failed diagram, localization, intervention, and reusable invariant.

Relationship to established debugging methods

Program slicing restricts code to statements relevant to a variable or computation (Weiser 1984). Delta debugging minimizes failure-inducing inputs or changes (Zeller and Hildebrandt 2002). Spectrum-based methods rank program elements by statistical association with failures (Abreu et al. 2009). Their later work addresses simultaneous diagnosis of multiple software faults (Abreu, Zoeteweij, and van Gemund 2011). Diagrammatic

debugging complements these methods: slices and reduced inputs become subobjects; ranked suspects become candidate arrows; categorical tests check relationships among their evidence.

Stop conditions

Stop expanding a categorical model when:

- the next distinction cannot affect the operational decision;
- the model has no testable law;
- the cost exceeds the incident value;
- the data cannot distinguish rival models;
- ordinary local reasoning already settles the issue.

We do not need to extend the categorical model when the additional structure does not help the investigation.

Chapter 23 — Case study: a distributed timeout

> ***Scenario:*** *A checkout request returns HTTP 504. The gateway trace marks the request as a timeout, the order service terminal log says "submitted," and a payment metric reports one success.*

The initial contradiction

All three observations can be locally true: the gateway timed out while the order service completed asynchronously and payment succeeded. The error is to place them in one outcome category without modeling time and business semantics.

Define execution objects:

```
q0 request accepted
q1 order submitted
q2 payment authorized
q3 gateway response sent
q4 client outcome observed
```

Arrows include submission, authorization, response, and observation. Some are concurrent. The gateway's 504 describes the transport outcome at $q3$; the order log describes a business transition at $q1$; the metric aggregates authorizations at $q2$.

Normalization categories

Create two common codomains rather than one:

- $\mathcal{K}_{transport}$ for HTTP completion;
- $\mathcal{K}_{business}$ for order/payment outcome.

The gateway and client observations map to the first, and the order log and payment metric map to the second. A separate policy map relates business completion to the response contract.

The apparent triangle no longer claims that 504 = submitted = success.

The true failed square

The service contract states that if payment is authorized before the response deadline, the client response should be success or a recoverable accepted status. The trace shows authorization before the gateway timeout, but the client received 504. Two routes from payment authorized to client outcome disagree:

1. contract semantics predict accepted/success;
2. observed gateway path yields timeout.

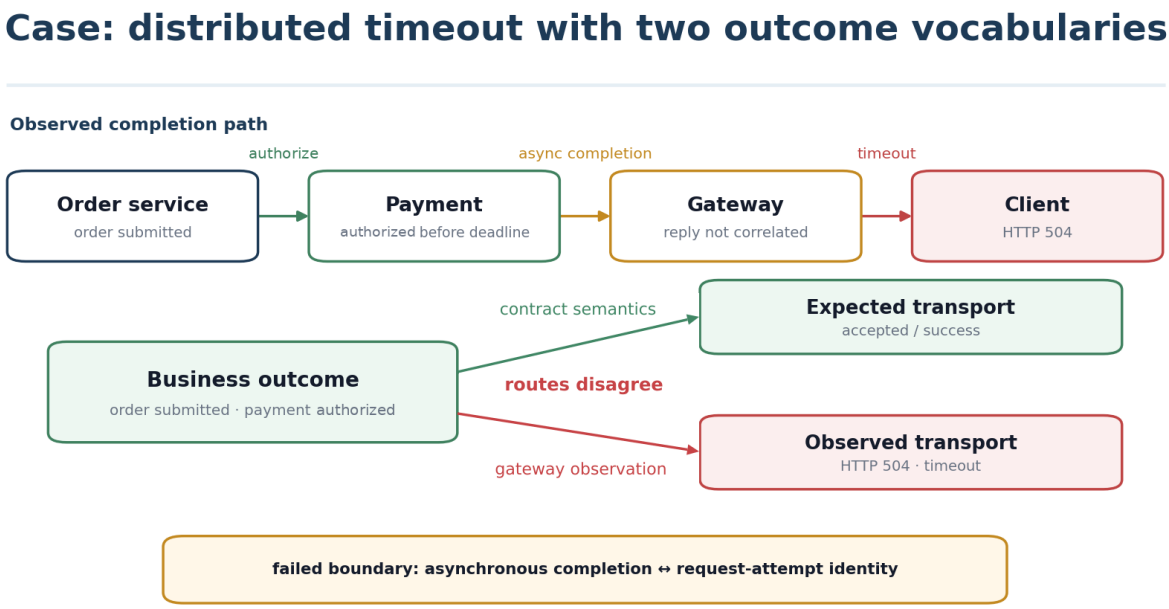


Figure 21. The timeout case becomes diagnostic only after transport and business outcomes are separated and the actual contract square is tested.

Localization

We decompose the response path:

```
payment callback -> order state update -> reply enqueue
-> gateway receive -> client send
```

The trace lacks a parent/link edge from the asynchronous payment callback to the original response activity. Logs share an order identifier but not the request attempt. A pullback on order ID correlates the business operations, but it cannot establish which gateway attempt should receive the result.

The smallest failed boundary is the mapping from asynchronous completion to request-attempt identity. The transport timeout is real; the business success is real; the categorical invariant violation lies in the response-correlation contract.

Corrective invariant

We add an immutable attempt identifier propagated through the callback and a test:

> *For every authorized payment linked to an open request attempt, the response-enqueue event and the gateway-receive span must form a compatible pullback over that attempt identifier within the deadline interval.*

The fix is not “make all signals agree.” It is “make the intended relationship observable and test it.”

Chapter 24 — Case study: a process memory leak

> ***Scenario:*** *Process private bytes rise over six hours. Three memory dumps show growing process heaps. Allocation stacks are available only in the final dump.*

Evidence categories

The metric series contains time-indexed aggregate states, and the dumps contain snapshots of memory regions and heap structures. Symbol files map addresses to modules and functions, and a leak hypothesis refers to retained allocations lacking intended release.

The source work uses a similar example when describing concrete execution artifacts, problem patterns, and analysis-pattern functors (Vostokov 2024b, 271–72). It also decomposes heap analysis into elementary operations such as symbol setup, instrumentation checks, heap listing, memory dumping, and searching (Vostokov 2024b, 268–71).

The pullback that matters

Joining by process ID is insufficient if the process restarted. We use a process-instance identity combining boot/session context, creation time, executable identity, and deployment build. We pull back metric samples and dumps over this object.

Then define snapshot comparison arrows that preserve heap identity and allocation class as far as the data permits. Symbolization is a separate functor whose correctness depends on binary-symbol version compatibility.

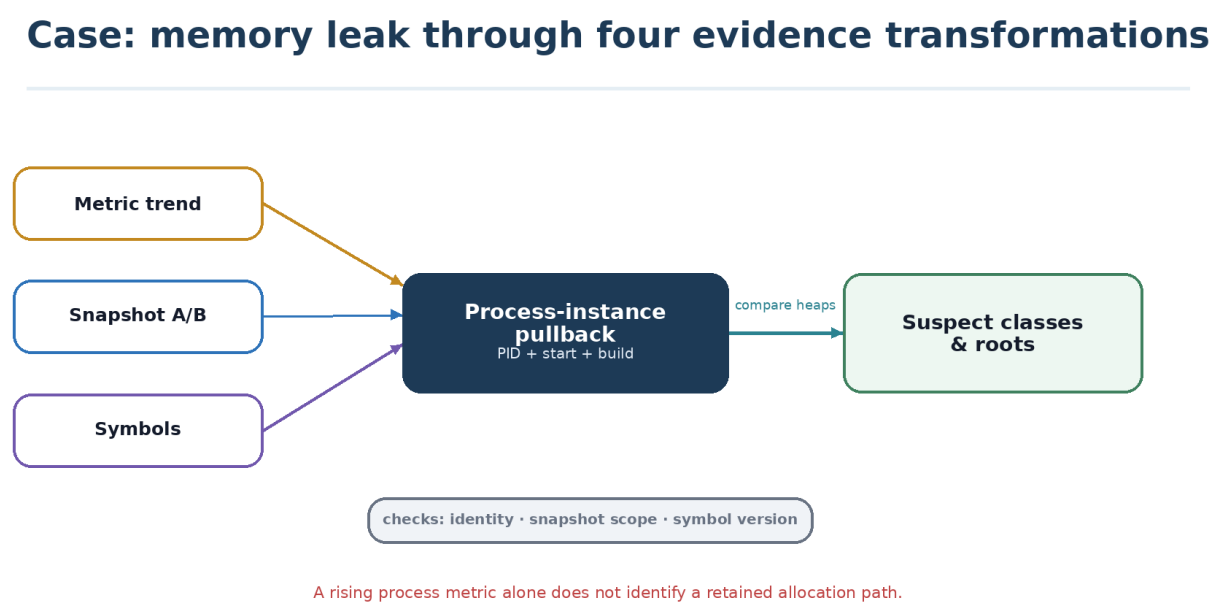


Figure 22. The leak analysis composes metric trend, process-instance correlation, snapshot comparison, and symbolization without treating any one view as complete.

Failed invariants

Three invariants are tested:

1. **Process identity:** every correlated metric and dump belongs to the same process instance.
2. **Symbol naturality:** symbolizing before or after selecting the growing allocation class gives equivalent module/function attribution.
3. **Growth consistency:** the aggregate increase predicted from comparable retained allocations agrees, within known untracked regions, with the private-bytes trend.

Suppose invariant 2 fails only for the final dump. We then discover that the symbol server supplied files for a later build. The apparent module responsible for growth is an analysis-path defect, not a runtime cause. Correct symbols restore commutativity.

Invariant 3 still shows a large residual: heap growth accounts for only 30% of private-byte growth. The investigation expands to mapped regions, stacks, or native allocators instead of declaring the first visible heap pattern the root cause.

Conclusion

Category theory does not detect the leak. It prevents three common errors: joining different process instances, trusting a non-natural symbolization path, and equating one growing substructure with the entire metric increase.

Chapter 25 — Case study: duplicate completion under concurrency

> ***Scenario:*** *A job occasionally emits two completion events, charges twice, and then reports one successful trace.*

Partial order, not log order

Two workers race after a lease-renewal delay. Their logs are merged by timestamp, but clock uncertainty makes the order unreliable. We build a happens-before category from verified local program order and message edges, and the two charge operations are incomparable until they reach the payment service.

The intended state machine has an idempotent completion arrow. Two paths should have the same external effect:

$$\text{complete} \circ \text{complete} = \text{complete}.$$

We have an idempotence law in the chosen effect category, not a general category axiom.

Evidence mismatch

The application trace samples one worker and shows one completion. The audit log records both charges, while the business metric aggregates two charges but one job. We map all three into a common category of (job, charge-attempt, outcome) and find that the trace functor is outside the shared domain for the unsampled worker; absence cannot be interpreted as non-execution.

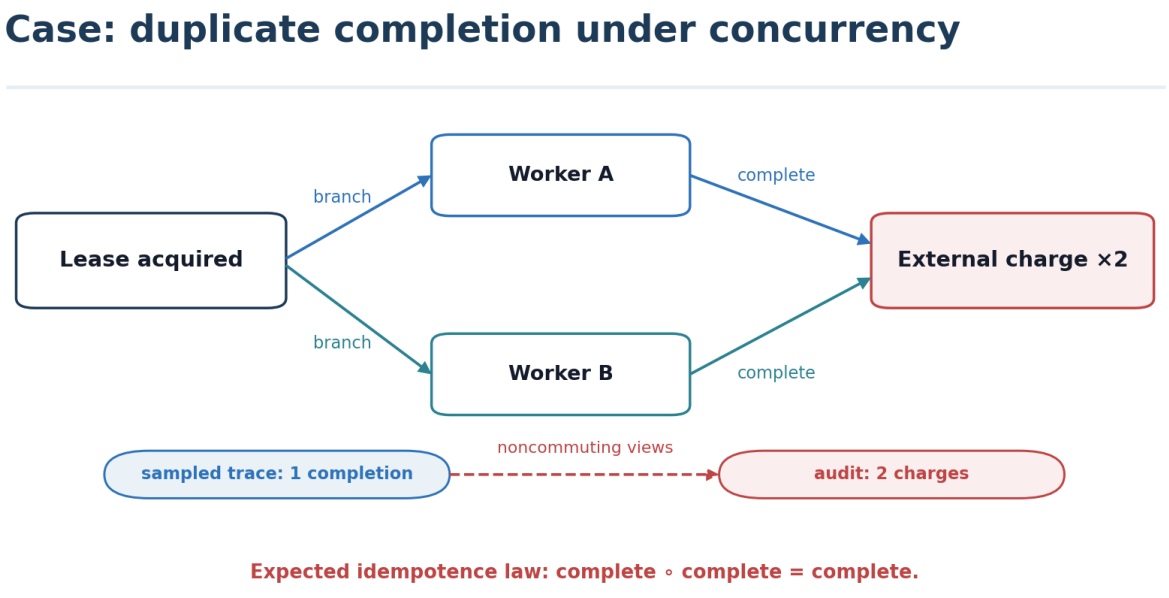


Figure 23. Concurrent completion paths violate the declared idempotent effect law even though a sampled trace shows only one path.

Smallest failed square

Two routes from lease acquisition to external effect should agree:

1. worker A completes, then worker B observes completed state;
2. worker B completes, then worker A observes completed state.

Both workers read the pre-completion state because the check and charge are not one atomic or idempotent operation. The square fails at the state-store/payment boundary.

Corrective model

Introduce a unique completion token and an atomic `claim-completion` transition. The payment operation becomes a morphism from a successfully claimed token, not directly from a job identifier. The new invariant tests that every charge factors through exactly one claim.

The phrase “exactly one” requires a model supporting the relevant uniqueness claim. In a database implementation, a unique constraint or transactional compare-and-set can realize it.

Chapter 26 — Case study: semantic-convention drift

Scenario: *After a telemetry-library upgrade, server-error dashboards fall by 40%, but customer error reports do not change.*

Versioned schemas

The old library records an unset status for application errors with HTTP 200 and an error event, and a normalizer classified the event as server error. The new library records status according to a newer semantic policy, while the normalizer still expects the old event structure.

Let $S_0 \to S_1$ be the semantic-schema migration and N_0, N_1 the corresponding normalizers into a common outcome category. For an event migration V and intended outcome preservation, test:

$$N_1 \circ V \stackrel{?}{=} N_0.$$

The square fails for application errors with successful transport status.

Why field-level diffing missed it

All required fields still exist. Their relationship changed: status now summarizes a different layer, and the error event carries the business failure. A schema diff sees names and types, and a categorical contract sees the path from event structure to outcome semantics.

Corrective action

We version the normalizer with the semantic convention, migrate historical data through the matching path, and maintain golden fixtures containing transport success plus application failure. The invariant belongs in CI and in backfill validation.

Novelty of the diagnosis

The category-theoretic contribution is not the idea of versioning schemas. It is the use of a commuting migration square to distinguish application change from observation change and to localize the semantic adapter responsible for the apparent improvement.

Chapter 27 — Case study: an agentic AI tool chain

> ***Scenario:*** *An agent tells a user that a change was deployed, while the deployment API audit trail shows no successful write.*

Multiple narratives

An agentic workflow has at least these objects:

```
user intent -> agent plan -> tool request -> tool response
-> world state -> agent statement -> user belief
```

Artifacts include model messages, tool-call records, API audit logs, deployment state, and final prose. These are not one trace. They are several observation functors over a partially shared world.

The accountability square

We map tool responses and independently observed world state into a common category of deployment outcomes, and we map the agent's final claim into the same category through a statement interpreter. Two invariants matter:

1. the tool-result interpretation must be natural with respect to the actual deployment transition;
2. the agent claim must factor through a verified tool result or verified world observation.

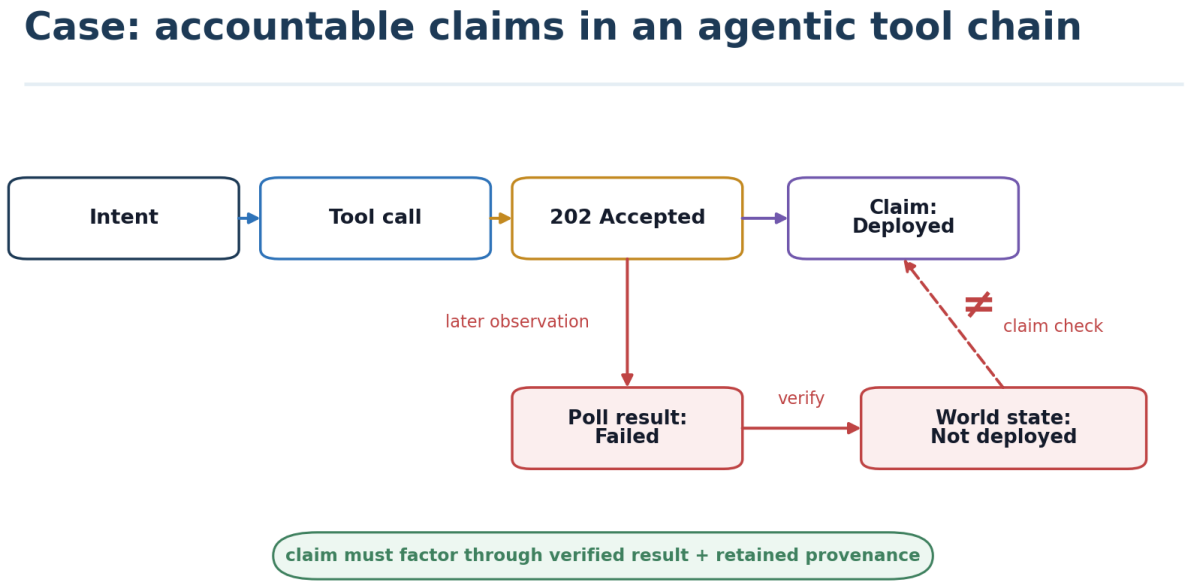


Figure 24. Agent accountability requires the final claim to agree with independently observed world state and to preserve provenance through tool results.

Violation

The tool returned a transport-level 202 with an operation identifier. A later poll failed, but the poll result was omitted from the context window. The agent normalized `accepted` as `deployed`. The audit log and world-state observer correctly report no successful deployment.

We have a naturality failure between the tool-protocol view and the world-state view, plus a provenance failure in the claim path. It is not necessarily "hallucination" in the narrow sense of inventing an API call; it is a semantic and contextual transformation defect.

Corrective invariants

- `accepted` may map only to `pending`, never directly to `succeeded`;
- every success claim factors through a terminal success response or an independent state observation;
- context restriction preserves unresolved operation identifiers until terminal state;
- the audit/log and agent/tool views satisfy cross-view residual thresholds.

Broader use

The same structure applies to security agents, automated remediation, code-generation agents, and LLM-based diagnostic assistants. Category theory provides the accountability skeleton, and access control, uncertainty, evaluation, and human approval remain separate requirements.

Part VI — Engineering, Evaluation, and Research

Chapter 28 — Automating categorical diagnostic checks

> ***Diagnostic question:*** *Which parts of the framework can be compiled into routine tests, and which still require human modeling judgement?*

What can be automated

Once categories and mappings are specified, many checks are mechanical:

- source/target typing of arrows;
- identity and composition closure;
- functoriality on generated paths;
- naturality squares for selected edges;
- pullback/equalizer queries;
- presheaf restriction laws;
- overlap residuals for sheaf-like assignments;
- operad input/output compatibility;
- equivalence of declared pipeline paths;
- version compatibility of adapters.

Automation does not remove our creative work of choosing useful objects, arrows, equality, and scope. It makes those choices executable and reviewable.

A compact diagram specification

```yaml
diagram: request_outcome_agreement
version: 3
domain: CompletedRequest
views:
  log:
    observe: app_terminal_log
    normalize: log_outcome_v4
  trace:
    observe: root_span
    normalize: span_outcome_v6
codomain: RequestOutcomeV2
law: naturality
comparator:
  kind: equality
missing:
  uninstrumented: outside_domain
  exporter_loss: violation
dimensions: [service, operation, build, collector_version]
```

The file is not category theory by syntax. Its semantics must define the categories and functors referenced by each identifier.

Path compilation

We represent a diagram as a directed multigraph with typed arrows. We enumerate parallel paths up to a bounded length or use declared equations to generate only relevant pairs. Each path compiles to a query or transformation pipeline.

For large diagrams, exhaustive path comparison can be exponential. Use a selected basis of generating commutative cells. If the large diagram is built by pasting commuting cells, the whole diagram commutes under the relevant categorical conditions. This makes small invariant cards the unit of scaling.

Residual evaluation

A residual evaluator should return more than pass/fail:

```
{
  "diagram": "request_outcome_agreement",
  "object": "trace/4f91...",
  "left_result": "server_error",
  "right_result": "ok",
  "residual": 1.0,
  "missing_edges": [],
  "versions": {
    "log_normalizer": 4,
    "span_normalizer": 6
  },
  "smallest_failed_cell": "status_mapping"
}
```

Preserving both results and versions makes the violation auditable.

Localization algorithms

Cell ranking: Evaluate every elementary square and rank by residual.

Binary path split: For a long pair of composites, compare compatible intermediate objects and recurse on the failing segment.

Overlap filtration: In a sheaf-like model, threshold overlap residuals and observe how the cover separates as tolerance tightens (Robinson 2020).

Version bisection: Evaluate the same diagram across builds, schemas, or collector versions to find the first noncommuting version.

Counterexample minimization: Apply delta debugging to the evidence or transformation set while preserving the violation (Zeller and Hildebrandt 2002).

Property-based testing

We generate execution fixtures from the source category's generators and relations. For each fixture, assert preservation. We include identities, short composites, alternative paths, concurrency, retries, cancellation, duplicates, missingness, and schema boundaries.

This approach is particularly suitable for instrumentation libraries and normalizers, and it tests structural laws over many event combinations instead of relying only on golden serializations.

Testing artifact transformations

Consider an analysis that groups trace messages by request identity and reports the number of complete request pairs. We replace every request identifier by a new identifier using one consistent one-to-one renaming, including all references to that identifier. If the analysis does not depend on the identifier's literal value, the number of complete pairs should remain unchanged. If the report retains request labels, those labels should change according to the same renaming.

This gives us a metamorphic test. We analyze the original artifact and then rename its report, or rename the artifact and then analyze it. We compare the two resulting reports under a declared equality policy. The preconditions matter: a partial renaming, a collision between identifiers, or an analysis that intentionally classifies identifiers by their contents would require a different test. This is the input/output-relation approach of metamorphic testing (Chen et al. 2018); Appendix B.25 writes out the comparison.

We can apply the same testing discipline to Chapter 31's example. A deliberately faulty classifier that labels every zero-byte record as 2xx agrees with the record having status 200 and fails on the two records having status 304. Either 304 record supplies a counterexample to the shortcut. This is an injected test defect, not a newly observed defect in the public artifact.

Property-based generation can supply additional well-typed records and renamings. Separate tests should exercise invalid inputs, missing fields, reused identifiers, and changed schema versions. The generator must record which assumptions it enforces, and a failing test should retain the input, both transformation results, the versions, and the comparison rule. QuickCheck is an established example of generating test inputs for executable properties (Claessen and Hughes 2000).

A commuting test square is not automatically a natural transformation. For that stronger claim, we must supply the source category, the two functors, the component arrows, and the required naturality equations. Testing selected inputs provides evidence about an implementation; it does not prove an unrestricted categorical law.

Quantitative dashboards

Aggregate residuals carefully. A mean can hide a small class of severe mismatches. Useful summaries include:

- fraction of objects within tolerance;
- residual percentiles;
- unmatched-left and unmatched-right rates;
- largest failed cells;
- first failing version;
- residual by semantic operation and propagation boundary;
- estimated coverage of the observation domain.

The dashboard should always link back to concrete failed diagrams.

Chapter 29 — A category-aware diagnostics architecture

> ***Diagnostic question:*** *What platform components are needed to make diagrammatic checks reusable across tools and teams?*

Architectural components

Schema and ontology registry: Stores category presentations, object and arrow types, equations, semantic versions, and viewpoint mappings.

Observation adapters: Map logs, traces, metrics, dumps, profiles, and external records into modality categories.

Normalization functors: Map modalities into shared comparison categories without pretending the modalities are identical.

Diagram registry: Stores invariant cards, tolerances, ownership, and fixtures.

Compiler: Turns paths and universal constructions into streaming queries, batch queries, or debugger commands.

Evaluator and localizer: Computes residuals and smallest failed cells.

Provenance store: Records artifacts, transformations, versions, and hashes.

Diagnostic workspace: Presents failed diagrams alongside raw and derived evidence.

Pattern-operad registry: Types and composes multi-input diagnostic operations.

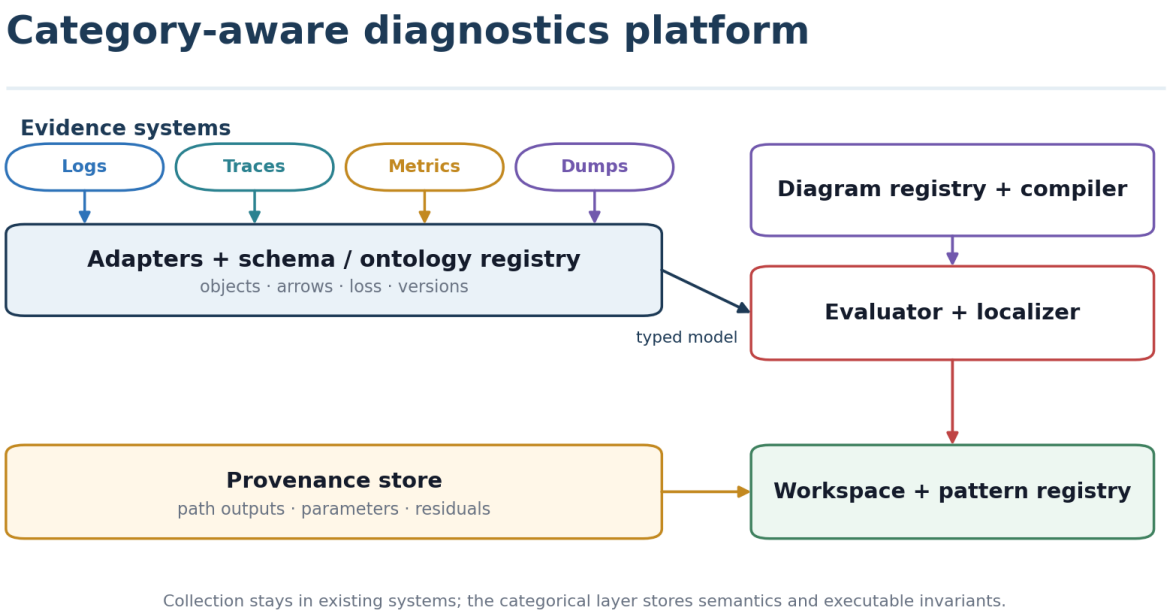


Figure 25. A category-aware diagnostics platform separates schemas, adapters, invariant compilation, evaluation, provenance, and human investigation.

Integration with existing telemetry

The architecture does not require a new collection protocol. OpenTelemetry can supply standard signal representations and context (OpenTelemetry Authors 2026b; W3C Distributed Tracing Working Group 2021). Existing log stores, trace backends, metric databases, dump repositories, and notebooks remain evidence systems. The categorical layer stores semantics and compiles cross-system checks.

Open interfaces

Each adapter should expose:

- source schema and version;
- target category and version;
- object-identity rules;
- arrow construction rules;
- preservation profile;
- loss and missingness policy;
- conformance fixtures;
- provenance for every produced object and arrow.

An adapter that emits only flattened rows is insufficient for relationship-level diagnostics.

Security and privacy

Cross-signal correlation can increase sensitivity by linking identities that were previously separate. The world model and pullback keys therefore need access control, retention, purpose limitation, and redaction rules. A mathematically valid join may be operationally or legally impermissible.

Provenance also creates accountability. Every derived claim can reveal which artifacts and transformations were used without exposing unrestricted raw data to every consumer.

Incremental adoption

We start with one high-value invariant at one boundary:

1. choose a recurring incident class;
2. model two views and one common codomain;
3. implement one square;
4. run it offline on historical incidents;
5. add versioning and CI fixtures;
6. expand only after the square changes decisions or catches drift.

The smallest useful categorical platform is a versioned adapter pair and a failed-square report.

Chapter 30 — Limits, evaluation, and a research agenda

Diagnostic question: *How can this framework be tested as engineering rather than admired as notation?*

Limits of the framework

Model dependence: Results depend on chosen objects, arrows, equality, and scope. A beautifully commuting wrong model is still wrong.

Partial evidence: Production telemetry is sampled, delayed, lossy, and perturbed. Exact categorical structures may need partial, lax, enriched, probabilistic, or sheaf-theoretic extensions.

Causality: Commutativity expresses agreement of routes, not causal responsibility. Causal claims need appropriate causal models, interventions, or domain semantics.

Cost: Maintaining schemas, adapters, diagrams, and provenance has organizational and computational cost.

Category mistakes: We can easily call any graph a category, any mapping a functor, any reverse operation an adjoint, any local data a sheaf, or any workflow an operad. Appendix B provides a checklist against these errors.

Human interpretation: A residual must still be understood in context. Automation can localize a mismatch without deciding its operational importance.

Evaluation questions

A credible evaluation should ask:

1. Does the framework find real inconsistencies earlier than existing dashboards or tests?
2. Does localization reduce investigation time or artifact volume?
3. How many violations are system defects, instrumentation defects, schema defects, or model defects?
4. What modeling and maintenance effort is required?
5. Do invariant cards transfer across services or organizations?
6. Can incident explanations be reproduced from stored provenance?
7. Does the framework improve instrumentation design or only post-hoc analysis?

Suggested study design

We select several recurrent incident classes and construct invariants without using the final root cause. Then we replay historical telemetry or controlled fault injections and compare:

- time to first useful suspect;
- precision of localized boundaries;
- number of artifacts inspected;
- false-positive and unknown rates;
- modeling hours;

- performance overhead;
- recurrence detected by the added regression invariant.

Blind review of the resulting explanations can reduce confirmation bias.

Research directions

Approximate naturality: Develop enriched or probabilistic codomains and calibrated residuals for noisy telemetry.

Partial functors: Formalize sampling, missing context, and exporter loss without collapsing all absence into one value.

Diagnostic sites and toposes: Define scope categories, observational-coverage topologies, evidence sheaves, and restriction-stable hypothesis subobjects; test whether contextual truth improves obstruction localization and diagnostic decisions.

For memory evidence, the source work proposes open and closed memory regions and region strata as candidate covering structures, and later names Structure Sheaves as distributed logs (Vostokov 2024b, 257, 354). Which scope category, Grothendieck topology, sections, and restriction maps make any such construction satisfy the sheaf axioms and support a useful diagnostic topos remains an open question. We therefore keep this item as a research direction, not a promoted theorem.

Observability equivalence: Characterize which execution distinctions are collapsed by a telemetry configuration and optimize the minimum separating instrumentation.

Pattern Category Theory: Treat pattern languages, pattern transformations, and pattern composition as a categorical system while preserving empirical grounding.

Diagram learning: Learn candidate commutative diagrams from normal executions, but require human validation of object, arrow, and equality semantics.

Agent accountability: Specify naturality and provenance laws across intent, tool calls, world state, and generated claims.

Category-aware diagnostic agents: Make agents produce typed evidence paths and explicit failed diagrams rather than unstructured explanations.

Final perspective

Category theory does not make software transparent, but it makes the contracts among partial views explicit. That is already a major shift. Logs, traces, metrics, dumps, profiles, code, and hypotheses cease to be a heap of loosely correlated evidence and become objects connected by named, typed, versioned transformations.

The practical ideal is modest and powerful: when two routes should agree, say exactly why; when they do not, preserve the discrepancy; when the model is only an analogy, label it, and when an incident teaches a new invariant, make that invariant executable.

Chapter 31 — Case study: a real HTTP log artifact

Diagnostic question: What categorical claims does a real log support, and which arrows would the analyst be inventing?

This case study uses actual records rather than an invented incident narrative. Its purpose is empirical and methodological: start with the artifact, identify the transformations that can be reproduced, and refuse to promote temporal proximity or a shared host label into causality without evidence.

Artifact and provenance

The Internet Traffic Archive describes the NASA-HTTP corpus as HTTP requests received by the NASA Kennedy Space Center web server. Its July trace covers 1–31 July 1995, and each line contains a host, timestamp, request, HTTP reply code, and byte count (Internet Traffic Archive 1995).

The file checked for this edition is NASA_access_log_Jul95.gz, 20,676,672 bytes, with SHA-256 199109ed0f273e095da6ccd5fc9dc4cd8bb58daa06d62135e62090fea9d27488. Recording the filename, byte length, and digest makes the worked result reproducible and distinguishes this exact artifact from a reformatted mirror.

The archive asks users not to analyze the data beyond general traffic patterns. Accordingly, host strings are pseudonymized below as H1, H3, and H4, and no host-level behavior is inferred. The 1995 artifact is used only to test representation and classification contracts.

A five-record excerpt

The following rows are lines 1, 4, 5, 6, and 7 of the decompressed July file. All timestamps are on 1 July 1995 with offset –0400. The target column is extracted from the quoted request.

Table 31.1. Five verified records from NASA_access_log_Jul95.gz; host labels are pseudonyms.

Host	Time	Request target	Status	Bytes
H1	00:00:01	/history/apollo/	200	6245
H4	00:00:11	/shuttle/countdown/liftoff.html	304	0
H3	00:00:11	/shuttle/missions/sts-73/sts-73-patch-small.gif	200	4179
H4	00:00:12	/images/NASA-logosmall.gif	304	0
H4	00:00:12	/shuttle/countdown/video/livevideo.gif	200	0

Two observations already constrain the model. Lines 4 and 5 share a one-second timestamp but not a host, and lines 6 and 7 share a one-second timestamp and a host pseudonym. The file records order at one-second resolution, but it contains no request identifier, session identifier, referrer, span identifier, or parent-child field. Neither pair therefore licenses a causal arrow.

Two categories, not one invented graph

The artifact supports two different categorical constructions, and keeping them separate prevents an important modeling error:

- Representation category. Its objects are types such as RawLine, ParsedRecord, TypedFields, NormalizedRecord, and DiagnosticOutcome. Its arrows are reproducible transformations such as parse, project, type, normalize, classify, and summarize.
- Event or causal category. If individual requests are treated as objects, this file directly supplies identities and a coarse file order. It does not supply causal, session, or parent-child morphisms. Those arrows require another artifact or an explicitly labeled hypothesis.

A sequence number derived from line position can define a total file order, but that is an ingestion order, not a proof of execution causality. Likewise, a shared host string can support grouping, but grouping is not composition. The category must record which relation each arrow means.

The pasted parser–normalizer check

We apply Figure 5A to every selected line. The upper route parses the quoted fields and normalizes them. The lower route extracts the status and byte fields directly, types them, and classifies the result. We use status families 2xx and 3xx, and we use payload flags zero-payload and positive-payload. Under exact equality, the two routes agree for all five rows.

$$\text{line 4: } (304, 0) \mapsto (\text{3xx, zero-payload})$$

$$\text{line 7: } (200, 0) \mapsto (\text{2xx, zero-payload})$$

The pair is deliberately revealing: both records have zero bytes, but their status families differ. A byte-only shortcut maps them to the same payload observation and cannot implement the declared diagnostic outcome. This is information loss. It should not automatically be called non-faithfulness: in category theory, faithfulness is a property of how a functor maps morphisms, not merely whether a value-level function is injective.

Residuals and their interpretations

For a categorical invariant card, store the left residual, right residual, and outer residual separately. Their meanings are local to the declared transformations:

- A left-cell residual points first to line extraction, parsing, field projection, or field typing.
- A right-cell residual points first to normalization, projection, status classification, or summarization.
- An outer residual with no matching local residual usually means the tests did not use the same record identity, software version, fixture, or equality policy.
- A requested causal residual is undefined here, because the log supplies no declared causal arrow to compare. Missing structure is not a zero residual.

Reproducible check

1. Verify the compressed file length is 20,676,672 bytes and its SHA-256 is 199109ed0f273e095da6ccd5fc9dc4cd8bb58daa06d62135e62090fea9d27488.

2. Read the file as Latin-1 text and select decompressed lines 1, 4, 5, 6, and 7.
3. Parse host, bracketed timestamp, quoted request, three-digit status, and byte token; keep a dash byte token as missing rather than silently converting it to zero.
4. Project status and bytes through both routes and compare the typed pair exactly before applying the status-family and payload classifiers.
5. Record the equality policy, parser version, selected line numbers, and pseudonymization rule with the result.

The five selected records pass both local commuting-square checks and the pasted outer check under that policy. We obtain a validation result about the parser and classifiers, but it is not an assertion that every line in the corpus has been validated by this example.

Limits

This is not a root-cause reconstruction of a NASA incident, and it does not turn a common-log-format file into a distributed trace. The corpus is historical, the excerpt is intentionally small, timestamp resolution is coarse, byte-field semantics depend on the server's logging convention, and the archive's privacy notice limits appropriate analysis. Those constraints are part of the model rather than footnotes to be ignored.

Practical lesson

We begin with transformations the artifact can support, and we paste local checks only across genuinely shared, typed interfaces. We preserve status and missingness distinctions until the diagnostic question permits a quotient. When an arrow is absent, add another source, state a hypothesis, or leave the hom-set empty: do not manufacture causality from adjacency.

Appendix A — Prerequisites from school mathematics

This appendix starts below university level on purpose. Category theory uses unfamiliar words, but its first ideas need only the mathematics of collections, arrows, and rules. If you can follow a route on a map and understand that doing step A and then step B may have the same result as doing step C, you already possess the central intuition.

Our goal is not to turn the reader into a pure mathematician but to make every symbol used in the main text readable and every claim checkable.

A.1 Reading symbols

Mathematics is compressed prose. The symbol

$$f: A \to B$$

reads: "f is a function from A to B." The left-hand collection A is the **domain**; the right-hand collection B is the **codomain**. If x is an item in A, then $f(x)$ is the item in B assigned to it.

Several symbols recur throughout the book:

- $x \in A$: x is an element of A;
- $A \subseteq B$: every element of A is also an element of B;
- $A \times B$: the collection of ordered pairs (a, b) with $a \in A$ and $b \in B$;
- $f \circ g$: first do g, then do f;
- id_A: the identity operation on A;
- $f = g$: the two expressions have the same value at the declared level of comparison;
- $x \mapsto y$: x is assigned or sent to y;
- $\forall x$: for every x;
- $\exists x$: there exists at least one x;
- $\Rightarrow$: implies;
- $\cong$: is isomorphic to—structurally the same by a reversible mapping;
- $\simeq$: is equivalent to, in a sense that depends on the context.

An arrow is not automatically a function between sets. In a category it may represent a state transition, schema migration, causal relation, inclusion, refinement, or diagnostic transformation. The arrow notation says that the thing has a declared source and target and may compose with suitable arrows.

A.2 Sets and elements

By a **set**, we mean a collection in which membership is well defined. Examples include:

$$P = \{\text{running}, \text{blocked}, \text{terminated}\}$$

and

$$S = \{\text{all spans in trace 42}\}.$$

Order and repetition do not matter inside an ordinary set. Thus $\{a, b\} = \{b, a\}$, and writing $\{a, a, b\}$ still names the set $\{a, b\}$. Logs, however, normally preserve order and repetition, so a log is not adequately modeled by a plain set when those properties matter.

The empty set $\varnothing$ has no elements. It is different from a set containing a special value such as {unknown}. This distinction matters in telemetry: "no event was returned" and "an event explicitly reported unknown" are not the same observation.

A **subset** $A \subseteq B$ contains only elements already in B. The set of failed requests may be a subset of all requests. The **intersection** $A \cap B$ contains elements in both sets. The **union** $A \cup B$ contains elements in either. These operations are useful, but cross-source joins require more than intersection when the sources contain different representations of the same execution.

A.3 Tuples, products, and records

An ordered pair (a, b) remembers which value is first and which is second. In general, $(a, b) \neq (b, a)$. The product $A \times B$ is the set of every possible pair with a first component from A and a second from B.

We can view a telemetry record as a tuple:

$$r = (\text{trace-id,span-id,timestamp,status}).$$

We have projection functions that retrieve components:

$$\pi_1 : A \times B \to A, \qquad \pi_2 : A \times B \to B.$$

In programming, these resemble field accessors. A categorical product adds a universal claim: any object that supplies a compatible A-component and B-component maps uniquely to $A \times B$. The ordinary tuple gives the intuition; Appendix B states the universal property.

A.4 Functions

A **function** assigns exactly one output to every input in its domain. Different inputs may have the same output. For example, a function that maps every HTTP status code to a class maps 200 and 204 to success.

A function is:

- **injective** if different inputs never produce the same output;
- **surjective** if every codomain value is produced by at least one input;
- **bijective** if it is both injective and surjective.

For a bijection, we have an inverse. If $f : A \to B$ is bijective, there is a function $f^{-1} : B \to A$ satisfying

$$f^{-1} \circ f = \mathrm{id}_A, \qquad f \circ f^{-1} = \mathrm{id}_B.$$

Software observations are rarely bijective. Aggregation, sampling, truncation, and redaction cause several runtime states to produce the same visible record. That failure of injectivity is a precise form of diagnostic blindness.

A **partial function** is undefined for some possible inputs. A symbol resolver may lack a module's symbols; a trace mapper may have no parent context for an orphan span. Treating a partial function as total by inventing a generic null can merge importantly different failure modes.

A.5 Composition and identity

Suppose

$$f: A \to B, \qquad g: B \to C.$$

The output type of f matches the input type of g, so they compose:

$$g \circ f: A \to C.$$

Read $g \circ f$ from right to left: do f, then g. If f parses a log line into a record and g classifies the record, $g \circ f$ classifies the line.

Every collection has an identity function:

$$\mathrm{id}_A(a) = a.$$

Composition obeys two school-algebra-like laws. For composable functions f, g, and h,

$$h \circ (g \circ f) = (h \circ g) \circ f$$

and

$$f \circ \mathrm{id}_A = f = \mathrm{id}_B \circ f.$$

The first law is **associativity**: parentheses do not change the result of a fixed route. The second is the **identity law**: a genuine do-nothing step changes nothing. These are the two defining laws of a category.

A.6 Relations

A **relation** between A and B specifies which pairs (a, b) are related. Unlike a function, one item may relate to many items or none. Examples include:

- a request is represented by several log messages;
- a source line contributes to several machine instructions;
- two spans overlap in time;
- a dump object is reachable from several roots.

An **equivalence relation** on a set obeys three laws:

1. reflexivity: $x \sim x$;
2. symmetry: if $x \sim y$, then $y \sim x$;
3. transitivity: if $x \sim y$ and $y \sim z$, then $x \sim z$.

It partitions a set into equivalence classes. For diagnostics, equality of values produced by a normalization function automatically defines an equivalence relation: two observations are equivalent when the function returns the same value. A fuzzy or threshold-based claim that two signatures are "the same" must instead be checked for reflexivity,

symmetry, and especially transitivity. A **quotient** keeps the classes rather than the individual elements. Quotienting reduces detail deliberately; it should therefore come with a loss statement.

A **preorder** is reflexive and transitive. A **partial order** is also antisymmetric: if $x \leq y$ and $y \leq x$, then $x = y$. The happens-before relation in distributed execution is generally a partial order, not one total timestamp order. Two events may be incomparable.

A.7 Graphs, paths, and trees

A directed graph consists of **vertices** and directed **edges**. A service dependency graph may use services as vertices and calls as edges, and an execution graph may use events as vertices and temporal or causal transitions as edges.

By a **path**, we mean a sequence of edges whose endpoints match. Paths compose by concatenation. A path of length zero stays at one vertex; it supplies the identity needed to turn a directed graph into its **free category**. Nothing forces two different paths with the same endpoints to be equal. Equations may be added later if the domain says the routes should have the same result.

A **directed acyclic graph** has no directed cycles. A **tree** has a distinguished root and a unique route from the root to each node. Many trace visualizations look like trees, but links, retries, asynchronous work, and fan-in often make the actual execution a more general graph.

A.8 Logic and counterexamples

By a diagnostic invariant, we mean a conditional statement. In prose:

> *If an accepted request completes successfully and propagation is enabled, then a server span with the same trace context should be observable within the export window.*

The statement has assumptions and a conclusion. If an assumption is false (perhaps sampling excludes the request), the conclusion need not follow. A failed conclusion is meaningful only after the scope and assumptions are recorded.

The universal statement $\forall x \in A, P(x)$ says that every relevant x has property P. One counterexample is enough to disprove it. A single trace with broken context propagation can refute a universal propagation law, but it does not establish why propagation broke.

An existential statement $\exists x \in A, P(x)$ needs one witness. "There exists a path that preserves the request identity" can be established by exhibiting the path and checking its mappings.

Do not confuse implication with equivalence. From $P \Rightarrow Q$ and Q, one cannot infer P. An error log may follow a timeout, but seeing that log does not prove the timeout was the only possible cause.

A.9 Equality and approximation

The symbol = is never innocent. It requires a chosen level of comparison. Two timestamps may be equal as integer milliseconds but unequal as nanoseconds. Two error texts may be equal after redaction but different byte for byte. Two execution paths may be equal by business outcome but different in resource consumption.

Therefore every operational diagram needs an **equality policy**. Common policies include:

- exact equality;
- equality after normalization;
- equality of selected fields;
- equivalence under a declared relation;
- distance below a threshold;
- membership in an allowed set.

Approximate equality is often written $x \approx y$, but the symbol alone supplies no rule. State the distance, tolerance, unit, and handling of missing values.

A.10 Algebraic laws as testable promises

An operation $\star$ is **associative** if

$$(a \star b) \star c = a \star (b \star c).$$

It has an **identity** e if $e \star a = a = a \star e$. It is **commutative** if $a \star b = b \star a$, and **idempotent** if $a \star a = a$.

These laws have direct engineering consequences. If a merge is advertised as associative, workers may regroup partial results. If it is also commutative, they may reorder them. If it is idempotent, a retry may safely repeat a merge. The names do not make the laws true, and tests and implementation must establish them.

Category composition requires associativity and identities, but it does not require commutativity. A **commutative diagram** says that particular parallel routes are equal. It does not say that every operation may be swapped.

A.11 Proof habits for diagnostic models

For this book, a useful "proof" is often a disciplined check:

1. name every object and its type;
2. name every arrow, source, and target;
3. check that adjacent arrows compose;
4. write the routes being compared;
5. state the equality or residual rule;
6. list assumptions and visibility limits;
7. test normal, boundary, missing, and malformed cases;
8. preserve a counterexample with provenance.

This is stricter than drawing an evocative diagram and lighter than developing a complete formal semantics. The level of formality should match the consequence of being wrong.

A.12 Exercises

Exercise 1: Let f map an HTTP code to its hundred class and g map the class to success, client-failure, or server-failure. What is $(g \circ f)(503)$?

Exercise 2: A trace exporter maps 1,000 runtime spans to 100 exported spans. Can the mapping be injective? What additional fact would be needed to decide whether it is surjective onto its declared codomain?

Exercise 3: Give two different paths from "raw event" to "incident class". State one reason they should agree and one reason they might legitimately differ.

Exercise 4: Is timestamp order a partial order, a total order, or neither when clocks can tie and drift? State the exact timestamp representation and comparison rule before answering.

Exercise 5: A deduplication merge is claimed to be idempotent. Write the equation that must hold. Propose a test case involving a retried export.

A.13 Short answers

1. $f(503) = 5$, then $g(5) =$ server-failure.
2. It cannot be injective if every runtime span is in the domain and only 100 distinct outputs exist. Surjectivity depends on what the codomain contains, not only on the observed count.
3. For example, parse then classify versus extract a code then look it up. They should agree under a shared classification contract; they may differ when parsing loses a field or the versions differ.
4. There is no representation-independent answer. With unique logical-clock pairs it may be total for display; physical timestamps alone need not represent causality and ties complicate antisymmetry.
5. $m(x, x) = x$, or more generally $x \star x = x$. Export the same batch twice and verify that the merged state equals the state after one export under the declared equality.

Appendix B — Category theory: a diagnostic quick course

In this appendix, we give the mathematical spine of the book in one place. It is a bridge to standard introductions by Mac Lane, Riehl, Leinster, Pierce, and Fong and Spivak (Mac Lane 1998; Riehl 2016; Leinster 2014; Pierce 1991; Fong and Spivak 2019). The explanations are operational. Standard mathematical definitions are meant literally; diagnostic constructions and guiding analogies are identified by the surrounding prose under the taxonomy of B.19.

B.1 Categories

A **category** $\mathcal{C}$ consists of:

1. objects $A, B, C, \ldots$;
2. for each ordered pair of objects, a collection of morphisms $f: A \to B$;
3. an identity morphism $\mathrm{id}_A: A \to A$ for every object;
4. a composite $g \circ f: A \to C$ whenever $f: A \to B$ and $g: B \to C$;
5. associativity and identity laws.

The definition does not say what an object "really is." It may be a set, type, state, schema, scope, time interval, or system interface. It does not say what a morphism "really does." The meaning comes from the model, while the laws make routes composable.

A category cannot violate its own category axioms. If a software mapping fails associativity or identity, then that mapping did not realize the proposed category. With this distinction, we prevent a common error: saying "the category is anomalous" when the observed implementation fails the categorical model.

Basic examples

Set: Objects are sets, and morphisms are total functions. Composition and identity are ordinary function composition and identity.

A preorder as a category: Objects are elements. We have one arrow $x \to y$ exactly when $x \leq y$. Reflexivity supplies identities and transitivity supplies composition, and a time or refinement order can sometimes be modeled this way.

A monoid as a one-object category: For a monoid, we have an associative operation and identity. Use one object; each monoid element is an endomorphism of that object; multiplication is composition. A sequence of trace-message kinds can motivate this model, but ordering and interpretation must be defined.

Free category on a graph: Objects are graph vertices, and morphisms are finite directed paths, including length-zero paths. Composition concatenates paths. This is often the safest starting point for runtime events because it does not equate different paths without evidence.

Schema category: Objects may be entity types and arrows may be declared attributes or relations. Instances can then be represented by functors into **Set**, an approach used in categorical databases and ologs (Spivak 2012; Spivak and Kent 2012).

Small, large, and local categories

We call a category **small** when its objects and arrows form sets. Foundational size issues rarely affect operational models because a diagnostic model normally covers a finite execution slice. More important is **scope**: which system version, time window, tenants, and artifact types are included? A finite category with an unstated scope can still be misleading.

B.2 Diagrams and commutativity

By a **diagram** in a category, we mean a collection of objects and morphisms arranged according to another category called its shape. Informally, it is a typed picture of routes.

The square has four typed arrows:

$$f: A \to B, \qquad g: A \to C,$$

$$k: B \to D, \qquad h: C \to D.$$

commutes when

$$k \circ f = h \circ g.$$

The equality is about the two composites from A to D. It is not a claim that f and g are interchangeable. Operationally, one route might normalize a runtime event and then export it; the other might export first and normalize in the telemetry schema. A failed equality is evidence of a version, implementation, loss, or modeling discrepancy.

A triangle, pentagon, or larger diagram works the same way: declared parallel paths should agree. Large diagrams are best decomposed into small cells, and a failed large route can then be localized to the earliest or smallest failed cell.

B.3 Functors

A **functor** $F: \mathcal{C} \to \mathcal{D}$ maps:

- every object A of $\mathcal{C}$ to an object $F(A)$ of $\mathcal{D}$;
- every morphism $f: A \to B$ to a morphism $F(f): F(A) \to F(B)$;

while preserving identities and composition:

$$F(\mathrm{id}_A) = \mathrm{id}_{F(A)},$$

$$F(g \circ f) = F(g) \circ F(f).$$

An instrumentation pipeline is a functor only when its mapping is defined at both object and arrow levels and these laws hold. “It turns runtime data into logs” is not yet enough.

Faithful, full, and essentially surjective

We call a functor **faithful** if distinct arrows between any fixed pair of objects remain distinct. In diagnostics, lack of faithfulness means two relevant transitions look the same after observation.

We call it **full** if every arrow in the target between mapped objects comes from an arrow in the source. A telemetry model may include apparent transitions that have no corresponding runtime relation, so fullness should not be assumed.

We call it **essentially surjective on objects** if every target object is isomorphic to an object in the image. These terms describe precise mathematical properties, and they are useful diagnostic metaphors only after the categories and mappings are formalized.

Contravariant functors

A contravariant functor reverses arrows. Formally, it is a functor $P: \mathcal{C}^{op} \to \mathcal{D}$, where $\mathcal{C}^{op}$ has all arrows of $\mathcal{C}$ reversed. Restricting data from a larger scope to a smaller scope is the standard intuition: an inclusion $U \subseteq V$ induces a restriction $P(V) \to P(U)$.

B.4 Natural transformations

Let $F, G: \mathcal{C} \to \mathcal{D}$ be functors. A **natural transformation** $\eta: F \Rightarrow G$ gives, for every object A of $\mathcal{C}$, a component

$$\eta_A: F(A) \to G(A)$$

such that for every $f: A \to B$ the naturality square commutes:

$$G(f) \circ \eta_A = \eta_B \circ F(f).$$

The components cannot be arbitrary per-object conversions: they must work uniformly with the source structure, which is why naturality is powerful for cross-view consistency. It compares two complete ways of observing a system, not merely two fields in one row.

A **natural isomorphism** has an invertible component at every object. It says two functors are the same up to coherent isomorphism. Two telemetry encodings may be naturally isomorphic even if their record layouts differ, provided the conversions are reversible and natural.

B.5 Universal properties

Category theory often defines an object by how every other object maps to or from it. Such a definition is a **universal property**. It determines the object uniquely up to a unique isomorphism, not by its internal storage layout.

Universal properties are engineering specifications. They identify the exact mediating map a candidate construction must provide and the uniqueness condition it must satisfy.

Terminal and initial objects

A **terminal object** 1 has exactly one arrow $A \to 1$ from every object A. In **Set**, any one-element set is terminal. An **initial object** 0 has exactly one arrow $0 \to A$ to every object; the empty set is initial in **Set**.

A generic "unknown" record is not terminal merely because every source can be mapped to it. Uniqueness must hold in the declared category, and the modeling meaning still matters.

Products and coproducts

A product $A \times B$ comes with projections

$$\pi_A : A \times B \to A, \qquad \pi_B : A \times B \to B.$$

For any X with arrows $f : X \to A$ and $g : X \to B$, there is a unique arrow $\langle f, g \rangle : X \to A \times B$ making both projection triangles commute.

A coproduct $A + B$ reverses the direction. It has injections $A \to A + B$ and $B \to A + B$; any pair of arrows out of A and B mediates uniquely through the coproduct. Tagged unions are a programming intuition.

Pullbacks and pushouts

Given $f : A \to C$ and $g : B \to C$, their **pullback** $A \times_C B$ consists, in **Set**, of pairs (a, b) whose images agree:

$$A \times_C B = \{(a, b) \mid f(a) = g(b)\}.$$

The pullback formalizes a join over a declared shared meaning, but it does not prove causality, identity correctness, or uniqueness of the key.

A **pushout** glues A and B along a shared source C. In software modeling it can express assembly across an interface, but conflicts and quotienting must be understood. "Merge" alone does not imply a pushout.

Equalizers and coequalizers

For parallel arrows $f, g : A \to B$, an **equalizer** selects the part of A on which f and g agree. In diagnostics, we can use it to identify observations that pass a consistency check. A **coequalizer** forces the two routes to become equal in a quotient; it intentionally forgets distinctions and therefore needs an explicit loss interpretation.

B.6 Limits and colimits

A **limit** generalizes products, pullbacks, and equalizers. Given a diagram $D : J \to \mathcal{C}$, a cone from X supplies compatible arrows from X to every object in the diagram. The limit is a universal cone: every other cone factors uniquely through it.

In evidence integration, the useful intuition is a maximally general compatible family. But a dashboard that merely aggregates several sources is not automatically a limit. The diagram, compatibility equations, and universal factorization must be stated.

A **colimit** reverses the arrows and generalizes coproducts, pushouts, and coequalizers. It assembles pieces while imposing declared identifications, and it may be useful for building an incident narrative from local fragments. The construction cannot decide whether the imposed identifications are semantically correct.

Limits and colimits are unique up to unique isomorphism. Two implementations may store the result differently yet realize the same universal property.

B.7 Adjunctions

An **adjunction** $F \dashv G$ between categories $\mathcal{C}$ and $\mathcal{D}$ is a natural correspondence

$$\mathrm{Hom}_{\mathcal{D}}(F(C), D) \cong \mathrm{Hom}_{\mathcal{C}}\big(C, G(D)\big).$$

It says that maps out of $F(C)$ correspond coherently to maps out of C into $G(D)$. The classic example is the free-category functor from directed graphs to categories, left adjoint to the functor that forgets composition and keeps the underlying graph.

An adjunction can also be described by a **unit** $\eta\colon \mathrm{Id}_{\mathcal{C}} \Rightarrow GF$ and **counit** $\varepsilon\colon FG \Rightarrow \mathrm{Id}_{\mathcal{D}}$ satisfying triangle identities. A pair of operations is not adjoint merely because one feels like filtering and the other like expansion, or one goes "forward" while the other goes "backward." The hom-set correspondence or unit/counit laws must be demonstrated.

Attribute maps: three adjoints from subsets

Let T be a set of trace messages, let V be a set of attribute values, and let $A\colon T \to V$ assign exactly one value to each message. The **powerset** $\mathcal{P}(T)$ is the set of all subsets of T; we order these subsets by inclusion. The same construction gives $\mathcal{P}(V)$. From this single attribute function, define three operations:

$\exists_A(S) = \{A(m)\colon m \in S\}$, the values that occur in S;
$A^*(X) = \{m \in T\colon A(m) \in X\}$, the messages whose values lie in X;
$\forall_A(S) = \{v \in V\colon A^{-1}(\{v\}) \subseteq S\}$, the values whose entire fibers lie in S.

The first adjunction is only a two-line set calculation. Saying that every value reached from S lies in X is the same as saying that every message of S maps into X:

$$\exists_A(S) \subseteq X \;\Leftrightarrow\; (m \in S \Rightarrow A(m) \in X) \;\Leftrightarrow\; S \subseteq A^*(X).$$

For the second, $A^*(X) \subseteq S$ says that every message whose value belongs to X is already in S. Group those messages by value:

$$A^*(X) \subseteq S \;\Leftrightarrow\; (v \in X \Rightarrow A^{-1}(\{v\}) \subseteq S) \;\Leftrightarrow\; X \subseteq \forall_A(S).$$

A subset order is a category with one arrow $S \to S'$ exactly when $S \subseteq S'$. The two equivalences are therefore hom-set correspondences, and we have $\exists_A \dashv A^* \dashv \forall_A$ (Davey and Priestley 2002). All three maps are monotone. A value absent from the trace has an empty fiber, so it belongs to $\forall_A(S)$ vacuously; in diagnostic work, we either take $V = \mathrm{im}(A)$ or preserve that absence explicitly.

For a concrete trace, suppose A is the operation column. The fiber $A^*(\{\text{read}\})$ is the read activity, the family of all nonempty fibers partitions the messages, and the quotient that identifies equal values is in bijection with $\mathrm{im}(A)$.

This is why one attribute map can support Adjoint Thread, Message Cover, Sheaf of Activities, Quotient Trace, and Adjoint Message. The mathematics is standard; choosing the column and assigning those diagnostic meanings is the proposal. If the column is partial or multivalued, first add a typed missing value or use a relation—the three-adjoint chain is not automatic.

Galois connections: a finite trace calculation

A **Galois connection** between ordered sets P and Q is a pair of monotone maps $F: P \to Q$ and $G: Q \to P$ such that $F(p) \leq q$ exactly when $p \leq G(q)$. We now verify the construction used for Galois Trace in Chapter 9 without assuming calculus or abstract algebra.

Let T_1 and T_2 be nonempty finite traces. Order each by calibrated timestamp and use a stable within-trace sequence number to break ties. Add a new least element $\bot$ before T_1 and a new greatest element $\top$ after T_2. For $a \in T_1$, $F(a)$ is the first T_2 message at or after $t(a)$, or $\top$ if none exists; set $F(\bot) = \min T_2$. For $b \in T_2$, $G(b)$ is the last T_1 message at or before $t(b)$, or $\bot$ if none exists; set $G(\top) = \max T_1$.

For ordinary messages a and b, the definition of "first at or after" gives $F(a) \leq b$ exactly when $t(a) \leq t(b)$. The definition of "last at or before" gives $t(a) \leq t(b)$ exactly when $a \leq G(b)$. For $p = \bot$, both sides are true for every $q \in Q$: $F(\bot) = \min T_2 \leq q$ and $\bot \leq G(q)$. For $q = \top$, both sides are true for every $p \in P$: $F(p) \leq \top$ and $p \leq G(\top) = \max T_1$. Hence

$$F(p) \leq q \iff p \leq G(q), \qquad F \dashv G.$$

The adjunction gives $p \leq G\big(F(p)\big)$ and $F\big(G(q)\big) \leq q$. The first composite is monotone, extensive, and idempotent, so $G \circ F$ is a **closure operator**. Idempotence can be checked directly: monotonicity and the two inequalities force $F\Big(G\big(F(p)\big)\Big) = F(p)$, and applying G gives $G\Big(F\Big(G\big(F(p)\big)\Big)\Big) = G\big(F(p)\big)$.

For example, let T_1 have times 1,5,5.5,7 and let T_2 have times 2,6. Then F sends the first three messages to times 2,6,6 and the last to $\top$; G sends 2,6, $\top$ to times 1,5.5,7. The round trip closes time 5 upward to 5.5, because both messages share the next boundary at time 6. A different calibration may give a different closure. Empty traces need a separately declared convention, and causal meaning needs causal arrows; neither is supplied by this order calculation.

B.8 Monads and comonads

A **monad** on $\mathcal{C}$ consists of an endofunctor $T: \mathcal{C} \to \mathcal{C}$, a unit $\eta: \mathrm{Id} \Rightarrow T$, and multiplication $\mu: T^2 \Rightarrow T$. The laws say that adding the context twice and flattening is associative and that the unit behaves as an identity.

Programming monads often represent computations with context: possible failure, state, nondeterminism, or asynchronous effects. We can model a telemetry wrapper as a monad only if its object/arrow mapping and unit/multiplication satisfy the monad laws.

A **comonad** reverses the structural arrows: it has a counit $\varepsilon: W \Rightarrow \mathrm{Id}$ and comultiplication $\delta: W \Rightarrow W^2$. It can model values equipped with inspectable context. "Observability is a comonad" is a research proposal until a concrete category, functor, and laws are supplied.

B.9 Presheaves

A **presheaf** on $\mathcal{C}$ is a functor

$$P: \mathcal{C}^{op} \to \mathbf{Set}.$$

It assigns a set $P(U)$ to each object U and a restriction map $P(V) \to P(U)$ to each arrow $U \to V$. The laws require identity restriction to do nothing and staged restriction to agree with direct restriction.

Let $E(U)$ be the events in a time window U, with event identity and timestamps fixed. We take $P(U)$ to be the set of all subsets of $E(U)$, including the empty subset. An element of $P(U)$ is therefore a possible event collection in U, not an individual event. To restrict a collection to a smaller window, we retain only its events in that window. Restricting twice gives the same collection as restricting directly. For hypotheses over code scopes, we must separately specify a total restriction operation and verify the same laws. In general, the presheaf laws alone do not imply the sheaf gluing laws.

For example, let $E(V) = \{a, b\}$, and suppose only b lies in a smaller window U. The power set of a set means the set of all its subsets. Thus $P(V)$ has four elements: $\emptyset$, $\{a\}$, $\{b\}$, and $\{a, b\}$. The set $P(U)$ has two: $\emptyset$ and $\{b\}$. The symbol ∩ means intersection, retaining only common members.

$$P(U) = \mathcal{P}(E(U)), \quad \rho_U^V(S) = S \cap E(U).$$

Restriction sends those four collections respectively to $\emptyset$, $\emptyset$, $\{b\}$, and $\{b\}$. Every input has an output, including a collection whose events all disappear from the smaller window. If the smaller window contains no events, its power set still contains the empty collection, so restriction remains a total function.

For nested windows $U \subseteq V \subseteq W$, intersecting first with $E(V)$ and then with $E(U)$ is the same as intersecting directly with $E(U)$. Restriction to the original window leaves the collection unchanged. These are the presheaf laws in this model.

B.10 Sheaves

By a **sheaf**, we mean a presheaf satisfying a gluing condition for a chosen notion of cover. In plain language:

1. two global sections that restrict to the same local section on every member of a cover are equal (**separation/locality**);
2. every compatible family of local sections glues to a global section (**existence**); separation makes that global section unique.

The technical setting may be a topological space, a site, or a more specialized base. For diagnostics, we may use the following local scopes: services, time windows, host regions, or evidence partitions. We must define covers and overlaps, not just use "sheaf" as a synonym for distributed data.

Sheaf methods are established for sensor fusion and consistency analysis (Robinson 2017, 2020). Their diagnostic value is precise: they can expose an obstruction to global consistency and localize disagreement to overlaps. They do not automatically supply a root cause.

B.11 Toposes and contextual truth

A **topos** (plural **toposes**) is a category that supports many set-like constructions and an internal logic. We need only two examples. If $\mathcal{B}$ is a small category, then the presheaves on it form the topos

$$\widehat{\mathcal{B}} = [\mathcal{B}^{\mathrm{op}}, \mathbf{Set}].$$

Choose a Grothendieck topology J on $\mathcal{B}$. The pair $(\mathcal{B}, J)$ is called a **site**, and the presheaves satisfying its gluing conditions form the sheaf topos

$$\mathbf{Sh}(\mathcal{B}, J).$$

Despite its name, a Grothendieck topology is not necessarily a collection of open regions from geometry. It is a rule saying which families of arrows count as covers. At school-mathematics level, we can read its laws as three promises:

the whole scope covers itself;
a cover remains a cover when pulled back to a smaller scope;
if every member of a cover is itself adequately covered, the combined family covers the original scope.

The next construction begins with an ordinary subset. For a set E and subset $H \subseteq E$, the characteristic function $\chi_H : E \to \{\text{false}, \text{true}\}$ says whether an element belongs to H. In a topos, a **subobject classifier** Ω plays the role of the truth-value set. Every subobject $H \hookrightarrow E$ has a characteristic map

$$\chi_H : E \to \Omega.$$

In the presheaf topos, an element of $\Omega(U)$ is a **sieve** on U: a collection of arrows $V \to U$ closed under further precomposition. In plain language, it is a collection of narrower scopes closed under further narrowing. For $e \in E(U)$,

$$\chi_{H,U}(e) = \{\, f : V \to U \mid E(f)(e) \in H(V) \,\}.$$

Thus the truth value does not merely say yes or no. It records every refinement on which the restricted evidence lies in the hypothesis. In a sheaf topos, the relevant truth values are the J-closed sieves. Mac Lane and Moerdijk give a systematic introduction to these constructions, and Borceux develops their internal logic (Mac Lane and Moerdijk 1994; Borceux 1994).

The truth values in Ω form a **Heyting algebra**, the algebraic setting for intuitionistic logic. We may use "and," "or," implication, and negation, but the law "P or not P" need not hold without evidence that decides P. This is why "not established here" need not equal "established false." It does not turn truth into probability; numerical confidence requires a separate model.

The inclusion $i : \mathbf{Sh}(\mathcal{B}, J) \to \widehat{\mathcal{B}}$ has a left adjoint a, called **sheafification**:

$$a \dashv i.$$

The sheaf aP is universal: every map from P to a sheaf factors uniquely through it. Sheafification repairs gluing relative to J; it does not recover an unrecorded runtime event.

A **geometric morphism** relates toposes by adjoint inverse-image and direct-image functors, with inverse image preserving finite limits. Comparing instrumentation regimes this way remains a research direction.

These definitions are standard mathematics (Mac Lane and Moerdijk 1994; Johnstone 2002). A diagnostic interpretation still requires declared scopes, covers, evidence, restrictions, hypotheses, and possible refutations.

B.12 Coalgebras

For an endofunctor $F: \mathcal{C} \to \mathcal{C}$, an **$F$-coalgebra** is an object X with a structure arrow

$$c: X \to F(X).$$

It describes how a state reveals an observation and possible successor structure. A simple deterministic machine with output set O and input set I may use

$$F(X) = O \times X^I.$$

The coalgebra returns a current observation and an input-indexed next state. Coalgebra is therefore suited to ongoing behavior and observation (Jacobs 2017).

A coalgebra homomorphism preserves behavior. **Bisimulation** relates states that cannot be distinguished by the modeled observations and transitions. If a fault state is bisimilar to a healthy state under the current observation functor, no diagnostic algorithm using only those observations can separate them. Instrumentation must refine the observation structure.

B.13 Monoidal categories and categorical trace

A **monoidal category** provides a way $A \otimes B$ to place objects side by side, a unit object I, and coherent associativity and unit isomorphisms. The tensor may mean parallel composition, resource combination, or interface juxtaposition; it is not automatically the Cartesian product.

A **traced monoidal category** has an operation for feeding part of an output back as part of an input, subject to trace axioms (Joyal, Street, and Verity 1996). This categorical trace is an abstract feedback operation. It is not the same as a software execution trace, though a formal relationship might be constructed in a particular model.

B.14 Operads

An **operad** describes many-input, one-output operations and how they substitute into one another. A colored operad assigns types, called colors, to inputs and outputs. A diagnostic operation might consume a dump, symbols, and configuration and produce a set of suspected allocations.

Operads are useful when the important composition is substitution of typed workflows rather than simple one-output-to-one-input chaining. Work on operads for complex system design demonstrates the broader compositional setting (Foley et al. 2021). A workflow engine becomes an operadic model only after operations, colors, identities, symmetry, and substitution laws are defined.

B.15 2-categories

A **2-category** has:

- objects;
- 1-morphisms between objects;
- 2-morphisms between parallel 1-morphisms.

The 2-morphisms compose vertically, and 1-morphisms plus their 2-morphisms compose horizontally. Compatibility is expressed by an interchange law.

In diagnostics, we may treat two execution paths as 1-morphisms and an explanation, rewrite, refinement, or evidence-backed comparison between them may be a 2-morphism. The interpretation must support both compositions. Merely annotating an edge does not create a 2-category.

Bicategories weaken strict equality of associativity and units to coherent isomorphisms. They may be more realistic when system composition is associative only up to renaming, buffering, or protocol mediation.

B.16 Yoneda's viewpoint

The **Yoneda lemma** says, roughly, that an object is completely characterized by all morphisms into it or out of it. Formally, natural transformations from the representable functor $\mathrm{Hom}_{\mathcal{C}}(-, A)$ to a presheaf P correspond to elements of $P(A)$.

The diagnostic intuition is relational: a component is understood not only by its internal fields but by every permitted interaction with other components. This supports interface-first analysis, but it does not mean that finite telemetry actually observes every relevant morphism.

B.17 Kan extensions

By a **Kan extension**, we mean a universal way to extend or transport a functor along another functor. Many categorical constructions, including adjoints in a suitable sense, can be expressed through Kan extensions.

For observability, a Kan extension may model the best systematic approximation of a view after changing schema or granularity. This book does not depend on the machinery. It is a research direction for principled projection, aggregation, and reconstruction across evolving telemetry categories.

B.18 Enriched, lax, and partial variants

Exact equality is often too strict for production data. An **enriched category** may assign a distance, cost, probability, or order rather than a plain set of morphisms. A **lax functor** preserves composition by a comparison morphism rather than exact equality. A **partial** categorical structure explicitly represents undefined mappings.

These variants can model sampling, timestamp tolerance, uncertainty, and loss. They are not permission to accept arbitrary mismatch. The comparison maps, order, metric, or partiality rules are additional structure that must be specified and tested.

B.19 Standard mathematics, diagnostic construction, and guiding analogy

We use three claim statuses:

Standard mathematics: A conventional categorical definition, theorem, or construction used with its established meaning and hypotheses.

Diagnostic construction: Objects, morphisms, laws, scope, and relevant comparison rules are specified for a software-diagnostic setting and can be checked against evidence.

Guiding analogy: The mapping is explicit and useful, but some laws or structures are not yet established. It may guide engineering while remaining provisional. A bare metaphor, with no mapping or law, falls outside these three claim statuses.

This three-status discipline is central to the book: it lets us distinguish structural claims from visual resemblance when using categorical language.

B.20 Validity checklist

Before calling a model categorical, we ask:

1. What are the objects, with exact scope and type?
2. What are the morphisms, with exact source and target?
3. Which pairs of morphisms compose?
4. What is the identity at every object?
5. Why is composition associative?
6. Which path equations are axioms, specifications, or empirical hypotheses?
7. If there is a functor, what does it do to arrows as well as objects?
8. Are identity and composition preservation tested?
9. If there is a natural transformation, what are its components and naturality squares?
10. If a universal construction is claimed, where is the mediating arrow and why is it unique?
11. If an adjunction, monad, sheaf, topos, operad, or 2-category is claimed, which defining laws have been checked?
12. What information is lost, hidden, sampled, approximated, or undefined?
13. What equality, equivalence, distance, or tolerance is used?
14. What evidence would falsify the model?

If these questions cannot yet be answered, label the construction a guiding analogy or proposal and retain the intuition without overstating the mathematics.

B.21 Recovering a diagnostic answer from an observation

Consider a finite set X of modeled execution cases. A function o assigns each case its retained observation. Write $Y_0 = o(X)$ for the observations actually possible in this model. A second function d assigns each case its required diagnostic answer in a set D. We ask whether a function φ can recover that answer from the observation alone. “There exists” means that at least one function with the stated property can be defined.

$$o: X \to Y_0, \quad d: X \to D, \quad \varphi: Y_0 \to D.$$

The answer can be recovered from the observation exactly when such a function satisfies:

$$d = \varphi \circ o.$$

This factorization exists if and only if every two cases with the same observation have the same required answer:

$$o(x_1) = o(x_2) \Rightarrow d(x_1) = d(x_2).$$

To prove the forward direction, substitute $\varphi(o(x))$ for $d(x)$. Equal inputs to φ give equal outputs. Conversely, assign to each observed value y the common diagnostic answer of the cases with observation y. The condition makes this assignment unambiguous. Every y in Y_0 has at least one such case. We make no arbitrary assignment for impossible observations outside Y_0.

The zero-byte records in Chapter 31 give a two-case counterexample. Let x_1 have status 200 and x_2 have status 304. A byte-only observation gives $o(x_1) = o(x_2) = 0$, but the required status families are 2xx and 3xx. No function of the byte count alone can give both answers for the same observed value. Retaining the status family instead satisfies the condition for this classification.

This is an elementary result about functions and the distinctions they preserve (see the image and quotient discussion in B.7; Riehl 2016). Its use in Chapter 18 is diagnostic sufficiency relative to a declared goal. It is not a new categorical theorem or a statement of statistical sufficiency, and agreement in a finite test sample does not establish the condition outside that sample.

B.22 Distances and approximate pasting

A distance measures how far two values are apart. For numbers, use the absolute difference: the distance from 5 to 8 is 3. The triangle inequality says that the distance from a to c is no greater than the distance from a to b plus that from b to c. A metric also has nonnegative distances, symmetry, and distance zero exactly between equal values. A pseudometric permits distinct values at distance zero; zero residual then means only the declared zero-distance equivalence.

Use the arrow names of Figure 5A: $p: L \to R$, $e: L \to F$, $q: R \to T$, $t: F \to T$, $n: R \to N$, $s: N \to D$, and $c: T \to D$. Here we use a separate timing model for these types, not the original status-valued example. For every admitted input x, suppose the left and right residuals have nonnegative bounds ε_1 and ε_2:

$$d_T(q(p(x)), t(e(x))) \leq \varepsilon_1,$$
$$d_D(s(n(p(x))), c(q(p(x)))) \leq \varepsilon_2.$$

The subscripts tell us which value space supplies the distance. Suppose, in addition, that the downstream map c amplifies distances by no more than a nonnegative number K throughout the relevant domain:

$$d_D(c(u), c(v)) \leq K\, d_T(u, v).$$

This is a Lipschitz bound: a stated upper limit on amplification that needs justification for the chosen map and domain. Insert the intermediate result $c(q(p(x)))$ between the two outer results. The triangle inequality bounds the outer distance by the sum of the distance to that intermediate result and the distance from it to the second result. The first distance is at most ε_2. The second is at most K times the left residual. Therefore:

$$d_D(s(n(p(x))), c(t(e(x)))) \leq \varepsilon_2 + K\,\varepsilon_1.$$

If T uses seconds and D uses milliseconds, the conversion $c(u) = 1000u$ has $K = 1000$ with the corresponding units. For $\varepsilon_1 = 0.002$ seconds and $\varepsilon_2 = 1$ millisecond, the outer bound is 3 milliseconds. The calculation is conditional on both local bounds and the amplification bound. Testing those bounds on selected inputs alone does not establish them on an unrestricted domain.

A yes/no threshold shows why the amplification assumption matters. Let the threshold be 10. Durations 9.999 and 10.001 differ by 0.002 but have opposite classifications. Give equal classifications distance 0 and different classifications distance 1. We can choose two durations arbitrarily close to the threshold on opposite sides: their numeric distance becomes arbitrarily small while their classification distance remains 1. No finite global K controls this classifier on that unrestricted domain.

The connection between distances and categorical composition has a substantial history (Lawvere 1973), as do quantitative consistency measures for sheaf assignments (Robinson 2020). The calculation here is an elementary two-square error bound for the declared diagnostic transformations.

B.23 Sound approximation of possible states

A sound overapproximation includes every concrete state that may occur under the model, possibly with additional states. “Sound” describes this inclusion guarantee. “Overapproximation” means that the retained set can contain possibilities that no actual execution realizes.

Suppose the retained information says a duration lies in the interval $[8, 12]$. This notation includes all values from 8 through 12, including both endpoints. The interval does not exclude a violation above 10, but it does not prove that one occurred. If the sound interval were $[8, 9]$, that violation would be excluded. This conclusion relies on the interval actually containing every duration consistent with the admitted evidence.

This example uses only interval membership and set inclusion. A full abstract interpreter additionally needs sound abstract operations, not merely a choice of summary (Cousot and Cousot 1977). The categorical vocabulary does not supply those obligations automatically.

B.24 Correlation witnesses and span composition

A span of sets consists of a witness set and two functions that give the endpoints of each witness. Let S contain source-value instances, A contain fully scoped storage locations, and D contain dump artifacts. Write W for source-to-location witnesses and Z for location-to-dump witnesses:

$$S \leftarrow W \rightarrow A, \quad A \leftarrow Z \rightarrow D.$$

The composite witness set consists of pairs (w, z) whose A endpoints agree. This is the pullback of W and Z over A (B.5). Its endpoint functions select the source endpoint of w and the dump endpoint of z. A categorical span is not a distributed-tracing span.

For a finite example, let w_1 connect source value u to location a, and let w_2 connect the same value to location b. Let z_1 connect location a to dump d, and let z_2 connect location b to the same dump. The location keys include address space, lifetime, and capture context, as specified in Chapter 7.

There are two composite witnesses, (w_1, z_1) and (w_2, z_2). Both connect u with d. Keeping only the endpoint relation produces the single pair (u, d) and discards the location path. The relation answers whether some witness exists; the span also retains which witnesses exist. Two witnesses do not by themselves establish independent confirmation or causality.

The identity span uses X itself as its witness set and identity functions at both ends. Repeated span composition groups compatible witness tuples in different ways. The results agree up to the canonical bijection that removes the grouping parentheses: $((w, z), v)$ corresponds to $(w, (z, v))$. This is a one-to-one matching of the same three witnesses, not literal equality of the differently grouped pairs.

One may use isomorphism classes of spans for an ordinary category. Here two spans are isomorphic when a bijection of their witness sets preserves both endpoint functions. Alternatively, one may retain actual spans and the comparison isomorphisms in a bicategorical treatment (compare B.15).

For discrete endpoint categories, assign to each pair (s, a) the set of witnesses with those endpoints. This is a set-valued profunctor in the convention:

$$S^{\mathrm{op}} \times A \to \mathbf{Set}.$$

Here S^{op} reverses the arrows of S, and the product category pairs source and target objects. "Discrete" means that only identity arrows are present, so no additional transport maps need to be chosen. In this case, a table whose cells are sets of witnesses gives the whole construction. If the endpoint categories have nonidentity arrows, we must also give maps that transport the witness sets along those arrows, contravariantly in the first input and covariantly in the second, preserving identities and composition (Fong and Spivak 2019; Riehl 2016).

B.25 The equation behind a renaming test

Let A map input artifacts to analysis reports. Let M rename identifiers in artifacts, and let N apply the corresponding renaming to report labels. The proposed comparison in Chapter 28 is:

$$A \circ M = N \circ A.$$

The left route renames an artifact and then analyzes it. The right route analyzes the artifact and then renames its report. Both begin with the same artifact type and end in the same report type. If reports contain only an invariant count, N is the identity. For example, a consistent one-to-one renaming of request identifiers preserves the number of complete request pairs when pairing does not depend on the identifiers' literal contents.

This equation expresses a metamorphic test under the renaming and analyzer assumptions in Chapter 28 (Chen et al. 2018). A family of such equations becomes categorical structure only after its objects, arrows, compositions, and preservation laws are specified. A tested square alone is not a natural transformation, and selected passing tests are not a proof for all admitted inputs.

Appendix C — Diagnostic invariant cards

These cards are reusable starting points, not universal truths. Each must be bound to a system version, scope, equality policy, and evidence contract. The common structure is:

- **arena:** the categories in which the objects and paths live;
- **objects and arrows:** the typed model;
- **expected diagram:** the routes that should agree;
- **residual:** how disagreement is measured;
- **failure interpretations:** competing explanations, including observation failure;
- **tests:** normal, boundary, negative, and drift cases;
- **consequence:** the action a violation may justify.

C.1 Status agreement

Intent: Detect contradiction among protocol outcome, span status, log classification, and user-visible result.

Arena: A category of completed operations and a category of normalized outcomes.

Objects and arrows: An operation completion maps through separate functors to HTTP response, trace status, log severity, and business outcome. Each view maps to a shared outcome vocabulary such as {success, expected-rejection, transient-failure, permanent-failure, unknown}.

Expected diagram: Every normalized route from the same completion should produce an allowed compatible outcome. Exact equality may be too strong: HTTP 404 can be an expected rejection for one endpoint and a failure for another, so the normalization includes operation semantics.

Residual: A compatibility matrix entry, optionally weighted by severity and evidence confidence.

Failure interpretations: Application defect, incorrect span status, stale log classifier, proxy rewrite, missing client cancellation, or wrong model of the endpoint.

Tests: Successful request, handled validation failure, unhandled exception, timeout, cancellation, and status-schema version change.

C.2 Context-propagation square

Intent: Detect a boundary that breaks trace or request identity.

Arena: Execution-boundary events and context records.

Objects and arrows: A send event s transitions to receive event r. An extraction functor maps each event to its context, and the protocol carrier maps outbound context to inbound context.

Expected diagram: Extract after receive should agree with transport-and-normalize after extracting at send.

Residual: Field-level mismatch over trace identifier, parent identifier, sampling flag, baggage, tenant, and version. Missing is typed as not-created, not-injected, not-carried, not-extracted, not-exported, or unknown when distinguishable.

Failure interpretations: Missing injection, unsupported carrier, intermediary stripping, asynchronous hand-off, new root by policy, corrupted header, sampling, or exporter loss.

Tests: Direct call, queue, retry, fan-out, fan-in, link-based work, malformed context, and mixed library versions.

C.3 Symbol-resolution naturality

Intent: Ensure that stack evidence resolves coherently across address relocation and module-version mappings.

Arena: Address spaces, module identities, symbols, and source locations.

Objects and arrows: One functor maps a captured address through runtime relocation into a canonical module-relative address. Another maps the raw address using the debugger's loaded-module view. Components translate both into symbol/source representation.

Expected diagram: Resolve after canonical relocation should agree with translate after raw-view resolution for the same module build identifier.

Residual: Symbol name, displacement, file, line, inline chain, and confidence mismatch.

Failure interpretations: Wrong symbols, stale module list, address-space mismatch, stripped binary, hot patch, just-in-time code, corrupted stack, or a legitimately non-symbolic frame.

Tests: Exact symbols, public symbols, mismatched build, relocated module, unloaded module, generated code, and repeated address in different processes.

C.4 Process-identity pullback

Intent: Join metrics, logs, traces, and dumps without confusing reused process identifiers.

Arena: Artifact records over a canonical process-lifetime object.

Objects and arrows: Each artifact maps to a key containing host or workload identity, process identifier, process-start time or boot identity, container/pod generation, and observation interval.

Expected diagram: The integrated evidence object is the pullback of artifact views over the canonical process-lifetime key, subject to interval compatibility.

Residual: Key disagreement, ambiguous multiplicity, or incompatible time windows.

Failure interpretations: PID reuse (the trace reference's Glued Activity; Vostokov 2026, 146), host-name collision, clock error, stale inventory, container restart, dump captured outside the metric window, or incomplete key.

Tests: Restart with PID reuse, failover, cloned host name, rolling deployment, clock shift, and artifacts with absent start time.

C.5 Schema-migration square

Intent: Detect observability drift during telemetry schema evolution.

Arena: Old and new runtime event schemas and their old and new telemetry schemas.

Objects and arrows: Runtime migration m_R changes the source event representation. Telemetry migration m_T changes the exported representation. Observation functors O_{old} and O_{new} connect the versions.

Expected diagram: For supported events,

$$O_{new} \circ m_R \approx m_T \circ O_{old}.$$

Approximation is governed by a declared migration policy: renamed fields, split values, deprecated attributes, and explicitly lost distinctions.

Residual: Semantic field difference after canonicalization, classified as intended migration, unimplemented mapping, silent loss, or undefined.

Failure interpretations: Version skew, producer migration without consumer migration, changed unit, changed enum meaning, default-value collision, or stale query.

Tests: Golden fixtures for every semantic variant, dual-write comparison, mixed-version traffic, absent optional fields, and rollback.

C.6 Restriction stability

Intent: Ensure a diagnostic view behaves consistently when the time, service, tenant, or code scope is narrowed.

Arena: A presheaf of observations or hypotheses on a category of scopes.

Objects and arrows: Each scope U has a set $P(U)$. An inclusion $U \subseteq V$ induces restriction $\rho_U^V : P(V) \to P(U)$.

Expected diagram: Identity restriction changes nothing, and staged restriction equals direct restriction:

$$\rho_U^V \circ \rho_V^W = \rho_U^W.$$

Residual: Difference between direct and staged views after a declared ordering and deduplication policy.

Failure interpretations: Scope-dependent normalization, stateful query, unstable pagination, late arrival, cache skew, implicit tenant filter, or time-zone boundary error.

Tests: Nested windows, empty scope, boundary event, repeated query, late data, and daylight-saving transition where relevant.

C.7 Overlap-consistency and gluing

Intent: Determine whether local incident reconstructions can form one global account.

Arena: A presheaf or sheaf over service/time regions with declared covers.

Objects and arrows: Each region has a local section: events, inferred states, or causal fragments. Restriction maps expose the overlap between regions.

Expected diagram: Local sections agree on every overlap. If the model is a sheaf and a compatible family exists, it glues uniquely to a global section.

Residual: Overlap disagreement by identity, order, payload, or inferred state, and a consistency radius may quantify approximate agreement.

Failure interpretations: Clock skew, duplicate identity, inconsistent replay, missing region, incompatible model versions, unauthorized redaction, or truly divergent local state.

Tests: Consistent partition, one contradictory overlap, absent overlap, triple overlap, delayed data, and two globally ambiguous assemblies.

C.8 Agent-claim accountability

Intent: Ensure that an AI or automation agent's claim is supported by its retrieved evidence and permitted actions.

Arena: Intent states, tool-call traces, world observations, provenance records, and claims.

Objects and arrows: One path interprets user intent, selects a tool, observes a result, and produces a claim. Another reconstructs the claim from retained provenance through a verifier.

Expected diagram: The produced claim and the verified claim agree at the declared semantic level; every decisive statement has a provenance path; actions stay within an authority subcategory.

Residual: Unsupported proposition, source mismatch, stale observation, omitted tool failure, policy violation, or semantic disagreement.

Failure interpretations: Hallucination, tool schema mismatch, data changed after observation, unrecorded transformation, prompt ambiguity, verifier defect, or insufficient evidence.

Tests: Successful lookup, empty result, tool error, conflicting sources, revoked authority, changing world state, and deliberately adversarial retrieved text.

C.9 From card to executable check

To operationalize any card:

1. bind abstract objects to schemas and versions;
2. implement arrow adapters with provenance;
3. choose an equality or residual algebra;
4. encode assumptions and typed missingness;
5. compile both paths independently;
6. retain path outputs, not only a pass/fail bit;
7. localize the smallest failed cell;
8. classify model, system, instrumentation, and data-quality explanations;

9. add a regression fixture after adjudication.

A card is valuable when its failure changes an engineering decision. If violations are unactionable, refine the scope, model, residual, or consequence.

Appendix D — Where category theory is established in computing

Category theory is well established in software semantics and architecture, model synchronization, categorical databases, coalgebraic behavior, compositional systems, sheaf-based consistency, and topos-based logic. In this appendix, we mark that established territory so that the book's contribution is neither isolated from prior work nor overstated.

"Established" here means that a field has definitions, a sustained research literature, and worked formal or computational applications. It does not mean that every industrial tool is categorical, that adoption is universal, or that the diagnostic constructions in this book follow automatically.

D.1 Software semantics and architecture

Category theory entered theoretical computer science through semantics, type theory, algebraic specification, and the study of compositional structure. Pierce's computer-science introduction presents categories as a unifying language for models and programming concepts (Pierce 1991). Goguen's categorical manifesto argues for structure-preserving mappings and composition as a broad foundation for computing (Goguen 1991).

Fiadeiro develops categorical techniques for software engineering across specification, coordination, architectures, and formal development (Fiadeiro 2005). The central architectural benefit is compositionality: describe components and interfaces so that assembling local pieces has a semantics determined by their declared connections. Categorical diagrams express constraints among architectural views; universal constructions express canonical composition or coordination.

Diagnostic inheritance: In this book, we inherit the discipline of typed components, interfaces, and compositional semantics. We change the primary subject from planned software structure to *runtime evidence about realized structure*. A trace or dump is not the architecture itself. The new work is to specify observation functors from execution into evidence and test whether architectural relations survive.

D.2 Model transformation and synchronization

Model-driven engineering must relate requirements, design models, code models, database schemas, and generated artifacts. Category theory supplies graphs, categories, functors, spans, pullbacks, pushouts, and algebraic structures for transformations and consistency. Diskin and Maibaum describe a trajectory from categorical semantics to practical design patterns for model transformation, merging, and synchronization (Diskin and Maibaum 2012).

The mature problem is not just converting a file format. It is maintaining correspondence as models evolve, accounting for partiality and amendment, and expressing when round trips or concurrent updates are consistent. Bidirectional transformations and lenses form a related literature, though not every lens is presented categorically.

Diagnostic inheritance: Observability pipelines are model-transformation pipelines: runtime events become signal schemas, normalized records, queries, and claims. In this book, we apply synchronization thinking to live and

post-mortem evidence. Our versioned observation square tests whether system evolution and evidence evolution remain compatible, and our naturality residual localizes where two views cease to transform coherently.

D.3 Categorical databases and knowledge representation

Categorical databases model a schema as a category and an instance as a functor from that schema to **Set**. Functorial data migration provides principled operations for transporting instances along schema mappings (Spivak 2012). Related work expresses database queries and constraints categorically (Spivak 2014b). Ologs use objects and arrows to represent typed concepts and meaningful relations, turning a knowledge model into a diagram whose paths may carry declared equivalences (Spivak and Kent 2012).

This body of work is directly relevant to telemetry because telemetry schemas are small databases with rapidly evolving semantics. For a span schema, we have entity types, attributes, and relations; a log normalizer migrates one instance to another; an ontology aligns heterogeneous names.

Diagnostic inheritance: We use categorical data ideas for evidence joins, schema migrations, and world models. Our additional concern is evidential provenance and loss: a migration may be valid as a data operation while becoming diagnostically unsafe because it collapses the distinction needed to separate two fault states.

D.4 Coalgebraic behavior

Coalgebra studies state-based systems through observations and transitions. An F-coalgebra $c: X \to F(X)$ describes how a state unfolds into observable structure and successor behavior. Bisimulation identifies states with indistinguishable behavior under the chosen functor. Jacobs provides a systematic account connecting coalgebra to automata, transition systems, probabilistic behavior, and state observation (Jacobs 2017).

Coalgebra is therefore established precisely where diagnostics asks "what can this observer distinguish about an evolving system?" It formalizes behavioral equivalence and supports coinductive reasoning over potentially unbounded execution.

Diagnostic inheritance: In this book, we use bisimulation as a limit statement about observability. If healthy and faulty states are behaviorally indistinguishable under the available observation functor, no downstream classifier can reliably separate them without new evidence or assumptions. The engineering contribution is a method for turning that theoretical blindness into an instrumentation-separation question.

D.5 Compositional and open systems

Applied category theory has developed substantial methods for systems assembled from parts with interfaces. Fong and Spivak present compositional modeling through categories, functors, monoidal structures, wiring diagrams, and related constructions (Fong and Spivak 2019). Structured cospans provide a formal language for composing open systems along interfaces (Baez and Courser 2020). Operadic approaches encode typed many-input composition for complex-system design and analysis (Foley et al. 2021).

Monoidal categories model parallel or side-by-side combination, while traced monoidal categories model feedback abstractly (Joyal, Street, and Verity 1996). These are established mathematical structures, not metaphors for

"several things happen" or for software trace files. Each application must define the tensor, unit, feedback, and laws.

Diagnostic inheritance: Distributed incidents cross service, process, protocol, and organizational interfaces. In this book, we import interface-aware composition into evidence analysis: boundary invariants, context-propagation squares, typed diagnostic operads, and pattern composition. We add failure localization to the compositional picture: when an end-to-end diagram fails, decompose it to the smallest boundary whose preservation contract fails.

D.6 Sheaves and local-to-global consistency

Sheaf theory formalizes compatible local data and its relation to global data. With a presheaf, we have restriction from larger scopes to smaller scopes. A sheaf adds unique gluing of compatible local sections. Robinson shows that sheaves provide a canonical structure for heterogeneous sensor integration and develops quantitative consistency measures using pseudometric sheaves (Robinson 2017, 2020). Schultz, Spivak, and Vasilakopoulou connect sheaves with dynamical systems (Schultz, Spivak, and Vasilakopoulou 2020).

This is established work in data integration and compositional modeling, and it is especially important because a sheaf does not merely aggregate. It records overlap, compatibility, and the obstruction that prevents a global assignment.

Diagnostic inheritance: Services, hosts, time windows, teams, and telemetry modalities produce local incident accounts. We apply restriction and gluing tests to those accounts, drawing also on the source patterns Sheaf of Activities and Trace Presheaf. Our practical extension is the evidence-consistency map: preserve local residuals and use them to locate where a global incident narrative becomes impossible.

D.7 Toposes, internal logic, and software semantics

Topos theory is established mathematics. Presheaf categories and categories of sheaves on sites are Grothendieck toposes; their subobject classifiers support an internal intuitionistic logic. Mac Lane and Moerdijk provide the standard introductory bridge from sheaves to logic, Borceux gives a detailed account of internal logic, and Johnstone supplies the advanced compendium (Mac Lane and Moerdijk 1994; Borceux 1994; Johnstone 2002).

Toposes are also established in computer science. Birkedal and colleagues use the topos of trees, together with its internal higher-order logic, to model guarded recursion, recursive types, and a language with higher-order store (Birkedal et al. 2012). This is direct software-semantics prior work, not an analogy added by this book.

Sheaf methods have also reached distributed computing. Felber, Hummes Flores, and Rincon Galeana construct task sheaves whose global sections correspond to terminating solutions and whose obstructions express limits on solvability (Felber, Hummes Flores, and Rincon Galeana 2025). This strengthens the prior-work boundary around any local-to-global diagnostic claim.

Diagnostic inheritance: We use the established mathematics to make observational sufficiency and contextual hypothesis support explicit. The scope category, coverage topology, evidence sheaf, hypothesis subobject, and operational comparator are domain constructions. The phrase **diagnostic topos** names their proposed combination; it does not claim that standard topos theory, internal logic, or distributed sheaf methods originate here.

D.8 Adjacent but distinct diagnostic traditions

Distributed tracing is independently mature. X-Trace propagated task metadata across layers (Fonseca et al. 2007); Dapper demonstrated scalable production tracing (Sigelman et al. 2010); Pivot Tracing combined dynamic instrumentation and happened-before joins (Mace, Roelke, and Fonseca 2015); request-flow comparison diagnosed performance changes from structural differences (Sambasivan et al. 2011). Program slicing, delta debugging, and fault localization likewise provide established ways to reduce a failing execution or rank suspicious program elements (Weiser 1984; Zeller and Hildebrandt 2002; Abreu et al. 2009).

These techniques are prior work, not instances retroactively "owned" by category theory. A categorical model may connect them: a slice can restrict an execution category, delta debugging can search for a minimal failed subdiagram, and request-flow comparison can compare path objects. The benefit must come from a clearer invariant, composition rule, or cross-view mapping, not from renaming an established algorithm.

D.9 Boundary of novelty

The literature reviewed above establishes the mathematical tools and many computing applications. The attached source books establish a prior program applying categorical ideas specifically to software diagnostics, trace patterns, APIs, and internals (Vostokov 2024b, 2026, 2024a, 2025). The novelty claimed by this book is the coherent operational synthesis:

- runtime execution, each telemetry modality, and diagnostic claims are separate but related categories;
- observation and normalization are explicit, loss-bearing mappings;
- cross-view consistency is expressed through naturality and shared codomains;
- expected diagrams compile into measurable residuals;
- the Categorical Invariant Violation pattern localizes non-preservation;
- observability drift compares system and evidence evolution;
- source-level, API-level, memory-level, and trace-level representations participate in one provenance chain;
- Pattern Category Theory is refined into typed specifications, detectors, instances, and transformations;
- diagnostic operads make multi-artifact workflows compositional.

a diagnostic site makes sufficient observational coverage compositional, while hypothesis subobjects and Ω preserve contextual support instead of forcing premature binary conclusions.

A targeted search updated on 4 September 2026 found extensive work in every establishment named in this appendix, including toposes in categorical logic and software semantics and sheaves in distributed computation. It found no peer-reviewed monograph organized around this exact runtime-evidence synthesis and no directly comparable published diagnostic-topos program. That is a bounded search result, not proof that no related work exists. The durable novelty test is more practical: whether the framework yields new falsifiable invariants, catches real cross-view defects, and reduces diagnostic search without hiding model error.

D.10 A reading route

For a school-level entry, begin with Lawvere and Schanuel's *Conceptual Mathematics*, the route that also motivated the Windows API source (Lawvere and Schanuel 2009; Vostokov 2024a, 253, 329). Continue with

Leinster or Riehl for a concise or broader categorical foundation (Leinster 2014; Riehl 2016), then Pierce and Fiadeiro for computing, and Fong and Spivak for compositional applied category theory. Select by problem: Spivak for databases, Jacobs for coalgebra, Robinson for sheaves and consistency, Diskin and Maibaum for model synchronization, Baez and Courser for open systems, or Foley and colleagues for operadic system design.

For toposes, continue from the presheaf and sheaf sections of Appendix B to Mac Lane and Moerdijk. Use Borceux for internal logic and Johnstone as an advanced reference. Birkedal and colleagues supply the software-semantics case, while Felber, Hummes Flores, and Rincon Galeana provide nearby distributed-computing prior work.

For more worked sheaf constructions, continue with Rosiak's Sheaf Theory through Examples (Rosiak 2022). Its concrete examples complement the consistency applications in Robinson's work. Treat it as further reading after the elementary restriction and gluing explanations in Appendix B.

Read the four source books alongside those texts. Their value is the diagnostic imagination: they propose categories of artifacts and patterns, presheaf debugging, trace fields and viewpoints, API categories, Observation Spaces, Declarative Memory, and Pattern Category Theory. In this book, we connect that imagination to stricter definitions, preservation contracts, and executable failure tests.

Appendix E — Source concordance and conceptual provenance

In this appendix, we record how the four supplied books contribute to the present synthesis. Page ranges refer to the cited editions. A source concept may be preserved, formalized, narrowed, or reinterpreted, and those changes are described so that provenance is visible.

E.1 *Theoretical Software Diagnostics*, Fourth Edition

The book's **MemD Category** and **Category Theory and Troubleshooting** articles provide early categorical orientations to memory-dump evidence and troubleshooting (Vostokov 2024b, 88–90, 109). In this book, we generalize from individual artifact types to a multi-layer system of execution, evidence, and hypotheses.

Categorical Foundations of Software Diagnostics distinguishes concrete execution artifacts, concrete problem patterns, analysis patterns that can act as functors, and General Analysis Patterns as natural transformations (Vostokov 2024b, 271–72). Chapters 2–5 retain that mapping and add the requirements of arrow typing, identity and composition preservation, explicit loss profiles, and executable commutativity tests.

Software Codiagnostics treats data transformations and cooperating analysis operations as part of diagnostics (Vostokov 2024b, 275–76). In Chapter 10, we turn these transformations into versioned artifact categories and provenance paths; Chapters 28–29 turn their preservation laws into a platform design.

Diagnostic Operads proposes compositional diagnostic operations (Vostokov 2024b, 279–81). Chapter 15 supplies colors, typed operations, substitution laws, algebras, and the boundary with properads and other workflow formalisms.

Debugging and Category Theory motivates debugging as a presheaf and natural transformations among debugging views (Vostokov 2024b, 319–30). Chapters 11–12 distinguish contravariant restriction from reverse execution and separate presheaf restriction from sheaf gluing.

Traces and Logs as 2-categories proposes a higher-categorical account of messages and their relations (Vostokov 2024b, 361–62). Chapter 14 adds parallelism requirements, vertical and horizontal composition, interchange, weak structure, and the distinction between execution traces and categorical trace.

E.2 *Trace, Log, Text, Narrative, Data*, Fifth Edition

Adjoint Message, Adjoint Space, and Adjoint Thread of Activity motivate cross-column correlations, trace–memory views, and alternative activity projections (Vostokov 2026, 43–52). Chapter 9 exhibits the hom-set correspondence carried by a total attribute column, distinguishes that standard adjunction from the diagnostic identification of the patterns, and gives a separate finite Galois connection for Galois Trace (2026, 144).

Sheaf of Activities and Trace Presheaf motivate local trace comparison and trace-to-memory association (Vostokov 2026, 266–67, 330). Chapters 9, 11, and 12 identify attribute fibers as a candidate cover, then keep restriction and gluing laws as obligations to test.

Trace Field treats a mapping from trace messages to another value domain as a functor (Vostokov 2026, 314). Chapters 16 and 19 develop a family of such fields with modality-specific codomains and explicit coupling relations.

Trace Join makes cross-trace selection and matching an analysis pattern (Vostokov 2026, 321). Chapter 7 identifies the shared key or semantic object and distinguishes pullback compatibility from causality.

Trace Viewpoints separates specialist perspectives, while **Visibility Limit** identifies boundaries of available evidence (Vostokov 2026, 344–45, 360). Chapters 18–19 model viewpoints as partial structured views and visibility as a preservation/loss profile.

The reference's **Traces and Logs as 2-categories** article supplies the trace-specific higher-category proposal (Vostokov 2026, 380–81). Chapter 14 gives the laws that a concrete realization must satisfy.

E.3 *Accelerated Windows API for Software Diagnostics*, Second Edition

The **Windows API Formalization** section presents categories, functors, natural transformations, adjunctions, diagrams, higher categories, operads/properads, and monoidal categories through API examples (Vostokov 2024a, 251–68). It proposes API calls as objects connected by glue code, translations between API layers and sequences as functors, stack transformations as natural transformations, and independent calls as monoidal composition.

Chapter 3 formalizes the call-as-object proposal through a compatibility graph and its free category, with typed call contracts as vertices and admissible glue programs as edges. It also develops a complementary call-as-arrow model whose objects are resource states, showing exactly when API calls compose. Chapter 9 requires the adjunction laws before using the label mathematically. Chapters 14–15 incorporate the higher, operadic, properadic, and monoidal ideas while marking their formal obligations.

This source also motivates cross-platform naturality. In this book, we map platform-specific API or internals categories into a common semantic category and treat mismatch in ownership, lifetime, error, or concurrency semantics as a translation residual.

E.4 *Memory Dump Analysis Anthology*, Volume 17

Trace Precision and Recall, **Trace Polynomial**, **Trace Context**, and **Trace Plan** extend trace analysis toward evaluation, algebraic representation, capture context, and requirements (Vostokov 2025, 17:36–42). Chapters 14 and 20 develop these proposals: Trace Polynomial becomes a two-composition model requiring explicit laws; Trace Plan becomes the desired observation construction; precision and recall become complementary adequacy measures rather than an assumed categorical duality.

Declarative Memory distinguishes source-level values from virtual and physical layouts and notes many-to-many effects of compilation and optimization (Vostokov 2025, 17:43–44). Chapter 3 represents the layers with partial functors, relations, or profunctors as appropriate and derives a symbolization commutativity test.

OS Internals and Category Theory proposes Windows, Linux, macOS, and cross-OS unification (Vostokov 2025, 17:49). The source notes that some AI-generated associations appeared strange. Chapter 3 therefore retains the research direction while limiting claims to typed semantic subcategories and testable preservation laws.

Observation Spaces enumerates the many memory, trace, metric, narrative, presentation, internal, state, and data spaces used in diagnostics (Vostokov 2025, 17:69–71). Chapter 16 treats them as a family of categories and mappings rather than one undifferentiated space.

The **Diagnostics–Observability Square** distinguishes two pairs: observability and diagnosability as properties, and observation and diagnostics as processes (Vostokov 2025, 17:72–74). In Chapter 16, we turn that conceptual square into a compatibility equation between specifications and implementations.

Pattern Category Theory proposes patterns as objects and transformations, inheritance, composition, and application as morphisms (Vostokov 2025, 17:96–97). The article itself cautions that some generated details or references may be hallucinated. Chapter 19 responds by separating categories of specifications, artifacts, and instances from detector functors and operadic pattern composition.

E.5 What is new in the synthesis

The source books give us the program and many of its key concepts. In this book, we add an operational semantics of *preservation and failure*:

- every categorical word comes with objects, arrows, laws, and scope—or is labeled analogy;
- every observation carries a loss contract;
- every cross-view claim is made well typed through common codomains or explicit mediators;
- every expected diagram has a comparator, missingness policy, residual, and localization strategy;
- every failed law distinguishes system, instrumentation, transport, schema, analysis, and model explanations;
- every incident can yield a reusable invariant card and executable regression check.

This concordance is also a research ledger. Future editions can add empirical evaluations, stronger formal constructions, corrections, and further source ideas without erasing where each concept began.

E.6 Source concepts and their claim status

The source catalogs close with alphabetical lists of mathematical concepts that inspired pattern names (Vostokov 2024b, 282–84; 2026, 382–83). The table applies our three-status discipline to the concepts this book actually uses. We classify the mathematical construction and its diagnostic interpretation separately, because a standard theorem does not make every domain identification standard.

Source concept or pattern	Status in this book	Where
Adjoint (Adjoint Thread; Adjoint Message)	Standard mathematics: $\exists A \dashv A^* \dashv \forall A$. Reading the patterns as fibers, quotients, and images is our Diagnostic construction.	Ch. 9; B.7
Galois connection (Galois Trace)	Standard mathematics for the explicit nonempty finite aligned orders. Calibration, tie handling, and sentinel meanings are Diagnostic choices.	Ch. 9; B.7
Cartesian product (Cartesian Trace)	Standard product when factors and independence are declared; the pattern identification is Diagnostic.	Ch. 7

Source concept or pattern	Status in this book	Where
Pullback (Trace Join)	Standard pullback over a declared key. It establishes compatibility, not causality.	Ch. 7
Quotient and image factorization	Standard for a total attribute map. Applying them to Quotient Trace and Adjoint Message is Diagnostic.	Ch. 9–10
Cover and sheaf (Message Cover; Sheaf of Activities)	Attribute fibers give a standard partition cover. A sheaf is a Diagnostic construction only after restriction and gluing axioms are verified.	Ch. 12
Toposes and contextual truth	Standard mathematics. Selecting scopes, covers, evidence, and hypotheses is Diagnostic; geometric-morphism comparisons remain a research direction until specified.	Ch. 12; B.11; D.7; H.7
Presheaf (Trace Presheaf; debugging)	Diagnostic construction on declared scopes; identity and composite restriction laws remain executable tests.	Ch. 11–12
Functor or scalar field (Trace Field)	Every column is a set function. It is functorial on a discrete message category, or on a richer category only when arrows are preserved.	Ch. 16
Flag and filtration; Trace Window	Inclusion orders and filtrations are Standard mathematics. Treating windows as restriction scopes is Diagnostic.	Ch. 11
Glued Activity	Diagnostic construction for a key collision such as PID reuse. The name makes no claim of topological gluing.	Ch. 7; C.4
2-categories and whiskering	Diagnostic construction only after parallel 1-cells, both compositions, identities, and interchange are supplied; otherwise a Guiding analogy.	Ch. 14
Operads (Diagnostic Operads)	Diagnostic colored-operad construction when colors, arities, substitution, identities, and laws are declared.	Ch. 15
Duality (Codiagnostics; CoTrace; CoActivity)	Guiding analogy unless an opposite category and the relevant contravariant construction are identified.	Ch. 10
Coalgebra and behavior functor	Coalgebra is Standard mathematics. A source-to-indicator functor changes category and is not a coalgebra by itself.	Ch. 13
Topology and Structure Sheaves	Guiding analogy and research proposal until a topology, cover, sections, restrictions, and sheaf axioms are defined.	Ch. 30

Other catalog names—among them braid, homotopy, fiber bundle, manifold, motive, and retraction—remain names or guiding analogies here unless a chapter supplies the defining structure and laws. Omission from this table is not a promotion. This ledger records where the book formalizes a source idea, where it proposes a diagnostic interpretation, and where it deliberately stops.

Appendix F — Notation and glossary

F.1 Notation

Notation	Meaning in this book
A, B, C	Objects in a currently declared category
$f: A \to B$	A morphism with source A and target B
$g \circ f$	First f, then g
id_A	Identity morphism on A
$\mathcal{C}, \mathcal{D}$	Categories
$\mathcal{C}^{op}$	Opposite category: arrows of $\mathcal{C}$ reversed
$F: \mathcal{C} \to \mathcal{D}$	A functor
$\eta: F \Rightarrow G$	A natural transformation
η_A	Component of η at object A
$F \dashv G$	F is left adjoint to G
$\mathrm{Hom}_{\mathcal{C}}(A, B)$	Morphisms from A to B in $\mathcal{C}$
$A \times B$	Product of A and B
$A + B$	Coproduct of A and B
$A \times_C B$	Pullback over C
$P: \mathcal{C}^{op} \to \mathbf{Set}$	A presheaf on $\mathcal{C}$
$(\mathcal{B}, J)$	A site: a scope category equipped with a Grothendieck topology
$\widehat{\mathcal{B}} = [\mathcal{B}^{\mathrm{op}}, \mathbf{Set}]$	The presheaf topos on the scope category
$\mathbf{Sh}(\mathcal{B}, J)$	The topos of sheaves satisfying coverage policy J
Ω	The subobject classifier, or object of contextual truth values
$\chi_H: E \to \Omega$	The characteristic map of hypothesis subobject H into evidence E
$A \otimes B$	Monoidal product; its meaning is model-specific
$x \cong y$	Isomorphic objects
$x \approx y$	Approximately equal under a declared rule

Notation	Meaning in this book
$d(x, y)$	Declared distance or residual measure
$\varnothing$	Empty set
$x \in A$	x is an element of A
$A \subseteq B$	A is a subset of B
$\rho_U^V : P(V) \to P(U)$	Restriction from the larger scope V to the smaller scope U

F.2 Glossary

Abstract interpretation. Reasoning about program behavior through sound approximations. A full construction specifies the concrete and abstract domains and sound operations (B.23).

Adjunction. A pair of functors related by a natural correspondence between hom-sets, equivalently by a unit and counit satisfying triangle identities. It is stronger than "two opposite operations."

Algebra of an operad. A concrete interpretation of abstract operadic operations as actual types and functions.

Anomaly-detector generator. Our phrase for category theory's role in supplying law templates—composition, naturality, gluing, or universality—that can be instantiated as anomaly tests.

Approximate pasting. Combining local discrepancy bounds into an outer bound, with a justified bound on downstream amplification (B.22).

Arrow. Informal synonym for morphism. Its source and target are part of its type.

Artifact category. A diagnostic category whose objects are evidence artifacts or views and whose morphisms are transformations or declared relations among them.

Associativity. The law $h \circ (g \circ f) = (h \circ g) \circ f$. It changes parentheses, not order.

Attribute map. A function from trace messages or artifacts to values such as thread, process, operation, function, or module. Its fibers, images, preimages, and quotient may carry diagnostic meaning.

Bicategory. A higher category in which associativity and identity of 1-morphisms hold up to coherent isomorphism rather than strict equality.

Bisimulation. A relation showing that states have matching observable behavior under a coalgebraic model.

Categorical Invariant Violation (CIV). A failure of a system, observation map, adapter, or implementation to realize an explicitly intended categorical law. It is not a failure of category theory itself.

Category. Objects and morphisms equipped with associative composition and an identity morphism for every object.

Category mistake. In this book, an unjustified use of categorical language—for example, calling a graph a category without identities or closure under composition.

Claim ledger. A record connecting each diagnostic claim to artifacts, transformations, assumptions, alternatives, and disconfirming evidence.

Closure operator. A monotone operation c on an ordered set satisfying $x \leq c(x)$ and $c\big(c(x)\big) = c(x)$. In Galois Trace, the round trip $G \circ F$ closes a message to its alignment boundary.

Coalgebra. For an endofunctor F, an object X with a map $X \to F(X)$, commonly used to model state and observable behavior.

Codiagnostics. Diagnostic work in which artifact transformations and cooperating analysis operations are part of extracting indicators and explanations.

Codomain. The declared target object or set of a mapping. It may contain values not actually reached.

Colimit. A universal compatible assembly of a diagram, generalizing coproducts, pushouts, and coequalizers.

Color. An input or output type in a colored operad.

Commutative diagram. A diagram whose declared parallel paths have equal composites.

Comparator. The operational rule used to compare path outputs: exact equality, normalized equality, equivalence, order, distance, or compatibility.

Composition. Combining $f: A \to B$ and $g: B \to C$ to obtain $g \circ f: A \to C$.

Cone. A compatible family of arrows from one object to the objects in a diagram. Limits are universal cones.

Contravariant. Reversing the direction of arrows; formally, covariant from the opposite category.

Coproduct. A universal object receiving injections from each component. In **Set**, it behaves like a tagged disjoint union.

Correlation witness. A retained record explaining an association between two evidence objects; witness pairs compose by matching the shared endpoint (B.24).

Diagnosability. A property relative to a fault set, observation model, and decision requirement: whether relevant conditions can be distinguished or inferred.

Diagnostic operad. A colored operad whose operations are typed diagnostic transformations or analyses.

Diagnostic sufficiency. The retained observation determines the required answer for every case in the declared model; cases with the same observation must have the same answer (B.21).

Diagnostics–Observability Square. The source distinction and relationship among observability and diagnosability as properties, and observation and diagnostics as processes; we make it a specification–implementation compatibility test.

Diagram. A functor from a shape category into a category. Informally, a typed arrangement of objects and arrows.

Endofunctor. A functor from a category to itself.

Enriched category. A category whose hom-objects live in a structure richer than sets, supporting quantities such as order, cost, or distance.

Equality policy. The exact semantic rule under which two path results count as equal or compatible.

Equalizer. A universal object selecting where two parallel arrows agree.

Essentially surjective. A functor property: every target object is isomorphic to some mapped source object.

Execution category. A model of runtime states, events, calls, or paths chosen for a diagnostic question.

Faithful functor. A functor that does not merge distinct parallel morphisms. Non-faithfulness can model intentional information loss.

Free category. The category generated by a directed graph: vertices are objects and finite paths are morphisms.

Full functor. A functor for which every target morphism between mapped objects is the image of a source morphism.

Functor. A mapping between categories that maps objects and morphisms while preserving identities and composition.

Galois connection. Monotone maps $F: P \to Q$ and $G: Q \to P$ between ordered sets such that $F(p) \leq q$ exactly when $p \leq G(q)$. It is an adjunction between preorder categories.

Geometric morphism. A standard relationship between toposes, given by adjoint inverse-image and direct-image functors whose inverse-image functor preserves finite limits.

Gluing. Combining compatible local sections into a global section. A sheaf guarantees unique gluing for its declared covers.

Grothendieck topology. A rule specifying which families of arrows or sieves count as covers, subject to identity, stability, and composition laws.

Guiding analogy. An explicit, useful categorical resemblance whose full structure or laws have not been established.

Heyting algebra. An ordered algebra of intuitionistic truth values supporting conjunction, disjunction, implication, and negation without requiring every proposition to be either true or false.

Hom-set. The collection of morphisms from one object to another in a locally small category.

Hypothesis category. A category or preorder of diagnostic claims, refinements, implications, or evidential support relations.

Identity. The do-nothing morphism $\mathrm{id}_A: A \to A$ satisfying left and right identity laws.

Internal logic. The logic interpreted by the objects, subobjects, and arrows inside a topos. It is generally intuitionistic rather than automatically Boolean.

Invariant card. A versioned specification of a diagnostic diagram, scope, comparator, residual, confounders, localization procedure, and consequence.

Isomorphism. A morphism with a two-sided inverse. It expresses reversible structural sameness inside a category.

Kan extension. A universal extension or transport of a functor along another functor.

Lax functor. A functor-like mapping that preserves composition and identity through specified comparison morphisms rather than strict equalities.

Limit. A universal compatible family over a diagram, generalizing products, pullbacks, and equalizers.

Lipschitz bound. An upper bound on how much a transformation can amplify a distance over the relevant domain (B.22).

Loss profile. A declaration of which identities, relations, orderings, multiplicities, values, or scopes an observation mapping preserves or discards.

Metamorphic testing. Testing expected relations among related inputs and outputs, including transformations of artifacts and their reports (B.25).

Model-based diagnosis. Finding component assumptions that restore consistency between a system description and observations; a failed comparison alone need not identify a unique fault.

Monad. An endofunctor with unit and multiplication satisfying identity and associativity laws.

Monoidal category. A category with a coherent tensor product and unit, used for a declared form of side-by-side composition.

Morphism. A typed arrow between objects. It may be a function, transition, refinement, relation instance, or transformation according to the category.

Natural transformation. A coherent family of component morphisms between two functors with the same source and target categories.

Naturality residual. A distance or discrepancy between the two paths around a naturality square.

Noncommutativity. Failure of specifically declared parallel paths to agree. It does not mean that all category composition was expected to commute.

Object. A node of a category, characterized categorically through its relationships rather than required internal contents.

Observability. A relational property of a system and observation scheme with respect to state distinctions and goals—not merely the presence of telemetry.

Observability drift. A failure of compatibility between system evolution and evidence-schema or instrumentation evolution.

Observation functor. A functorial model of how selected execution structure becomes evidence. Real instrumentation may require a restriction, partial map, or lax variant.

Observation Space. A source term for a domain in which software observations are organized; formalization requires objects, arrows, comparison, and scope.

Observational equivalence. Indistinguishability under a declared family of observations.

Operad. A structure for typed many-input, one-output operations and their substitution-based composition.

Opposite category. The category obtained by reversing every morphism while retaining objects and reversed composition.

Partial order. A reflexive, transitive, antisymmetric relation. Distributed happens-before is partial because concurrent events may be incomparable.

Pattern Category Theory. A source proposal treating patterns and their relations categorically; we refine it into separate structures for specifications, artifacts, instances, detectors, and composition.

Presheaf. A contravariant set-valued functor, commonly representing data that restricts from larger scopes to smaller scopes.

Product. A universal object with projections to each factor. A product pairs without imposing a shared-key constraint.

Profunctor. A generalized relation between categories, useful when a many-to-many correspondence is more honest than a functor.

Properad. A structure for operations with multiple inputs and multiple outputs and their wiring-based composition.

Property-based testing. Generating inputs to test executable properties, while recording their preconditions and retaining counterexamples.

Pullback. A universal compatible join over a shared target. It expresses matching, not causality.

Pushout. A universal gluing of targets along a shared source.

Residual. A structured or numerical account of how two expected-to-agree routes differ.

Restriction. A presheaf map from data on a larger scope to data on a smaller scope.

Schema category. A category whose objects and arrows encode entity types and relations; a set-valued functor can represent an instance.

Sheaf. A presheaf satisfying unique gluing of compatible local sections for declared covers.

Sheafification. The universal conversion of a presheaf into a sheaf for a declared topology. It repairs formal gluing behavior, not missing telemetry.

Sieve. A collection of arrows into one object that remains closed when any arrow is precomposed. For scopes, it is a family of narrower views closed under further narrowing.

Site. A category equipped with a Grothendieck topology, hence with a declared notion of cover.

Sound overapproximation. A retained set guaranteed to include every concrete state admitted by the model, possibly including additional states (B.23).

Span, categorical. A witness object with two arrows to its endpoint objects. In sets, composition matches witnesses through a pullback (B.24).

Subobject classifier. An object Ω that classifies every subobject by a characteristic map, generalizing the set $\{\text{false}, \text{true}\}$.

Topos. A category with set-like constructions and an internal logic. Presheaves on a small category and sheaves on a site form the toposes used in this book.

Trace span. A tracing record for a unit of work, with its context, timing, attributes, and events; distinct from a categorical span.

Typed missingness. Representation that distinguishes causes such as not-created, not-observed, sampled, not-exported, undefined, and unknown when evidence permits.

Universal property. A definition by a unique factorization relationship to every competing object of the relevant kind.

Viewpoint. A structured partial perspective with its own objects, arrows, granularity, relevance, and vocabulary.

Visibility Limit. A source pattern naming a boundary beyond which evidence is unavailable or invisible; we refine it as a scoped loss profile.

Whiskering. In a 2-category, composing a 2-morphism with an adjacent 1-morphism to extend it by a common prefix or suffix.

Yoneda viewpoint. Understanding an object through all morphisms into or out of it; formally grounded in the Yoneda lemma.

Appendix G — The conceptual emblem

The conceptual emblem from the cover.

G.1 A visual synopsis, not an unlabeled proof

The emblem on page 1 compresses the book's central argument into one picture. It is deliberately conceptual: because its nodes and arrows are not labeled and no equality has been declared, it is not by itself a formal commutative diagram. In this appendix, we supply its intended reading.

Read the left node as an initial software state, evidence artifact, or diagnostic question, and the right node as a claim or view to be checked. The upper and lower nodes are two intermediate representations. They form two routes from the same source to the same target:

$$A \xrightarrow{f} B \xrightarrow{k} D \qquad \text{and} \qquad A \xrightarrow{g} C \xrightarrow{h} D.$$

The routes may stand for, for example, a conclusion derived through traces and the same conclusion derived through logs; a source-level value reconstructed through symbols and the value reconstructed from raw memory; or an operation interpreted through two telemetry-schema versions. The colors distinguish heterogeneous representations and transformations. They do **not** assign a permanent color to any particular evidence modality.

G.2 The lens and the red point

The central magnifying glass represents diagnostic comparison and localization. Once both routes have been mapped into a common comparison domain, the intended invariant can be written

$$k \circ f \;\approx\; h \circ g.$$

The symbol $\approx$ is intentionally general. Depending on the invariant card, it may mean exact equality, equality after normalization, semantic equivalence, agreement within a tolerance, or another declared compatibility relation. If a distance d is meaningful, the red point can be read as the residual

$$r = d(k \circ f, h \circ g),$$

the discrepancy that the investigator tries to measure and localize. A zero or acceptable residual supports the invariant. A material residual is a candidate **Categorical Invariant Violation**. It is evidence to investigate, not automatic proof of an application defect: the cause may lie in the system, instrumentation, transport, schema, analysis, or model.

G.3 Structure · Evidence · Invariance

The cover line names the three layers of this reading:

- **Structure:** the typed nodes, arrows, and composable paths;
- **Evidence:** the runtime observations or derived artifacts carried by those nodes;
- **Invariance:** the explicitly expected agreement of independent routes.

The emblem therefore acts as a compact promise for the method used throughout the book: construct more than one well-typed route from evidence to a claim, declare how their results can be compared, and turn disagreement into a localizable diagnostic question. Appendices A and B explain the notation and mathematical prerequisites from school-level mathematics, and Appendix C shows how to record the operational details as invariant cards.

Appendix H — Applied category theory: the diagnostic program

We present this book as a work of applied category theory (ACT). By this label, we do not mean that category theory is merely illustrated by software examples. We mean that categorical structures are built from an operational domain, used to compose models across boundaries, and made answerable to evidence. We intend not only a diagram or theorem but also an engineering contract: what must agree, how disagreement is measured, and what should change when the contract fails.

No mathematics beyond school-level ideas is needed to read this appendix. We may read an object as a kind of thing, an arrow as an allowed transformation, and a path as steps performed in order. We say that a diagram commutes when two declared routes give compatible results. Appendices A and B develop the mathematics separately from sets, functions, logic, and graphs, and this appendix explains what makes their use applied.

H.1 What 'applied' commits us to

Applied category theory uses the abstractions of category theory to organize a subject outside pure category theory while preserving the subject's own meanings. In this book, calling a model ACT makes five commitments:

Domain meaning. Every important object and arrow has an operational interpretation: an execution state, artifact, schema, observation, transformation, or diagnostic claim.

Typed composition. Arrows compose only when their declared output and input match. A convenient but ill-typed pipeline is rejected before its numerical result is trusted.

Preserved structure. The model states which identities, compositions, orders, interfaces, restrictions, or equivalences should survive observation and normalization.

Operational consequence. A law is connected to an inspection, query, comparison, residual, or decision. If no observable consequence can be named, the construction remains exploratory.

Revision by evidence. A failed test may expose the software, instrumentation, transport, schema, analysis, or categorical model. The model is not protected from the evidence it organizes.

These commitments distinguish an applied categorical model from decorative relabeling. Calling services 'objects' and calls 'morphisms' is only a starting analogy. The work begins when sources and targets are typed, composition is defined, laws are stated, and their diagnostic value is tested.

H.2 Why the framework in this book is ACT

The framework repeatedly moves from domain evidence to categorical structure and back again:

Execution and evidence categories separate what the software did from what a log, trace, metric, profile, dump, or derived view records about it.

Observation and normalization functors make representation changes explicit and expose loss, aggregation, omission, and schema drift.

Commutative diagrams and naturality tests turn expected agreement between routes into executable comparisons in a declared common evidence category.

Limits, colimits, presheaves, sheaves, toposes, coalgebra, and operads supply disciplined models for joins, fusion, scoped evidence, local-to-global consistency, evolving behavior, and typed workflow composition.

The Categorical Invariant Violation pattern packages an expected diagram with scope, equality policy, residual, provenance, and localization procedure.

Artifact-based cases force the abstractions to meet actual data, declared parsing rules, reproducible transformations, and explicit limits.

The newer source ideas, Observation Spaces, the Diagnostics–Observability Square, Declarative Memory, Trace Plan, Trace Polynomial, and Pattern Category Theory, therefore enter as attributed proposals. They become ACT only after their types, composition rules, preservation claims, and possible refutations are stated (Vostokov 2025, 17:38–49, 69–74, 96–97).

H.3 The applied diagnostic cycle

We can organize an ACT investigation as a repeatable cycle. The order is practical rather than dogmatic, and later evidence may require an earlier choice to be revisited.

Step 1 — Frame the diagnostic question. Name the uncertainty or decision before choosing a categorical construction.

Step 2 — Type the domain. List the relevant kinds of states, artifacts, records, schemas, and claims, together with allowed transformations.

Step 3 — Declare composition and scope. Say which operations can follow which, where identities live, and what time window, version, host, request, or population the model covers.

Step 4 — Choose comparison routes. Construct independent paths to a common target or provide a justified mapping into a common comparison domain.

Step 5 — Derive the invariant. State exact equality, normalized equality, semantic equivalence, tolerance, gluing condition, or other compatibility rule.

Step 6 — Measure and localize. Compute the residual or failed predicate, then decompose the path until the smallest failed preservation contract is found.

Step 7 — Adjudicate and revise. Decide whether the evidence challenges the software, observation pipeline, diagnostic hypothesis, or model, and record the resulting change.

The cycle is what makes the abstraction operational, and it also prevents a common mistake: treating a commuting diagram as proof of causality. Agreement supports only the declared invariant. Disagreement localizes a question, but it does not by itself identify the responsible component or mechanism.

H.4 Composition, interfaces, and black-boxing

A central theme of ACT is compositionality: model parts together with their interfaces so that a larger model can be assembled from smaller ones. Hypergraph categories, decorated cospans, and structured cospans make this idea precise for network-like systems (Fong 2016; Fong and Spivak 2019; Baez and Courser 2020). Open reaction networks demonstrate the method in detail: networks compose by connecting outputs to inputs, and functors carry that composition into dynamical and steady-state semantics (Baez and Pollard 2017).

In plain language, an open system is a component whose boundary is part of the model. Black-boxing replaces its internal construction by the externally visible relation it imposes at that boundary, and this is powerful because

different implementations can be compared through the same interface. It is also diagnostically dangerous when the black box discards the distinction needed to separate two fault states.

Software incidents are full of such boundaries: service calls, process crossings, queues, storage APIs, schema versions, symbol layers, and collector pipelines. Chapters 5, 14, 18, 23, and 27 apply interface-aware composition to evidence. The worked pasting example in Chapter 5 shows the diagnostic payoff: an outer failure can be decomposed into local squares, so the first broken interface can be investigated without assuming that every internal component is visible.

String and wiring diagrams provide a readable syntax for this compositional view. Their value is not the drawing alone but the rule that connecting diagrams corresponds to composing the processes they denote. The broader system–process reading links categorical diagrams across computation, logic, topology, and physics (Baez and Stay 2011); we specialize that reading to transformations of runtime evidence.

H.5 From diagrams to computational artifacts

Applied category theory need not stop at pen-and-paper models. Attributed C-sets, or acsets, represent graphs, tables, wiring diagrams, Petri nets, and related structures as category-shaped data with typed attributes, and they have an implementation designed for technical computing (Patterson, Lynch, and Fairbanks 2022). This is relevant to diagnostics because evidence graphs combine topology with attributes such as time, process, status, schema version, and provenance.

A tool-facing ACT design for our framework can be developed in layers:

Categorical schema. Encode artifact types, arrows, boundary ports, and permitted compositions.

Evidence instance. Load concrete events and attributes while preserving typed missingness and provenance.

Transformation layer. Implement parsing, normalization, migration, restriction, projection, and aggregation as declared mappings.

Invariant compiler. Translate commuting diagrams, gluing requirements, and preservation laws into predicates or quantitative residuals.

Diagnostic report. Return the failed path, smallest local discrepancy, equality policy, scope, and model version rather than only a generic alert.

The present manuscript specifies the conceptual contracts and includes executable-style recipes and one real artifact case. It does not claim that this full toolchain has already been implemented. The distinction matters: an implementable design is a proposal until code, tests, performance, and diagnostic utility are demonstrated.

H.6 Established foundations and the book's proposal

In Appendix D, we survey established categorical work in software semantics and architecture, model synchronization, databases, coalgebraic behavior, open systems, sheaf-based consistency, and topos-based logic. Standard category theory and those application traditions are not claimed as inventions of this book. Spivak develops category theory as a modeling language across the sciences (Spivak 2014a), while Fong and Spivak center compositionality in an explicitly applied introduction (Fong and Spivak 2019).

The source books establish an earlier program of categorical ideas for software diagnostics, trace patterns, Windows APIs, memory analysis, and software internals (Vostokov 2024b, 2026, 2024a, 2025). The proposal made here is their operational synthesis: modality-specific evidence categories connected to execution and claims; cross-view naturality in a declared common codomain; measurable residuals; versioned observation functors; local-to-global evidence consistency; and compositional diagnostic patterns.

The novelty is therefore a new application program within ACT, not a claim to have invented ACT or its base constructions. Its strongest claims are methodological and testable: categorical structure can generate useful diagnostic invariants, organize heterogeneous evidence, and localize failed preservation contracts. Whether a particular invariant improves real diagnosis remains an empirical question.

H.7 The diagnostic topos proposal

Chapters 11 and 12 and Appendix B let us state the proposal as one typed construction:

Base category $\mathcal{B}$. Objects are diagnostic scopes; arrows are permitted refinements or inclusions.
Coverage topology J. Covers state what counts as sufficient observation of a scope.
Evidence object E. A presheaf or sheaf assigns typed evidence states and restriction maps.
Hypothesis subobject $H \hookrightarrow E$. Membership is required to survive restriction.
Characteristic map $\chi_H : E \to \Omega$. Its value records the refinements under which the evidence belongs to H.
Operational consequence. Gluing failures and changes in the supporting sieve become reportable diagnostic results.

The first five lines use standard topos constructions. Their software meanings are our diagnostic construction. The final line is an engineering claim that must be evaluated against artifacts, incident decisions, and competing methods. This separation is the novelty boundary.

A future version may compare two instrumentation regimes by a geometric morphism between their evidence toposes. Such a comparison could express how evidence and contextual truth move across schema versions, sampling policies, or collector boundaries. We keep it as a research direction until the two toposes, the inverse-image and direct-image functors, and the finite-limit preservation claim are supplied.

The diagnostic topos does not manufacture missing data, assign probabilities, prove causality, or identify root cause by itself. Its contribution is compositional: it makes evidence scope, adequate coverage, compatible gluing, and context-dependent hypothesis support part of one checkable model.

H.8 Books and papers: a reading route

The following route complements the references in Appendix D and avoids requiring advanced mathematics at the start:

Toposes and categorical logic. Mac Lane and Moerdijk provide the principal bridge from sheaves to toposes and logic; Borceux develops internal logic; Johnstone is the advanced reference. Birkedal and colleagues give a software-semantics application, while Felber, Hummes Flores, and Rincon Galeana connect sheaves to distributed task solvability (Mac Lane and Moerdijk 1994; Borceux 1994; Johnstone 2002; Birkedal et al. 2012; Felber et al. 2025).

School-level conceptual entry. Lawvere and Schanuel begin with arrows, composition, universal properties, and everyday examples, making *Conceptual Mathematics* a natural bridge from Appendix A to the categorical vocabulary of Appendix B (Lawvere and Schanuel 2009).

Models across science. Spivak's Category Theory for the Sciences begins from sets, functions, databases, and examples, then develops categories, functors, natural transformations, limits, sheaves, monads, and operads (Spivak 2014a).

Compositional ACT. Fong and Spivak's An Invitation to Applied Category Theory is the principal book-length route through compositional modeling, including orders, databases, wiring diagrams, and signal flow (Fong and Spivak 2019).

Sheaves through examples. Rosiak develops sheaf theory through concrete constructions, with applied and advanced connections. This is further reading for the local-to-global methods used here, rather than a prerequisite for the school-mathematics route or prior work on this particular diagnostic synthesis (Rosiak 2022).

Open and interconnected systems. Fong's thesis develops hypergraph categories, decorated cospans, corelations, and semantic functors; Baez and Pollard provide a concrete compositional application to reaction networks (Fong 2016; Baez and Pollard 2017).

Computational representation. Patterson, Lynch, and Fairbanks connect functorial data models to implementable attributed data structures for technical computing (Patterson, Lynch, and Fairbanks 2022).

Processes across disciplines. Baez and Stay explain how monoidal categorical language relates systems and processes in computation, logic, topology, and physics (Baez and Stay 2011).

A reader interested specifically in diagnostics can then return to Chapters 3–7 for observation functors and commutativity, Chapters 14–19 for schema, pattern, and workflow composition, Chapters 21–22 for the invariant pattern and debugging protocol, Chapters 23–27 and 31 for case studies, and Chapters 28–29 for automation and architecture. Appendices A and B remain the school-mathematics route to every formal term used there.

H.9 Claim discipline for applied category theory

ACT is strongest when different kinds of claims are kept separate. Before publishing or automating a construction from this book, record its status:

Standard mathematics. Give the accepted definition or theorem and cite an appropriate source.

Diagnostic construction (domain interpretation). Define what the objects, arrows, composition, and preserved structure mean for the software and evidence at hand.

Guiding analogy or proposal. Label a suggested construction as such until its typing, laws, and operational consequences have been established.

Measured result. Record artifact identity, scope, transformations, equality policy, residual, and reproducibility conditions.

Diagnostic conclusion. State what the result supports, what it does not establish, and which alternative explanations remain.

Here applied category theory requires us to specify the operational meaning of the objects and arrows and to test the laws used in the diagnostic construction. Categorical structure will be tied to operational semantics, compositional interfaces, and refutable evidence. It will not manufacture missing data, causal mechanisms, or correctness. We therefore label each proposed construction, record the conditions of every evidence-supported result, and explain the needed mathematics in separate appendices that begin with school-level prerequisites.

References

Abreu, Rui, Peter Zoeteweij, Rob Golsteijn, and Arjan J. C. van Gemund. 2009. "A Practical Evaluation of Spectrum-Based Fault Localization." *Journal of Systems and Software* 82 (11): 1780–92. https://doi.org/10.1016/j.jss.2009.06.035.

Abreu, Rui, Peter Zoeteweij, and Arjan J. C. van Gemund. 2011. "Simultaneous Debugging of Software Faults." *Journal of Systems and Software* 84 (4): 573–86. https://doi.org/10.1016/j.jss.2010.11.915.

Baez, John C., and Kenny Courser. 2020. "Structured Cospans." *Theory and Applications of Categories* 35 (48): 1771–1822. https://doi.org/10.70930/tac/dca3obx4.

Baez, John C., and Blake S. Pollard. 2017. "A Compositional Framework for Reaction Networks." *Reviews in Mathematical Physics* 29 (9): 1750028. https://doi.org/10.1142/S0129055X17500283.

Baez, John C., and Mike Stay. 2011. "Physics, Topology, Logic and Computation: A Rosetta Stone." In *New Structures for Physics*, edited by Bob Coecke, 95–172. Lecture Notes in Physics 813. Springer. https://doi.org/10.1007/978-3-642-12821-9_2.

Birkedal, Lars, Rasmus Ejlers Møgelberg, Jan Schwinghammer, and Kristian Støvring. 2012. "First Steps in Synthetic Guarded Domain Theory: Step-Indexing in the Topos of Trees." *Logical Methods in Computer Science* 8 (4:1): 1–45. https://doi.org/10.2168/LMCS-8(4:1)2012.

Borceux, Francis. 1994. "Internal Logic of a Topos." In *Handbook of Categorical Algebra 3: Categories of Sheaves*, 342–431. Cambridge University Press. https://doi.org/10.1017/CBO9780511525872.008.

Chen, Tsong Yueh, Fei-Ching Kuo, Huai Liu, Pak-Lok Poon, Dave Towey, T. H. Tse, and Zhi Quan Zhou. 2018. "Metamorphic Testing: A Review of Challenges and Opportunities." *ACM Computing Surveys* 51 (1), article 4: 1–27. https://doi.org/10.1145/3143561.

Claessen, Koen, and John Hughes. 2000. "QuickCheck: A Lightweight Tool for Random Testing of Haskell Programs." In *Proceedings of the Fifth ACM SIGPLAN International Conference on Functional Programming*, 268–79. ACM. https://doi.org/10.1145/351240.351266.

Cousot, Patrick, and Radhia Cousot. 1977. "Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints." In *Proceedings of the Fourth ACM SIGPLAN–SIGACT Symposium on Principles of Programming Languages*, 238–52. ACM. https://www.di.ens.fr/~cousot/COUSOTpapers/POPL77.shtml.

Davey, B. A., and H. A. Priestley. 2002. *Introduction to Lattices and Order*. 2nd ed. Cambridge University Press. https://doi.org/10.1017/CBO9780511809088.

Diskin, Zinovy, and Tom Maibaum. 2012. "Category Theory and Model-Driven Engineering: From Formal Semantics to Design Patterns and Beyond." *Electronic Proceedings in Theoretical Computer Science* 93: 1–21. https://doi.org/10.4204/EPTCS.93.1.

Eilenberg, Samuel, and Saunders Mac Lane. 1945. "General Theory of Natural Equivalences." *Transactions of the American Mathematical Society* 58 (2): 231–94. https://doi.org/10.1090/S0002-9947-1945-0013131-6.

Felber, Stephan, Bernardo Hummes Flores, and Hugo Rincon Galeana. 2025. "A Sheaf-Theoretic Characterization of Tasks in Distributed Systems." arXiv:2503.02556 [cs.DC]. https://doi.org/10.48550/arXiv.2503.02556.

Fiadeiro, José Luiz. 2005. *Categories for Software Engineering*. Springer. https://doi.org/10.1007/b138249.

Foley, John D., Spencer Breiner, Eswaran Subrahmanian, and John M. Dusel. 2021. "Operads for Complex System Design Specification, Analysis and Synthesis." *Proceedings of the Royal Society A* 477 (2250): 20210099. https://doi.org/10.1098/rspa.2021.0099.

Fong, Brendan. 2016. *The Algebra of Open and Interconnected Systems*. DPhil thesis, University of Oxford. https://arxiv.org/abs/1609.05382.

Fong, Brendan, and David I. Spivak. 2019. *An Invitation to Applied Category Theory: Seven Sketches in Compositionality*. Cambridge University Press. https://doi.org/10.1017/9781108668804.

Fonseca, Rodrigo, George Porter, Randy H. Katz, Scott Shenker, and Ion Stoica. 2007. "X-Trace: A Pervasive Network Tracing Framework." In *4th USENIX Symposium on Networked Systems Design and Implementation*, 271–84. https://www.usenix.org/conference/nsdi-07/x-trace-pervasive-network-tracing-framework.

Goguen, Joseph A. 1991. "A Categorical Manifesto." *Mathematical Structures in Computer Science* 1 (1): 49–67. https://doi.org/10.1017/S0960129500000050.

Horstmeyer, Leonhard, and Sharwin Rezagholi. 2020. “Partially Observable Systems and Quotient Entropy via Graphs.” https://arxiv.org/abs/1909.00376.

Internet Traffic Archive. 1995. “NASA-HTTP: Two Months of HTTP Logs from the KSC-NASA WWW Server.” Collected by Jim Dumoulin; contributed by Martin Arlitt and Carey Williamson. Accessed August 30, 2026. https://ita.ee.lbl.gov/html/contrib/NASA-HTTP.html.

Jacobs, Bart. 2017. *Introduction to Coalgebra: Towards Mathematics of States and Observation*. Vol. 59. Cambridge Tracts in Theoretical Computer Science. Cambridge University Press. https://doi.org/10.1017/CBO9781316823187.

Johnstone, Peter T. 2002. *Sketches of an Elephant: A Topos Theory Compendium*. 2 vols. Oxford Logic Guides 43–44. Clarendon Press. https://global.oup.com/academic/product/sketches-of-an-elephant-9780198524960.

Joyal, André, Ross Street, and Dominic Verity. 1996. “Traced Monoidal Categories.” *Mathematical Proceedings of the Cambridge Philosophical Society* 119 (3): 447–68. https://doi.org/10.1017/S0305004100074338.

Lamport, Leslie. 1978. “Time, Clocks, and the Ordering of Events in a Distributed System.” *Communications of the ACM* 21 (7): 558–65. https://doi.org/10.1145/359545.359563.

Lawvere, F. William. 1973. “Metric Spaces, Generalized Logic, and Closed Categories.” *Rendiconti del Seminario Matematico e Fisico di Milano* 43: 135–66. https://doi.org/10.1007/BF02924844.

Lawvere, F. William, and Stephen H. Schanuel. 2009. *Conceptual Mathematics: A First Introduction to Categories*. 2nd ed. Cambridge University Press. https://www.cambridge.org/highereducation/books/conceptual-mathematics/00772F4CC3D4268200C5EC86B39D415A.

Leinster, Tom. 2014. *Basic Category Theory*. Cambridge University Press. https://doi.org/10.1017/CBO9781107360068.

Mac Lane, Saunders, and Ieke Moerdijk. 1994. *Sheaves in Geometry and Logic: A First Introduction to Topos Theory*. Universitext. Springer. https://doi.org/10.1007/978-1-4612-0927-0.

Mac Lane, Saunders. 1998. *Categories for the Working Mathematician*. 2nd ed. Vol. 5. Graduate Texts in Mathematics. Springer. https://doi.org/10.1007/978-1-4757-4721-8.

Mace, Jonathan, Ryan Roelke, and Rodrigo Fonseca. 2015. “Pivot Tracing: Dynamic Causal Monitoring for Distributed Systems.” In *Proceedings of the 25th Symposium on Operating Systems Principles*, 378–93. https://doi.org/10.1145/2815400.2815415.

OpenTelemetry Authors. 2026a. “OpenTelemetry Semantic Conventions: Trace Conventions.” Accessed August 30, 2026. https://opentelemetry.io/docs/specs/semconv/general/trace/.

———. 2026b. “OpenTelemetry Specification: Overview and Signals.” Accessed September 4, 2026. https://opentelemetry.io/docs/specs/otel/overview/.

Patterson, Evan, Owen Lynch, and James Fairbanks. 2022. “Categorical Data Structures for Technical Computing.” *Compositionality* 4 (5): 1–27. https://doi.org/10.32408/compositionality-4-5.

Pearl, Judea. 2009. *Causality: Models, Reasoning, and Inference*. 2nd ed. Cambridge University Press.

Pierce, Benjamin C. 1991. *Basic Category Theory for Computer Scientists*. Foundations of Computing. MIT Press. https://mitpress.mit.edu/9780262660716/basic-category-theory-for-computer-scientists/.

Reiter, Raymond. 1987. “A Theory of Diagnosis from First Principles.” *Artificial Intelligence* 32 (1): 57–95. https://doi.org/10.1016/0004-3702(87)90062-2.

Riehl, Emily. 2016. *Category Theory in Context*. Dover Publications. https://math.jhu.edu/~eriehl/context/.

Robinson, Michael. 2017. “Sheaves Are the Canonical Data Structure for Sensor Integration.” *Information Fusion* 36: 208–24. https://doi.org/10.1016/j.inffus.2016.12.002.

———. 2020. “Assignments to Sheaves of Pseudometric Spaces.” *Compositionality* 2 (2). https://doi.org/10.32408/compositionality-2-2.

Rosiak, Daniel. 2022. *Sheaf Theory through Examples*. MIT Press. https://mitpress.mit.edu/9780262542159/sheaf-theory-through-examples/.

Sambasivan, Raja R., Alice X. Zheng, Michael De Rosa, Elie Krevat, Spencer Whitman, Michael Stroucken, William Wang, Lianghong Xu, and Gregory R. Ganger. 2011. “Diagnosing Performance Changes by Comparing Request Flows.” In *8th USENIX Symposium on Networked Systems Design and Implementation*. https://www.usenix.org/conference/nsdi11/diagnosing-performance-changes-comparing-request-flows.

Schultz, Patrick, David I. Spivak, and Christina Vasilakopoulou. 2020. “Dynamical Systems and Sheaves.” *Applied Categorical Structures* 28: 1–57. https://doi.org/10.1007/s10485-019-09565-x.

Sigelman, Benjamin H., Luiz André Barroso, Mike Burrows, Pat Stephenson, Manoj Plakal, Donald Beaver, Saul Jaspan, and Chandan Shanbhag. 2010. “Dapper, a Large-Scale Distributed Systems Tracing Infrastructure.” Google. https://research.google/pubs/dapper-a-large-scale-distributed-systems-tracing-infrastructure/.

Spivak, David I. 2012. “Functorial Data Migration.” *Information and Computation* 217: 31–51. https://doi.org/10.1016/j.ic.2012.05.001.

———. 2014a. *Category Theory for the Sciences*. MIT Press. https://mitpress.mit.edu/9780262028134/category-theory-for-the-sciences/.

———. 2014b. “Database Queries and Constraints via Lifting Problems.” *Mathematical Structures in Computer Science* 24 (6). https://doi.org/10.1017/S0960129513000479.

Spivak, David I., and Robert E. Kent. 2012. “Ologs: A Categorical Framework for Knowledge Representation.” *PLOS ONE* 7 (1): e24274. https://doi.org/10.1371/journal.pone.0024274.

Vostokov, Dmitry. 2021. *Visual Category Theory Brick by Brick: Diagrammatic LEGO Reference*. OpenTask. https://www.dumpanalysis.org/visual-category-theory.

———. 2024a. *Accelerated Windows API for Software Diagnostics: With Category Theory in View*. 2nd ed., rev. 2.01 (December 2024). Republic of Ireland: OpenTask.

———. 2024b. *Theoretical Software Diagnostics: Collected Articles*. 4th ed. Republic of Ireland: OpenTask.

———. 2025. *Memory Dump Analysis Anthology*. Vol. 17. Republic of Ireland: OpenTask.

———. 2026. *Trace, Log, Text, Narrative, Data: An Analysis Pattern Reference for Information Mining, Diagnostics, Anomaly Detection*. 5th ed., rev. 5.06 (August 2026). Republic of Ireland: OpenTask.

W3C Distributed Tracing Working Group. 2021. “Trace Context.” W3C Recommendation. Accessed August 30, 2026. https://www.w3.org/TR/trace-context/.

Weiser, Mark. 1984. “Program Slicing.” *IEEE Transactions on Software Engineering* SE-10 (4): 352–57. https://doi.org/10.1109/TSE.1984.5010248.

Zeller, Andreas, and Ralf Hildebrandt. 2002. “Simplifying and Isolating Failure-Inducing Input.” *IEEE Transactions on Software Engineering* 28 (2): 183–200. https://doi.org/10.1109/32.988498.

Index

A

Abstract interpretation, 72, 126, 143
Adjunctions, 7, 40–42, 65, 117–118, 124, 139, 141, 143
Agent accountability, 83, 94–95, 101, 103, 131
Analysis patterns, 3, 11, 43, 59, 61, 64, 75, 138
Applied category theory, 7, 134, 137, 151, 153–156
Applied diagnostic cycle, 152
Approximate pasting, 28, 125, 143
Architecture, categorical foundations, 133, 154
Architecture, category-aware, 100
Artifact category, 14, 143
Associativity, 16, 58, 109, 111, 113, 143–144
Attribute map, 41, 117–118, 141, 143
Attributed C-sets (acsets), 153

B

Behavioral equivalence, 54–55, 134
Bidirectional transformations, 40, 133
Bisimulation, 55, 121, 134, 143
Black-boxing, 152–153

C

Cartesian Trace, 33, 141
Categorical databases, 37, 40, 114, 133–134
Categorical invariant, 8, 13, 24–26, 76, 81–83
Categorical Invariant Violation, 8, 26, 76, 81–83, 88, 136, 144
Category, 16, 113–114, 143–144
Category-aware diagnostics, 100
Causal inference, 8, 34, 67
Causal order, 67
Claim discipline, 155–156
Closure operator, 42, 118, 144
Coalgebra, 7–8, 54–55, 121, 133–134, 137, 141, 144
Coalgebraic behavior, 133–134, 154
Codiagnostics, 43, 138, 144
Colimits, 7, 37–38, 45, 116–117, 144, 152
Commutative diagrams, 24, 79, 83, 103, 111, 144, 149, 152–153
Composition, 7, 16, 18, 57, 113, 133, 135–136, 144, 152, 154–155
Compositional systems, 7, 133–135
Compositionality, 133, 152, 154–155
Conceptual emblem, 149–150
Context propagation, 7, 14, 23, 63, 67–68, 71, 128
Contextual truth, 52–53, 120, 142, 154
Correlation, 15, 19–20, 25, 33–34, 56, 138
Correlation witnesses, 35–36, 126–127, 144

D

DAG. See Trace-as-DAG
Debugging protocol, 84–86

D

Diagnostic architecture, 100–101
Diagnostic category, 143
Diagnostic invariant cards, 128–132
Diagnostic operads, 8, 59–60, 75, 85, 135–136, 138, 141
Diagnostic outcome, 26–27, 105
Diagnostic sufficiency, 72, 124–125, 145
Diagnostic topos, 52, 103, 136, 154
Diagnostics–Observability Square, 8, 64–65, 140, 145, 152
Diagram pasting, 26–28, 106, 125
Distributed traces, 66–67, 106

E

Equality policy, 28, 105–106, 111, 128, 144, 152–153
Equivalence relations, 11, 41, 55, 70, 109–110
Event graph. See Graphs
Evidence fusion, 37, 45
Evidence joins, 33, 134
Evidence provenance, 15, 35, 44, 100–102, 131, 138
Execution category, 8, 14, 63, 67, 136, 145

F

Faithful functors, 21, 70, 105, 115, 145
Fault localization, 7, 85–86, 135–136
Fiber (of an attribute), 41, 51, 117–118, 141, 143
Flag (pattern), 47, 141
Free category, 16–18, 20, 40, 84, 110, 113, 145
Functions, 108–109
Functoriality. See Functors
Functors, 21–23, 63–64, 105, 114–115, 145

G

Galois connection, 41–42, 118, 138, 141, 145
Galois Trace, 35, 41–42, 69, 118, 138, 141
General Analysis Patterns, 3, 75, 138
General Problem Patterns, 75
Geometric morphisms, 53, 121, 145, 154
Glued Activity, 34, 130, 141
Gluing, 50–52, 77, 82, 135, 145, 147–148
Graphs, 16, 40, 110, 117, 145
Grothendieck topology, 8, 103, 120, 142, 145, 148

H

Heyting algebra, 121, 146
Hom-set, 42, 106, 117–118, 138, 143, 145–146
Hypothesis category, 14–15, 146

I

Identities, 16, 109, 113, 142–146
Image factorization, 41, 141
Information loss, 55, 70, 83, 105, 145

I

Instrumentation, 8, 14, 21, 71, 77–79, 81, 102
Internal logic, 7, 120–121, 135, 146, 148
Invariant cards. See Diagnostic invariant cards

K

Kan extensions, 123, 146

L

Lax functors, 22, 123, 146
Limits, 7, 37, 116–117, 146, 152
Lipschitz bound, 125, 146
Local-to-global reasoning, 50, 52–53, 103, 120–121, 135, 152, 154
Logs, 56, 65–66, 104, 138–139

M

Message Cover, 41, 118, 141
Metamorphic testing, 98–99, 127, 146
Metrics, 8, 14, 21–22, 32, 59, 61, 63, 66
Missingness, 26, 32, 37, 44, 78, 98, 101, 105
Model synchronization, 133, 137, 154
Model-based diagnosis, 7, 146
Monads, 7, 25, 82, 119, 124, 146
Monoidal categories, 58, 60, 122, 135, 139, 146
Morphisms, 12, 56, 58, 75, 113, 122–123, 143–144, 146

N

NASA-HTTP artifact, 104
Natural transformations, 7–8, 30, 48, 64, 74–75, 115, 139, 146
Naturality residual, 31, 134, 147
Noncommutativity, 24, 147
Normalization, 8, 26–28, 32, 48, 67, 77, 83–84, 87

O

Objects, 12, 21, 37, 128–131
Observability, 63–65, 70–72, 147
Observability drift, 8, 70–71, 130, 136, 147
Observation functors, 21, 55, 63, 70–71, 74, 154
Observation Spaces, 8, 64, 137, 140, 147, 152
Observational coverage, 8, 52, 103
Open systems, 7, 134, 137, 153–155
OpenTelemetry, 7, 63, 66–67, 71, 101
Operads, 7–8, 59–60, 122, 135–136, 139, 141, 144–145, 147
Opposite category, 142, 144, 147

P

Partial order, 17, 49, 67, 91, 110, 112, 147
Pasting. See Diagram pasting
Pattern Category Theory, 8, 75–76, 103, 136, 140, 147
Presheaves, 47–48, 50–53, 119–121, 135, 138, 141, 147–148, 154
Products, 33, 37, 108, 116, 141, 146–147
Profunctors, 18, 22, 36, 127, 140, 147
Properads, 60, 138–139, 147
Property-based testing, 7, 98–99, 147
Provenance. See Evidence provenance

P

Pullbacks, 33–35, 37, 40, 61, 88–89, 116, 126, 129, 141, 147
Pushouts, 37–38, 116–117, 133, 144, 147

Q

Quotients, 11, 16, 41, 43–44, 55, 110, 116, 141

R

Relations, 12, 19, 98, 109
Residuals, 8, 24, 28, 31, 84, 98–99, 105, 129, 131, 148, 150
Restriction, 22, 47–49, 51, 119–120, 130, 135, 147–148
Root cause, 38, 53, 90, 102, 106, 120

S

Schema migration, 81, 93, 107, 130, 134
Sets, 107
Sheaf-based consistency, 7, 51, 120–121, 133, 135, 154
Sheafification, 53, 121, 148
Sheaves, 7, 50–53, 120–121, 131, 135, 141, 148, 154
Sieves, 53, 120–121, 145, 148, 154
Sites, 52, 103, 120, 135, 142, 148
Software architecture, 133, 154
Software semantics, 133, 135, 154–155
Sound overapproximation, 72, 126, 148
Spans, categorical, 35–36, 126–127, 148
Spans, tracing, 11, 14, 21, 24, 26, 69, 112, 148
Status classification, 105–106, 141
Structure Sheaves, 103, 141
Subobject classifier, 7–8, 53, 120, 135, 142, 148

T

Toposes, 7–8, 52–53, 103, 120–121, 133, 135–137, 141–142, 145–146, 148, 152, 154–155
Trace, 50, 56–58, 66–67, 106, 122
Trace context, 7, 33, 37, 67–68, 78, 110
Trace Field, 66, 74, 137, 139, 141
Trace Join, 11, 34, 139
Trace Plan, 78, 139, 152
Trace Presheaf, 51, 66, 135, 139
Trace spans. See Spans, tracing
Trace Viewpoints, 32, 73, 139
Trace Window, 47, 141
Trace-as-DAG, 66, 110
Typed missingness, 132, 148, 153

U

Universal properties, 25, 81, 108, 115, 117, 148

V

Visibility limits, 22, 64, 70, 112, 139, 148

W

Wiring diagrams, 134, 153, 155
World models, 73, 76, 101, 134

Y

Yoneda viewpoint, 122, 148