Contents

Overview

Four pages, read in order. They explain what computational causality is, the axiom DeepCausality is built on, the problem classes that become tractable once that axiom is in place, and the sixteen innovations the project ships to address them.

- **Why?** — what computational causality is, what it does, and where DeepCausality fits in.
- **Premise** — one working definition: causality as a spacetime-agnostic monadic process.
- **Problem** — seven categories of problems where dynamic causality is the right tool and conventional tooling is not.
- **Innovations** — the catalog of sixteen innovations that, taken together, set DeepCausality apart.

Once those land, move to Getting started for runnable code or Concepts for the type-level reference.

Why?

Computational causality and deep learning solve different problems. Deep learning excels at pattern matching from large data: object detection, fraud detection, next-token prediction in large language models. These systems infer from statistical co-occurrence in the training distribution, and that is exactly the right tool for many problems. Other problems need a different substrate: the ones where correlation is not causation and the deployment regime can shift away from training.

Computational causality is that substrate. It gives you three things a purely correlational system cannot:

- **Deterministic reasoning**: same input, same output. The system gives the same answer on Tuesday that it gave on Monday.
- **Probabilistic reasoning**: explicit odds, explicit confidence. The system tells you not just *what* it concluded but *how sure* it is.
- **Full explainability**: a logical line of reasoning. You can ask the system "why" and get a structured answer.

These properties matter in regulated and high-stakes domains: medicine, finance, robotics, avionics, industrial control. In those domains, a regulator, an operator, or a courtroom is going to ask why a decision was made, and an audit trail of deterministic reasoning becomes critical.

Deep learning and DeepCausality are complementary methodologies with different strengths that compose well in a single system. Deep learning is a strong choice for perception: object recognition, anomaly detection in raw signals, embeddings of unstructured text. DeepCausality is a strong choice for dynamic reasoning with a verifiable audit trail.

Consider a drone entering a tunnel. A deep-learning vision system handles perception: it recognizes the tunnel mouth from the camera feed. When GPS lock is lost a moment later, a classical finite state machine could switch to dead-reckoning, but only if that transition was anticipated at design time. Real flight encounters combinations the design-time state space does not enumerate: GPS loss *and* low battery *and* a passenger-corridor restriction *and* deteriorating wind, all at once. DeepCausality reasons over the propagating effect of those signals at runtime, asks the Effect Ethos whether the proposed fallback (e.g., immediate landing on a permitted surface) is permissible under the current operating rules, and emits an audit trail explaining the decision. A finite state machine cannot represent the permissibility check, cannot infer states that were not enumerated up front, and produces no audit trail beyond its transition

log. Each method plays to its strengths, and either can feed the other depending on what the system requires.

What makes DeepCausality unique

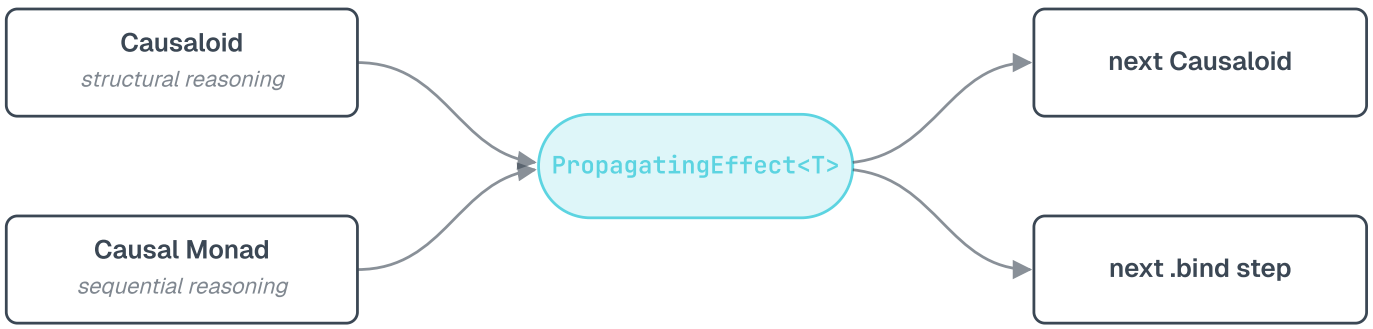
Most causality frameworks pick one paradigm. Pearl's Structural Causal Models pick a graph. Granger causality and the Rubin causal model pick a sequence. State-space models and control theory pick a process. Each is sound on its own ground, and each pays for the other paradigms through escape hatches, glue layers, or external orchestration. The price shows up when a real system needs the structural reasoning, the sequential reasoning, and the stateful threading in one inference path. Then you spend more time gluing models together than reasoning about the problem.

DeepCausality collapses those paradigms into one carrier: the propagating effect.

The library has two reasoning primitives that emit the same propagating-effect type:

- The **Causaloid** handles structural reasoning and composes isomorphic-recursively. A Singleton Causaloid, a collection of Causaloids, and a graph of Causaloids all nest into each other. A collection of causaloids nests into a singleton causaloid, which then becomes a node in a causaloid graph. The graph itself might be a node in a larger causaloid graph.

- The **Causal Monad** handles sequential reasoning. `pure` lifts a value into a chain; `bind` chains the next step; `intervene` rewrites a value mid-chain for counterfactual analysis. The chain accumulates an audit log automatically and short-circuits cleanly on error.



Because both primitives return the same propagating-effect type, you can take a Causaloid's verdict and `.bind` directly onto it. You can run a Causal Monad bind chain and feed its result into a Causaloid. The boundary between "structural reasoning" and "sequential reasoning" moves as

the problem evolves. Instead of picking one, two, or more frameworks and wiring them together, you just choose one framework and pick the modality relative to the problem you're solving.

The same carrier covers two reasoning regimes. The non-Markovian `PropagatingEffect<T>` is the simpler case where each step depends only on its inputs and rules. The Markovian `PropagatingProcess<T, S, C>` carries state and context through every step. Both are aliases of the same underlying 5-arity type, so promoting a non-Markovian chain into a Markovian one is one constructor call rather than a rewrite.

The avionics flight envelope monitor runs a Causaloid Collection over five sensor-health checks, a three-step Causal Monad bind-chain for state estimation, and a Causaloid hypergraph of six envelope protections, all threading through one `PropagatingProcess<T, FlightState, AircraftConfig>` with state and audit log carried across every stage. The same PropagatingEffect supports the physics, medicine, and distributed-systems examples in the repository.

The Causaloid alone gives you structure. The Causal Monad alone gives you sequencing. Neither would deliver the multi-domain composition the examples show. The fact that both emit the same type, that both can be lifted between Markovian and non-Markovian forms, and that they nest freely, is what makes DeepCausality unique.

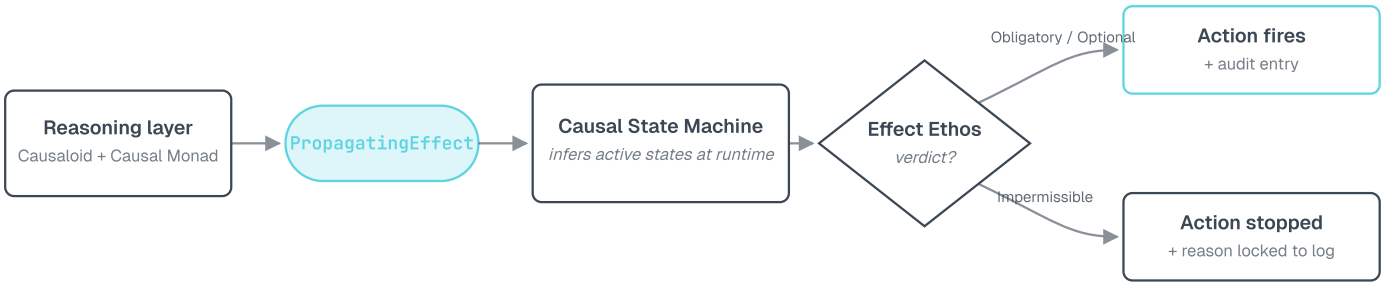
From reasoning to action

Reasoning composition is only half of the story. DeepCausality enables reasoning-based action with two further primitives:

- The **Causal State Machine (CSM)** is the bridge from inference to the outside world. It reads the propagating effect produced by the reasoning layer, evaluates which of its registered causal states have become active, and proposes the action linked to each active state. The state space is inferred at runtime rather than enumerated at design time, which is how the CSM avoids the limitation of a classical finite state machine.

- The **Effect Ethos** is a programmable safety guardrail above the CSM. Every action the CSM would otherwise fire is intercepted by the Ethos and evaluated against an immutable graph of computable norms. The Ethos returns a verdict: `Obligatory`, `Impermissible`, or `Optional` with an associated cost. Based on the verdict, the proposed CSM action may execute or get stopped out. When it gets stopped out, the reason from the Effect Ethos is logged together with the outcome, so that in a subsequent audit you can see why the

action was stopped, what its line of reasoning was, what the last proposed action was, and why it was stopped.



The combination matters most in the dynamic and emergent regimes. When the underlying reasoning is fully deterministic, the Ethos is unnecessary and the CSM can fire directly. When the causal structure itself evolves at runtime, static verification of the reasoning is no longer feasible, and the Ethos becomes the layer that restores verifiability at the action boundary. Whether the inference behind a proposed action was statically provable or emerged at runtime, the action only leaves the system if the Ethos says it is permissible under its encoded ethos.

Where to go from here

DeepCausality treats causality itself as a dynamic process. The next page explains what that means in practice, and the page after that gives you the single idea on which the library is built.

Further reading

Conceptual deep-dives in this documentation:

- Causaloid: structural reasoning primitive
- Causal Monad: sequential reasoning primitive
- Effect propagation process: the `PropagatingEffect` carrier
- Causal State Machine: runtime-inferred state with action proposal
- Effect Ethos: programmable permissibility layer
- Dynamic causality: how the substrate handles regime change

Background on why correlation-based systems break down in practice:

- Why is correlation not causation?: the four mechanisms
- Why is distribution shift a problem in AI?: silent failures on out-of-distribution inputs

- Why correlation breaks under regime change: sign reversal, magnitude collapse, spurious appearance
- Why do LLMs struggle with causality?: Pearl's hierarchy and where token predictors sit
- Why LLMs can't do physics: the structural inversion DeepCausality applies

External references on the foundational frameworks DeepCausality unifies:

- Pearl's Structural Causal Models (book)
- Granger causality
- Rubin causal model
- State-space representation

Premise
A different premise

DeepCausality starts from a different premise than classical causality.

Classical causality is rooted in a static space-time assumption that traces back to Seneca's definition of causality, formulated approximately two thousand years ago. In the meantime, contemporary science has advanced in foundational fields such as quantum physics, general relativity, and quantum physics, where the fixed space-time assumption simply does not hold any longer. Causal rules can evolve. Spacetime itself can curve. The background against which "cause precedes effect" becomes dynamic.

DeepCausality responds by rooting itself in **Whitehead's process philosophy**, which shifts the Aristotelian assumption of a static snapshot in time toward a dynamic *process of becoming*. The project then adapted the essence of process philosophy into a **spacetime-agnostic dynamic causal process**. The theoretical foundation deserves its own book one day; the Effect Propagation Process preprint is the long form of the underlying premise.

The axiom

That premise is captured in a single foundational axiom, condensed into the following working definition:

> Dynamic Causality is the spacetime-agnostic monadic process in which one propagating effect is obtained from another by applying a causal function within the monad:
>
> $m_2 = m_1 \gg\!= f$

That is a dense phrase. Let's unpack it:

- **Monadic process**: a propagating effect is a type alias over an arity-5 monad that carries state, context, error, and an audit log alongside the value. The monad laws (left identity, right identity, associativity) guarantee that the carrier's bookkeeping is threaded through every step automatically, which is what gives the chain its end-to-end explainability.

- **Functional dependency**: each propagating effect is obtained from the previous one by applying a causal function `f` within the monad: $m_2 = m_1 \gg\!= f$. The function consumes one propagating effect and emits the next. Chained, those steps form a process of effect

propagation. Therefore, the key mechanism of dynamic causality is the effect propagation process.

- **Spacetime-agnostic**: time and space are not built into the relation. They are data the causal function reads from a context, the same way it reads any other input it needs to compute its result.

- **Explicit context**: because spacetime is not built in, anything time-like or space-like has to live in an explicit Context. This makes the embedded causal function independent of any specific spacetime, so you can encode Euclidean space, Minkowski or Lorentzian spacetime, and anything in between.

A useful intuition is a ripple in a pond. One ripple is an effect. It propagates outward and produces the next ripple. DeepCausality is a framework for defining how those ripples spread, what each one carries, and what happens when the rules for spreading themselves change.

For more details on the axiom and the properties it unlocks, see The Axiom.

Where to go from here

Once you have the axiom, the rest of the library is the operational machinery that makes it computable:

- The Causaloid is the unit that wraps the causal function.
- The Context is the hypergraph that holds the world the function reads from.
- The Effect Ethos is the safety layer for the cases where the causal rules themselves evolve.
- The Dynamic causality page is the technical entry point: four reasoning modalities, the philosophical move behind them, and the adaptability-versus-verifiability trade-off.

For the formal treatment, see the Effect Propagation Process preprint.

Problem

Classical computational causality, pioneered by Judea Pearl and others, is powerful, well-validated, and covers a large class of useful problems. However, classical computational causality is rooted in three assumptions that prevent its application to dynamic systems.

The classical way of thinking

Imagine a simple thermostat:

- **Cause**: room temperature drops below 68°F.
- **Effect**: the furnace turns on.

A classical model captures this because three things hold:

1. **Time is a straight line.** The temperature drops *before* the furnace turns on. There is a clear "happens-before" relationship.
2. **The causal rules are fixed.** "If temperature < 68, turn on the furnace" is the same rule tomorrow as it is today.
3. **Context is implicit.** Whatever the thermostat does not measure is absorbed into the background.

Most of classical computational causality, from Pearl's Structural Causal Models to Granger's time-series analysis, lives inside them.

Where the assumptions break

Now imagine a financial trading system, or a fleet of autonomous wildfire-fighting drones:

1. **Time is not a straight line.** A trading system observes events on nanosecond scales, but its decisions depend on the hourly high, yesterday's close, and the day's volume. Time becomes multi-layered and multi-scaled.
2. **The rules can change.** During a normal market day, "low interest rates push stock prices up" is a workable rule. During a crash, that rule breaks and "high fear pushes every asset down" takes over. The causal relationships in the system have changed mid-flight.
3. **Context changes continuously.** An autonomous drone navigating by GPS works fine until it enters a tunnel and loses signal. The computer vision system saw the tunnel coming, but if context is implicit, there is nowhere to put that fact and nothing to do with it.

The third point is the critical one. When the context changes, the rules can change. When the rules can change, you need a framework that treats both as first-class moving parts.

What dynamic causality enables

When causality is treated as a dynamic process rather than a static graph, the problems classical causality cannot reach come back into scope. DeepCausality is built around exactly this premise: an explicit Context that holds the world, a Causaloid that can be a singleton or a hypergraph for arbitrary structure, a Causal Monad for sequential composition with first-class intervention, an Effect Ethos that verifies actions when the reasoning itself can no longer be statically verified, and a deployment surface that runs on Tokio for production.

This page first walks through the problem categories that become tractable once the static assumption is dropped. Each category is a problem class where a real practitioner today either picks two or three separate libraries and writes glue forever, or ends up writing a custom solution because nothing else worked. DeepCausality was built from the ground up for the dynamic case, which enables a number of use cases that conventional tooling cannot reach. Where you find yourself on the list is roughly the answer to whether DeepCausality is the right tool for the work you are doing.

1. Dynamic control systems

Classical control theory does well when the plant model is fixed. A PID controller for a thermostat, a Kalman filter for an inertial measurement unit, a model-predictive controller for a refinery column. The industry has spent decades building qualified toolchains around exactly this case. Simulink Embedded Coder, SCADE Suite, Ansys SCADE, certified compilers under DO-178C and DO-330. Each of them translates a fixed plant model into proven-correct generated code, and the regulatory trajectory points toward more of that, not less. Static control is largely a solved problem and is not the gap DeepCausality fills.

Dynamic control is the gap. A loop whose plant model evolves with the operating regime, whose causal structure rewires when the environment changes, whose adaptation cannot be captured as a parametric gain schedule, has no settled answer. Adaptive wind turbine control under shifting atmospheric stratification. Power grid management as the renewable mix oscillates intraday. Adaptive control surfaces on a vehicle whose aerodynamics shift with payload, fuel state, or damage. The plant itself is dynamic, and the rules that govern it are moving targets that no code-gen pipeline can pin down ahead of time.

DeepCausality is built for exactly this case. The four reasoning modalities (static, dynamic, adaptive, emergent) map onto a set of control problems. The Context primitive carries the operating regime explicitly, so a regime change becomes a Context update rather than a model rebuild. The Causaloid graph can be reshaped at runtime when the structure of the plant changes. The Causal State Machine proposes control actions whose state combinations were not enumerated at design time, and the Effect Ethos checks each action against an immutable graph of computable norms before it actuates.

2. Multi-physics and multi-regime simulation

Some simulations cross a single mathematical regime: a tensor solver for fluid flow, a manifold integrator for orbital mechanics, a Clifford-algebra rotor for rigid-body dynamics. Each one is well served by an existing library. The harder problems cross regimes mid-pipeline. A relativistic plasma that needs Newtonian magnetohydrodynamics in the weak-field zone, full general relativity near a compact object, and breaks down at the event horizon where the physics itself stops being defined. A multi-scale climate model that switches from cloud-resolving physics to large-eddy simulation to global circulation along one atmospheric column. A weld pool that goes from solid mechanics to viscous flow to free-surface thermodynamics in a few millimeters of travel.

The conventional alternative is to stitch separate solvers together across language boundaries. General-relativity code in C++, plasma code in Fortran, orchestration in Python, regime transitions in a hand-written switching layer that is hard to debug. The handoffs leak precision, lose audit trails, and obscure where the model actually crosses from one regime into the next.

DeepCausality lets you write the physics the way you would write it on paper, regime changes included. The `grmhd` example is a five-step `bind` chain that crosses two regime boundaries inside one closure. Step 1 builds the Schwarzschild metric and the Einstein tensor in tensor algebra. Step 2 inspects the local curvature and selects the metric signature for what comes next: Minkowski for the relativistic regime, Euclidean for the classical regime. That is the first regime transition, captured as a single closure inside the chain. Step 3 leaves tensor algebra entirely and computes the Lorentz force as a Clifford bivector in the metric just chosen dynamically in the previous step. Step 4 returns to tensor algebra and contracts the electromagnetic field tensor with the spacetime metric to produce the stress-energy tensor that feeds back into Step 1. Step 5 is a stability analysis that decides whether the simulation is still physically meaningful or whether it is approaching a singularity where the physics itself stops being defined. That is the second regime boundary. Two regime changes, four mathematical domains, one `bind` chain that reads top to bottom like the derivation.

The simulation reads the way the physicist thinks about it: step one, two, three, regime change, step four, five, regime boundary.

Precision is one type alias for the whole pipeline; flip `f64` to `Float106` and composition drift drops from ~10^{-16} to ~10^{-31} on the capstone spinor example. The algebraic trait hierarchy enforces associativity, commutativity, and distributivity at compile time, so an algorithm that requires associativity cannot silently accept a type that violates it.

3. Financial systems under regime change with compliance pressure

Trading systems where “low rates lift stocks” inverts during a credit crisis. Risk engines that must explain why they sized a hedge the way they did. Anti-fraud platforms where the adversary evolves faster than the model. The rules that govern the system are themselves moving targets, and compliance asks for explanations that survive the next regime.

The Context primitive is a typed hypergraph mutated in place across the lifetime of a run, so changing the regime does not require rebuilding the model. The Adjustable trait absorbs sensor drift, missing prints, and calibration deltas without losing history. Counterfactual analysis is a first-class operation: `intervene` rewrites a value mid-chain, and `extra_contexts` carry parallel hypothetical worlds for what-if replay. The propagating effect carries the audit log alongside the value, so the compliance report comes out of the same pipeline that produced the decision.

A typical trading stack hard-codes the regime in static configuration. When the regime breaks, someone rewrites the config. A counterfactual replay tool is usually a separate offline pipeline that approximates the production engine. DeepCausality runs the same engine in both modes, so the replay is the production behavior with one input swapped.

4. Scientific discovery

Turbulence forecasting at the MIT Aerofluids, Learning & Discovery Lab. Computational biology pipelines that screen for drug targets. Materials science loops that propose alloys and feed the proposals back into a simulator. The structure of the problem is part of the answer; discovery and inference are two halves of the same workflow, and shipping a model that took six months to discover usually means rewriting it in a serving language.

The Causal Discovery Language is a typestate-builder DSL that hosts two discovery algorithms as compile-time-isolated pipelines: SURD decomposes how variables drive a target, and BRCD ranks which variable’s mechanism changed across a normal and an anomalous regime. SURD,

MRMR, and BRCD live in `deep_causality_algorithms` and feed the pipeline directly. The output of the final stage is a `Causaloid` indistinguishable from a hand-written one, so the discovered model feeds the rest of the framework with no translation step. `Uncertain<T>` keeps noisy measurements honest end to end. Discovery in Python lands in a notebook. Productionizing it means a rewrite into C++ or Java behind a serving framework. DeepCausality closes the gap between the discovery stage and the inference stage because both can be built in the same ecosystem.

5. Autonomous systems in open environments

The Causal State Machine infers active states from the propagating effect rather than enumerating them at design time, so combinations the designer did not foresee still produce sensible actions. Emergent causality lets the reasoning graph rewire itself in response to context. The Effect Ethos restores verifiability at the action layer when verifiability at the reasoning layer is no longer feasible. Every action goes through the deontic check. Reasoning is free to be emergent. Actions are not.

A classical finite-state machine cannot represent the compound condition “GPS loss and low battery and a passenger-corridor restriction and deteriorating wind” unless someone enumerated it at design time. A learned policy has no permissibility layer. A static rule engine cannot rewire itself when the world changes shape. DeepCausality is the substrate that lets reasoning be emergent while keeping actions verifiably safe.

6. The intersection that needs almost everything

Some problems sit at the intersection of multiple categories above. An avionics flight envelope monitor is at once a real-time safety-critical control loop, a multi-physics computation over sensor fusion, an open-environment autonomous system whose state space cannot be enumerated, and a regulated artifact that an authority will audit. Each axis would individually be hard. Together they are intractable without a unifying substrate.

The same propagating effect carries a tensor verdict from a physics stage, a state from a Markovian step, an uncertain reading from a noisy sensor, and an audit log from every prior step, all under one composition law. The CSM proposes actions whose state combinations no one enumerated. The Effect Ethos checks each action against the permissibility graph. DeepCausality provides one carrier and one composition law that cover the entire pipeline.

7. Dynamic emergent causality (frontier)

A class of problems sits beyond what production engineering currently handles. The causal structure of the system is itself co-evolving with a dynamic context. Examples include long-running closed-loop control of self-modifying biological systems, multi-agent ecologies where the agents redesign each other, and certain frontier safety questions about AI systems whose internal causal graph rewires in response to inputs. The central problem is: “how do we even formalize it?”

Emergent causality is the experimental fourth modality in DeepCausality. New Causaloids and new edges can be introduced by a generative process at runtime. The Causaloid graph is allowed to take shapes no upfront proof can foresee. The Effect Ethos is what makes the experiment safe to run: reasoning evolves freely, but every action is checked against an immutable ethos of computable norms before it leaves the system. The mode is deliberately experimental, and the work is currently confined to research.

Where to go from here

The next page gives you the single axiom on which all of this is built. If you recognized your problem in one of the seven categories above, the rest of the documentation describes the primitives in detail; the Concepts index is the entry point. If you did not recognize your problem in any of them, the conventional Pearl, Granger, or Rubin tooling probably already covers what you need, and DeepCausality would be more substrate than the work asks for.

Innovations

DeepCausality is built on a single axiomatic foundation introduced earlier in this section and addresses the problem classes discussed on the previous page. This page covers seventeen distinct innovations that, taken together, form a coherent substrate for dynamic causality.

The innovations are grouped into seven sections in logical order: foundation, mathematical substrate, causal discovery from data, causal modeling, causal inference, action, and production deployment. Each entry is brief by design; the Concepts section elaborates every primitive in full.

I. Foundation: the axiom

Causality has to be defined before it can be computed. DeepCausality picks one definition, low enough in the stack that the classical methods drop out as special cases.

1. A spacetime-agnostic causal axiom. Causality is generalized as a monadic functional dependency, captured by `m₂ = m₁ >>= f`. Pearl SCMs, dynamic Bayesian networks, Granger causality, the Rubin causal model, and conditional average treatment effects all drop out as parametric specializations of that same axiom; the classical causality examples implement each one directly. *See Axiom for the working definition.*

II. The mathematical substrate

Once the axiom is set, it needs a numerical floor that the rest of the library stands on. Four innovations cover that floor end to end.

2. A uniform mathematical surface. Tensors, multivectors, manifolds, sparse matrices, and propagating effects all implement the same `Functor`, `Monad`, and `CoMonad` surface. `fmap`, `bind`, `extend`, and `extract` mean the same thing on every container, which is how cross-domain pipelines stay readable and how bridge code disappears. *See Uniform Math.*

3. An explicit algebraic trait hierarchy. `Magma → Semigroup → Monoid → Group → AbelianGroup`; then `Ring → CommutativeRing → Field → RealField → ComplexField<R>`; plus `Module<R>`, `Algebra<R>`, `AssociativeAlgebra<R>`, `DivisionAlgebra<R>`, and `EuclideanDomain`. Marker traits move algebraic laws into the type system, so an algorithm that requires associativity cannot accept a type that violates it. *See Uniform Math.*

4. Higher-Kinded Types via the witness pattern. Rust does not have native HKTs. `deep_causality_haft` fills the gap with zero-sized witness structs that stand in for type constructors, so the Causal Monad's `bind` is written once generically and specialized per concrete instantiation through monomorphization. No boxed type erasure, no virtual calls. *See Higher-Kinded Types.*

5. Precision as a parameter. Because every math container stands on the generic algebraic floor, numerical precision becomes one type alias for the whole pipeline. `pub type FloatType = Float106;` flows through every tensor contraction, multivector rotation, manifold extension, and monadic step. The Multi-physics and multi-regime simulation section on the Problem page walks through what this earns in practice. *See Uniform Math.*

III. Discovery: from raw data to a model

A project rarely starts with a finished causal model. It starts with data, and the first job is to find the structure worth running inference against. Two innovations cover the path from observations to an executable model.

6. The Causal Discovery Language. CDL is a typestate-builder DSL that hosts two discovery algorithms as compile-time-isolated pipelines, driven by one explicit config. The typestate enforces stage order and algorithm isolation at compile time. SURD, MRMR, and BRCD ship as discovery primitives: SURD reports which variables are uniquely or synergistically causal (and flags redundant ones), while BRCD ranks the root cause of a regime shift across a normal and an anomalous dataset. *See Causal Discovery Language.*

7. Uncertainty as a first-order type. `Uncertain<T>` wraps a value with the distribution that produced it and uses the Sequential Probability Ratio Test for confidence-bounded decisions. `MaybeUncertain<T>` separates presence from distribution, so missing readings propagate explicitly rather than silently. *See Uncertainty.*

IV. Modeling: the primitives that hold causal structure

With data on hand, four primitives carry the model. The first two share one premise: cause and effect are folded into one entity. They emit the same propagating effect and compose freely with each other.

8. The Causaloid as an isomorphic-recursive unit. A Singleton, a Collection, and a Hypergraph all implement the same `Causable` and `MonadicCausable` trait surface, so they nest into each other to arbitrary depth. Pick the structure the problem demands at every level and freely compose. *See Causaloid.*

9. A state-threading monad with first-class intervention. The carrier effect implements the `CausalMonad` trait: `pure`, `bind`, and `intervene`. `bind` threads Markovian state through the chain; `intervene` implements Pearl's `do()` operator mid-chain, putting counterfactual reasoning in the same engine that runs factual reasoning. The three monad laws (left identity, right identity, associativity) are test-covered. *See Causal Monad.*

10. The explicit Context as a typed hypergraph. A typed weighted hypergraph of `Contextoids` mutated in place across a run, carrying data, space, time, spacetime, and symbolic payloads. Counterfactual analysis comes through parallel `extra_contexts` evaluated against the same Causaloid without disturbing the primary one. *See Context.*

11. The Adjustable trait. `update` replaces the stored value outright with a default that rejects the type's zero sentinel; `adjust` applies a correction relative to the stored value with a default that rejects negative results. Both are const-generic over grid dimensions, so the same trait covers scalar, 2D frame, 3D volumetric, and 4D spacetime corrections. *See Context.*

V. Inference: the carrier and how it grows

Three innovations cover how data actually flows through the model, and how you write the pipelines that drive it.

12. The Propagating Effect as a unified carrier. `CausalEffectPropagationProcess<Value, State, Context, Error, Log>` is the load-bearing five-field record, and it implements the Causal Monad trait directly. Every other primitive in the library (Causaloid, Context, CSM, Effect Ethos) exchanges work through this one type, and the audit log accumulates automatically across every step. *See Effect Propagation Process.*

13. Non-Markovian and Markovian under one type. `PropagatingEffect<T>` fixes state and context to `()`. `PropagatingProcess<T, S, C>` keeps them generic. Both are aliases of the same struct, so lifting from one form into the other is a single constructor call. Start non-Markovian and upgrade the carrier the moment state becomes necessary. *See Effect Propagation Process.*

14. The Causal Flow DSL. `CausalFlow` is a fluent facade over the Causal Monad that reads a pipeline as a sequence of verbs: `value` and `process` to seed, `map` and `try_step` and `next` to step, `branch` to route, `iterate_n` / `iterate_until` / `iterate_to_fixpoint` to loop, and `intervene` for a mid-pipeline `do()`. Every verb lowers to `pure` or `bind`, so the DSL adds reading clarity rather than new semantics, and the monad laws still hold. It hides the `EffectValue` wrapping and the manual error short-circuit, and it is the surface the library's own examples are written on. *See Causal Flow.*

VI. From inference to safe action

A causal verdict on its own does nothing. Two innovations bridge inference to the outside world while keeping the system safe to deploy.

15. The Causal State Machine. A thread-safe registry of `(CausalState, CausalAction)` pairs where the active state space is inferred at runtime from the propagating effect rather than enumerated at design time. *See Causal State Machine.*

16. The Effect Ethos. Every action the CSM proposes is intercepted and evaluated against a graph of `Teloids` under DDIC (Defeasible Deontic Inheritance Calculus from Olson, Salas-Damian, and Forbus). Conflict resolves through Lex Posterior, Lex Specialis, Lex Superior. Reasoning is free to be emergent; actions are not. *See Effect Ethos.*

VII. Production deployment

Once the model is approved for production, it has to serve traffic.

17. Native async serving on Tokio. DeepCausality is a Rust library. The CSM is thread-safe by construction (`Arc<RwLock< ... >>`) so it embeds natively inside any async runtime. A production deployment with Tokio comes down to embedding the causal model into a regular request handler.

Where to go from here

The Concepts section elaborates each primitive in detail. For a worked example of how the seventeen innovations compose in one pipeline, the Why DeepCausality page walks through the avionics flight envelope monitor end to end. For physics, see the Multi-physics section on the Problem page, which walks through the GRMHD example as a five-step `bind` chain.

Literature

This page collects the main publications that shaped DeepCausality, grouped by the contribution each one made to the framework. It is adapted from the Precedent section of the EPP monograph.

Whitehead and Bergson: process philosophy

The EPP's primary departure point is a rejection of the classical Newtonian conception of a static, absolute background spacetime. This move is rooted in the tradition of process philosophy, which argues that reality is not composed of enduring, static substances but is a dynamic flow of interconnected events. The idea finds its clearest expression in the work of Alfred North Whitehead, who posited a universe of "actual occasions" (Whitehead, *Process and Reality*), and Henri Bergson, who described reality as a continuous "creative evolution" (Bergson, *Creative Evolution*). Their shared insight of reality as a process inspires the EPP's redefinition of causality itself, shifting from a static, happens-before relation to a dynamic process of effect propagation.

Moggi and Wadler: monadic composition

The carrier's `CausalMonad` trait and its `bind` operation inherit a programming discipline whose theoretical move belongs to Eugenio Moggi and whose practical articulation belongs to Philip Wadler. Moggi proposed in 1989 that monads from category theory provide a useful structuring tool for the denotational semantics of programming languages (Moggi, *Computational lambda-calculus and monads*). His key move was distinguishing a value type a from a computation type `M a`, and showing that monads can encapsulate features such as state, exception handling, and continuations within a single uniform interface. Wadler then demonstrated in 1992 that this insight transfers directly into a programming discipline (Wadler, *The essence of functional programming*). A function of type `a → b` can be lifted into monadic form `a → M b`, and the resulting programs can gain error handling, state, output, or non-deterministic choice by changing the monad while leaving the program structure essentially intact.

The EPP adopts this discipline for causal composition. The monadic axiom $m_2 = m_1 \gg\!= f$ is the Kleisli composition of a monad whose context parameters carry the state, configuration, error condition, and audit log of the causal process. The flexibility Wadler showed for lifting plain functions into a context-aware monadic form is the same flexibility with which the carrier effect,

which implements the `CausalMonad` trait, interoperates with the Causaloid through the shared `PropagatingEffect` type.

Russell: a critique of causality

Bertrand Russell's critique of causality, formulated in his 1912 essay *On the Notion of Cause* (Russell, 1912), led to the realization that the central issue is not necessarily causality itself, but the underlying assumption of time asymmetry. That assumption sits at odds with Russell's observation that most successful theories in physics are based on time symmetry. While physics routinely models dynamic change in complex systems, computational causality consistently struggles to capture dynamic causality. There is truth to causal invariance, yet dynamic systems also emit different causal structures depending on dynamic change, and that is where computational causality is at odds with physics and, to an extent, with reality. From there, it became clear that for causality to handle dynamics, it requires a new foundation of causality itself.

Hardy: the causaloid

Lucien Hardy introduced the "causaloid" (Hardy, 2005), a concept that encapsulates a spatial region and the causal connections within it, as a foundation for his work on a theory of quantum gravity. Unlike all prior forms of causality, Hardy's causaloid is spacetime-agnostic because it folds cause and effect into one entity and removes the need for temporal order. His seminal work *Probability Theories with Dynamic Causal Structure* had a three-fold impact on the EPP. First, his causaloid formalism proved instrumental in the formation of isomorphic-recursive causal data structures. Second, his insight that the formalism puts deterministic and probabilistic structures on equal footing led directly to the multi-modal reasoning of the EPP. Third, his demonstration that fundamental differences of theoretical foundations are contained in a causaloid informed the representation of causal relations as a causal function, which resulted in the single axiomatic formulation of the EPP.

Pearl: the structural causal model

Judea Pearl, with his Structural Causal Model, established the foundation upon which the entire field of computational causality was subsequently built. His work in *Causality: Models, Reasoning, and Inference* (Pearl, 2000) was as influential as his later critique in *Theoretical Impediments to Machine Learning with Seven Sparks from the Causal Revolution* (Pearl, 2018).

His contribution to the algorithmization of counterfactuals proved instrumental for the development of contextual counterfactuals in the EPP.

Bareinboim: transportability of causal effects

Bareinboim's calculus of transportability (Bareinboim and Pearl, 2012) and his subsequent work on data fusion formalize the very problem of contextual variance that the EPP's explicit Context is designed to manage at a computational level. Where Bareinboim provides the logical framework for reasoning about moving causality between discrete contexts, the EPP provides the computational primitive, a dynamic, queryable, multi-modal Context, to operationalize that reasoning.

Forbus: a defeasible deontic calculus

Kenneth Forbus's work on formalizing deontic calculus (Olson, Salas-Damian, and Forbus, 2024) proved invaluable for the problem of conflicting norms in the Effect Ethos. In practice, it is rarely possible to write conflict-free norms, and a recurring theme during the development of the Effect Ethos was the acceptance of that reality. The search for a solution led to the adoption of the Defeasible Deontic Calculus as the primary means to resolve normative conflicts.

Bornholt: an uncertain type

The contribution of Bornholt and colleagues in *Uncertain<T>: A First-Order Type for Uncertain Data* (Bornholt, Mytkowicz, and McKinley, 2014) informed the unification of deterministic and probabilistic reasoning in the EPP. Instead of representing a value with a single number, the Uncertain type represents a value with a full probability distribution, or even a computation graph that produces one. The EPP reasoning logic can lift simpler deterministic and probabilistic effects into the Uncertain distribution, aggregate the distributions, and infer a logical combination of all inputs without loss of information. The final output collapses rich uncertainty into a single value at the last moment while preserving second-order properties such as the standard deviation or confidence level. Decisions, and crucially deontic reasoning, become more robust under uncertainty.

Zhao et al.: feature selection for discovery

The Causal Discovery Language draws its feature-selection stage from Maximum Relevance and Minimum Redundancy (mRMR), the filter method Zhao, Anand, and Wang developed for Uber's marketing machine-learning platform (Zhao, Anand, and Wang, 2019). mRMR selects the features most relevant to a target while controlling redundancy among the selected set. That balance is what lets the CDL reduce a large observational feature space to a compact, informative subset before any causal structure is inferred, which keeps the downstream discovery step both tractable and interpretable.

Martínez-Sánchez et al.: causality by states

The causal-discovery algorithms build on SURD, the decomposition of causality into synergistic, unique, and redundant components by Martínez-Sánchez and Lozano-Durán (Martínez-Sánchez and Lozano-Durán, 2025). Their state-and-interaction-type formulation quantifies causal influence as a function of system state and separates redundant from synergistic interactions, rather than reporting a single average causal strength. The `deep_causality_algorithms` crate implements this as a discovery method, which suits dynamic causality directly: causal structure that varies with the state of the system is exactly what the EPP is built to represent.

Liu et al.: hypergraph analytics

UltraGraph, the two-phase hypergraph data structure that backs the Causaloid Graph and the Context Hypergraph, is built on the hypergraph-analytics work of Liu, Firoz, Gebremedhin, and Lumsdaine (Liu et al., 2022). Their NWHy framework for hypergraph representations, data structures, and algorithms informed UltraGraph's separation of a mutable construction phase from an immutable, cache-friendly analysis phase. That split is what makes performance-constrained hypergraph composition practical inside the EPP, wherever the Causaloid Graph or the Context grows large.

References

- Whitehead, A. N. *Process and Reality*. Simon and Schuster, 2010 (originally 1929). PDF
- Bergson, H. *Creative Evolution*. Routledge, 2022.
- Moggi, E. "Computational lambda-calculus and monads." *Proceedings of the Fourth Annual Symposium on Logic in Computer Science*, IEEE Computer Society, 1989, pp. 14–23. ACM · PDF

- Wadler, P. “The essence of functional programming.” *Proceedings of the 19th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, ACM, 1992, pp. 1–14. doi.org/10.1145/143165.143169
- Pearl, J. *Causality: Models, Reasoning, and Inference*. Cambridge University Press, 2000. PDF
- Pearl, J. “Theoretical Impediments to Machine Learning with Seven Sparks from the Causal Revolution.” 2018. arxiv.org/abs/1801.04016
- Bareinboim, E., and Pearl, J. “Transportability of Causal Effects: Completeness Results.” *Proceedings of the AAAI Conference on Artificial Intelligence*, vol. 26, no. 1, 2012, pp. 698–704. ACM · PDF
- Olson, T., Salas-Damian, R., and Forbus, K. D. “A Defeasible Deontic Calculus for Resolving Norm Conflicts.” 2024. arxiv.org/abs/2407.04869
- Russell, B. “On the Notion of Cause.” *Proceedings of the Aristotelian Society*, vol. 13, 1912, pp. 1–26. doi.org/10.1093/aristotelian/13.1.1
- Hardy, L. “Probability Theories with Dynamic Causal Structure: A New Framework for Quantum Gravity.” 2005. arxiv.org/abs/gr-qc/0509120
- Bornholt, J., Mytkowicz, T., and McKinley, K. S. “Uncertain<T>: A First-Order Type for Uncertain Data.” 2014. microsoft.com/en-us/research
- Zhao, Z., Anand, R., and Wang, M. “Maximum Relevance and Minimum Redundancy Feature Selection Methods for a Marketing Machine Learning Platform.” *2019 IEEE International Conference on Data Science and Advanced Analytics (DSAA)*, 2019. IEEE · arXiv
- Martínez-Sánchez, Á., and Lozano-Durán, A. “Observational causality by states and interaction type for scientific discovery.” 2025. arxiv.org/abs/2505.10878
- Liu, X. T., Firoz, J., Gebremedhin, A. H., and Lumsdaine, A. “NWHy: A Framework for Hypergraph Analytics: Representations, Data Structures, and Algorithms.” *2022 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, 2022, pp. 275–284. IEEE

Getting started

Six short pages. Install first, then five “hello, X” programs that each isolate one moving part. By the end you have written, evaluated, and composed every moving part the library is built on.

- **Install** — add DeepCausality to a Rust project.
- **Hello, Causal Flow** — the high-level DSL that reads causal reasoning as a pipeline. The clearest place to start.
- **Hello, Causal Monad** — the `pure` and `bind` engine the flow is built on. Walk a value through a three-step chain and look at what flowed.
- **Hello, Causaloid** — build, evaluate, and compose Causaloids in the smallest possible program.
- **Hello, Context** — build a Context hypergraph and let a Causaloid read from it.
- **Hello, Effect Propagation** — how a Causaloid’s structural reasoning and a bind-chain compose over one shared carrier effect.

For the conceptual model behind the code, see Concepts.

Install

DeepCausality is a workspace of twenty independently published crates. Pick what you need.

Prerequisites

DeepCausality targets the Rust 2024 edition:

```
rustup update stable
rustc --version
```

The umbrella crate

Most users start here:

```
cargo add deep_causality
```

The `deep_causality` crate re-exports the user-facing types: `Causaloid`, `CausaloidGraph`, `Context`, the propagating-effect machinery, and the surrounding aliases. It pulls in `deep_causality_core`, `deep_causality_haft`, `deep_causality_uncertain`, `deep_causality_ast`, `deep_causality_data_structures`, and `ultragraph` transitively, so a single `cargo add` is enough for a first project.

Specialized crates

Reach for one of these when the umbrella is broader than you need:

- `deep_causality_algorithms`: MRMR feature selection, SURD information decomposition.
- `deep_causality_data_structures`: sliding-window and grid-array containers, useful on stream workloads.
- `deep_causality_discovery`: the Causal Discovery Language (CDL) pipeline.
- `deep_causality_ethos`: the Effect Ethos and its Teloids.

- deep_causality_topology, deep_causality_physics, deep_causality_multivector: math and physics primitives.
- deep_causality_tensor, deep_causality_sparse: numerical containers.
- deep_causality_uncertain: a first-order type for uncertain values.

Each crate stands on its own and links back to the others through clear seams. The full API reference is on docs.rs, one page per crate.

Verify the install

A tiny program that lifts a value into a `PropagatingEffect` is enough to prove the install is wired:

```rust
use deep_causality::PropagatingEffect;

fn main() {
    let effect: PropagatingEffect<f64> = PropagatingEffect::pure(42.0);
    println!("ok: {:?}", effect.value);
}
```

```
cargo run
```

If you see `ok: Value(42.0)`, the install is good. The next page writes your first causal pipeline with the high-level Flow DSL.

A note on the docs.rs reference

Every crate ships rustdoc on docs.rs. The pages in this site's reference section are short overviews: what the crate is for and when you should reach for it. The exhaustive API stays on docs.rs.

Hello, Causal Flow

This is the friendly front door. `CausalFlow` is a fluent facade over the Causal Monad, and it lets you write a causal pipeline the way you read it: top to bottom, one verb per line. The monad underneath is doing the work, but you do not see its wrapping ceremony. Start here, then the next page opens the engine.

The smallest possible program

```rust
use deep_causality::CausalFlow;

fn main() {
    let outcome = CausalFlow::value(2_i64)
        .try_step(|x| Ok(x + 3)) // 2 + 3 = 5
        .map(|x| x * 10)         //     5 * 10 = 50
        .finish();

    println!("outcome = {:?}", outcome); // prints: outcome = Ok(50)
}
```

Four lines tell the whole story. `value(2)` seeds the flow with a starting value. `try_step` runs a fallible step. `map` transforms the value. `finish` ends the flow and hands back a `Result`. No `EffectValue` to unwrap, no `pure`/`with_state` constructors, no manual error checks between steps.

Run it:

```
cargo new hello_flow
cd hello_flow
cargo add deep_causality
# paste the code above into src/main.rs
cargo run --release
```

You should see `outcome = Ok(50)`.

Reading the pipeline

Each verb is a step the value flows through. The common ones:

- **`value(v)`** seeds a flow with a starting value.
- **`map(|v| ...)`** transforms the value and passes it on. Use it when a step only changes the value.
- **`try_step(|v| Ok(u))`** runs a fallible step. An `Ok` becomes the next value; an `Err` moves the flow into its error channel.
- **`finish()`** ends the flow and returns `Result<Value, CausalityError>`. Its sibling `run(on_ok, on_err)` dispatches to a handler instead of returning.

Every verb lowers to a single Causal Monad operation, so the flow has the exact semantics of the monad. The DSL adds reading clarity, not new behavior.

The error channel is automatic

A step that fails short-circuits the rest of the flow. You do not write `?` between steps, and you do not check for failure after each line.

```
use deep_causality::{CausalFlow, CausalityError, CausalityErrorEnum};

let outcome = CausalFlow::value(-1_i64)
    .try_step(|n| {
        if n >= 0 {
            Ok(n)
        } else {
            Err(CausalityError::new(CausalityErrorEnum::Custom("negative
input".into())))
        }
    })
    .map(|n| n * 2) // skipped: the flow is already in its error channel
    .finish();

assert!(outcome.is_err());
```

The `map` after the failing `try_step` does not run. The first failure carries straight through to `finish`, with the audit log of every step that did run still intact.

Loops and branches

Two more verbs give a flow real control flow. `iterate_n(n, step)` runs a step n times. `branch(cond, on_true, on_false)` routes the flow by a test on the current value. Both arms of a branch are themselves flows, so a branch arm can be a whole sub-pipeline.

```
use deep_causality::CausalFlow;

let total = CausalFlow::value(0_i64)
    .iterate_n(5, |tick| {
        tick.branch(
            |n| n % 2 == 0,            // is the value even?
            |even| even.map(|n| n + 10), // yes: add 10
            |odd| odd.map(|n| n + 1),    // no: add 1
        )
    })
    .finish();

assert_eq!(total, Ok(50)); // the value stays even, so +10 runs five times
```

Two more loop verbs handle open-ended iteration: `iterate_until(pred, max, step)` runs until a predicate holds, and `iterate_to_fixpoint(max, step)` runs until the value stops changing. Both take a step bound and fail with a `MaxStepsExceeded` error rather than looping forever.

Intervention: the `do()` operator

`intervene(v)` force-substitutes the value, the same `do(v)` operation from Pearl's Ladder of Causation, and records the override in the audit log. Factual and counterfactual reasoning run in the same flow, on the same engine.

```
use deep_causality::CausalFlow;

let counterfactual = CausalFlow::value(8_i64)
    .intervene(0)      // do(value = 0)
    .map(|n| n + 1)
    .finish();

assert_eq!(counterfactual, Ok(1)); // the 8 is gone; the flow continues from 0
```

`intervene_if(cond, f)` is the conditional form: it overrides the value only when a test holds, which is how a closed-loop controller fires a correction only when a monitor trips.

State when you need it

The flows above carry a value and nothing else. When a pipeline needs memory, seed it with `process(state)` instead of `value(v)` and evolve the state as it runs.

```
use deep_causality::CausalFlow;

let final_process = CausalFlow::process(0_i64) // state = 0
    .update_state(|state, _value| state + 1)   // the state evolves and carries
forward
    .into_process();

assert_eq!(final_process.state, 1);
```

This is the same flow, now Markovian: each step sees the state the previous step left. `value(v)` is the stateless form; `process(s)` is the stateful one. The verbs are identical.

What it lowers to

Nothing here is new machinery. `CausalFlow` is sugar over the Causal Monad: `value` lowers to `pure`, `map` to `fmap`, `try_step` and `branch` and `iterate_n` to `bind`. The Causal Flow concept page lays out the full verb set and what each one lowers to.

Where this goes next

The next page drops one level down to `pure` and `bind`, the two operations the whole flow is built on. After that, Hello, Causaloid wraps a step as a named, composable causal unit, and Hello, Context gives a step a world to read from. For a complete runnable program that walks Pearl's three rungs through a flow, see `examples/starter_example`.

Hello, Causal Monad

The previous page drove a pipeline with the high-level Causal Flow DSL. This page opens the engine underneath it. It introduces the monad on its own, before any Causaloid appears, because everything else in the library, the flow DSL included, composes on top of `pure` and `bind`. Understanding those two operations is important to read every other example.

What a monad is

A monad is two operations:

- **`pure(v)`**: wrap a plain value in the monad.
- **`bind(m, f)`**: given a wrapped value and a function that *also* produces a wrapped value, chain them. The function gets to see the unwrapped value, and its result becomes the next link in the chain.

Everything else follows from those two operations satisfying three identities (the *monad laws*). These operations are not a separate type; they are a trait, `CausalMonad`, implemented by the carrier effect `CausalEffectPropagationProcess`. The carrier *is* the monad.

The smallest possible program

```rust
use deep_causality::PropagatingEffect;

fn main() {
    let result = PropagatingEffect::pure(10_i32)
        .bind(|v, _state, _ctx| {
            let n = v.into_value().unwrap_or_default();
            PropagatingEffect::pure(n + 1)
        })
        .bind(|v, _state, _ctx| {
            let n = v.into_value().unwrap_or_default();
            PropagatingEffect::pure(n * 2)
        });

    let final_value = result.value.into_value().unwrap_or_default();
    println!("result = {}", final_value); // prints: result = 22
}
```

Three lines of substance. `pure(10)` lifts the integer into a `PropagatingEffect<i32>`. The first `bind` unwraps the value, increments it, and re-wraps. The second `bind` unwraps again, doubles, re-wraps. The final value is read off `result.value`.

`PropagatingEffect<T>` is the everyday alias for the carrier effect, exported from both `deep_causality` and `deep_causality_core`. The full name is `CausalEffectPropagationProcess<T, (), (), CausalityError, EffectLog>`: five type parameters, three of them pinned to defaults. The defaults are sane for almost every starting program, which is why the alias is what most code reaches for.

Run this:

```
cargo new hello_monad
cd hello_monad
cargo add deep_causality
# paste the code above into src/main.rs
cargo run --release
```

You should see `result = 22`.

What just happened

Each `bind` closure takes three arguments. The first is the wrapped `EffectValue` from the upstream link. The second is the threaded `state`. The third is the threaded `context`. The example ignores state and context because both are `()` at this scale. They become useful when the chain has a reason to carry them, and the type system makes that promotion explicit.

The closures return *new* `PropagatingEffects` rather than bare values. That is the whole point. Returning a `PropagatingEffect` means a step can also signal:

- *I produced no value.* Return `PropagatingEffect` carrying `EffectValue::None`.
- *I failed.* Return `PropagatingEffect::from_error(err)`. The chain short-circuits; downstream binds are no-ops.
- *I want to relay to a different rule.* Return `EffectValue::RelayTo(idx, sub)`. The chain reroutes.

The bare `Result` your everyday Rust code uses gives you the failure branch alone. The Causal Monad gives you all three branches plus a structured log that survives across every link.

Look at the log

Add one line to the first closure:

```
.bind(|v, _state, _ctx| {
    let n = v.into_value().unwrap_or_default();
    let mut next = PropagatingEffect::pure(n + 1);
    next.logs.add_entry("incremented");
    next
})
```

Now read the log out at the end:

```
for entry in &result.logs {
    println!("log: {:?}", entry);
}
```

Every step's log entries accumulate in `result.logs`. The chain has an audit trail by construction; you did not write a logger. The trail survives errors. If a later bind fails, the log of every step that ran before the failure is still there.

Carrying state and context

The example above ignored the second and third arguments because a `PropagatingEffect` pins both to `()`. When the chain needs real state or a real context, reach for `PropagatingProcess<T, S, C>`. It is the same carrier and the same `bind`; the only change is that `State` and `Context` now carry information, so the closure can read the running state and return an evolved one.

```rust
use deep_causality::{EffectLog, EffectValue, PropagatingProcess};

// value: i32, Markovian state: i32, context: String
let process = PropagatingProcess::<i32, i32, String>::pure(10)
    .bind(|value, state, context| {
        let n = value.into_value().unwrap_or_default();
        PropagatingProcess {
            value: EffectValue::Value(n + 1),
            state: state + 1, // the state evolves and carries forward
            context,          // the context threads through
            error: None,
            logs: EffectLog::new(),
        }
    });
```

`bind` keeps the state and context of the process the closure returns, so the chain is Markovian: each step sees the state the previous step left. The stateless `PropagatingEffect<T>` is exactly this with `State = Context = ()`. Same trait, same `bind`; the type parameters decide whether the chain carries memory.

The three laws: a quick check

A monad earns its name by satisfying three identities. The Causal Monad satisfies them.

- **Left identity.** `pure(a).bind(f)` is the same as `f(a)`.
- **Right identity.** `m.bind(pure)` is the same as `m`.
- **Associativity.** `(m.bind(f)).bind(g)` is the same as `m.bind(|v, s, c| f(v, s, c).bind(g))`.

In practice this means you can freely refactor a chain. Pull a step out into a helper. Inline a step back in. Regroup three steps into two-then-one or one-then-two. The chain still computes the same answer. The library's test suite covers all three laws explicitly.

Where this goes next

The next page wraps a function in a Causaloid and evaluates it. A Causaloid is a named, identified, composable causal function whose evaluation returns a `PropagatingEffect`. Wrapping the kind of closure you wrote above as a Causaloid is one constructor call. For a

complete, runnable end-to-end version that walks Pearl's Ladder of Causation through `pure`, `bind`, and `intervene`, see `examples/starter_example`.

The page after that adds a Context. The Context is the third argument the bind closure has been ignoring on this page. When the rule needs to read from the world, the Context is where the world lives.

Hello, Causaloid

This page walks through the smallest program that exercises `Causaloid`: a single Causaloid that wraps a predicate, then a two-node graph that composes two Causaloids, then evaluation.

What a Causaloid is

A Causaloid is a self-contained unit of causality. It carries an identifier, a human-readable description, and a causal function from an input value to a `PropagatingEffect`. Causaloids compose isomorphic-recursively into Collections and hypergraphs that share the same trait surface, which is what the Causaloid concept page covers in full.

For this example you only need a simple causaloid.

A first Causaloid

```
use deep_causality::{BaseCausaloid, Causaloid, MonadicCausable,
PropagatingEffect};

fn above_zero(x: f64) -> PropagatingEffect<bool> {
    PropagatingEffect::pure(x > 0.0)
}

fn main() {
    let causaloid: BaseCausaloid<f64, bool> =
        Causaloid::new(1, above_zero, "value is greater than zero");

    let effect = causaloid.evaluate(&PropagatingEffect::pure(3.5_f64));
    println!("effect = {:?}", effect.value);
}
```

Three things to notice.

The causal function has signature `fn(I) -> PropagatingEffect<O>`. It takes a plain value and returns a `PropagatingEffect`. There is no `Result` wrapping: errors are conveyed through the `PropagatingEffect` itself with `PropagatingEffect::from_error(...)`, and the chain short-circuits automatically.

`Causaloid::new(id, causal_fn, description)` takes an integer id, the causal function, and a description string. The id and description show up in the `EffectLog` when the Causaloid fires, which is what makes a chain auditable later.

`evaluate` takes a reference to an incoming `PropagatingEffect`, not a bare input. To pass a plain value, lift it with `PropagatingEffect::pure(...)` first. The return is another `PropagatingEffect`, which you read through `effect.value.into_value()` when you need the inner type back.

Compose two Causaloids in a graph

A trading-style example: a fast/slow moving-average cross plus a volume confirmation. Each rule is its own Causaloid. A two-node `CausaloidGraph` composes them.

```rust
use deep_causality::{
    CausableGraph, Causaloid, CausaloidGraph, MonadicCausableGraphReasoning,
PropagatingEffect,
};

fn cross(active: bool) -> PropagatingEffect<bool> {
    // Stand-in: the upstream effect carries whether the cross fired.
    PropagatingEffect::pure(active)
}

fn confirm(active: bool) -> PropagatingEffect<bool> {
    // Stand-in: the upstream effect carries whether volume confirmed.
    PropagatingEffect::pure(active)
}

fn main() {
    let c1 = Causaloid::<bool, bool, (), ()>::new(1, cross, "fast MA above slow
MA");
    let c2 = Causaloid::<bool, bool, (), ()>::new(2, confirm, "volume above
1.4x median");

    let mut graph: CausaloidGraph<Causaloid<bool, bool, (), ()>> =
CausaloidGraph::new(1);
    let root = graph.add_root_causaloid(c1).unwrap();
    let next = graph.add_causaloid(c2).unwrap();
    graph.add_edge(root, next).unwrap();

    // The graph must be frozen before it can be reasoned over.
    graph.freeze();

    let effect = graph.evaluate_single_cause(root,
&PropagatingEffect::pure(true));
    println!("{:?}", effect.value);
}
```

Three things changed compared to a singleton:

- The graph is built imperatively. `add_root_causaloid` returns the index of the root, `add_causaloid` adds further nodes, and `add_edge(from, to)` wires them up.
- The graph must be **frozen** before evaluation. Internally, freezing switches the underlying `ultragraph` backend from its dynamic build phase to its CSR query phase, which is what

gives DeepCausality sub-second traversal on graphs of ten million nodes or more.

- Evaluation uses graph methods on the `MonadicCausableGraphReasoning` trait, not the singleton `evaluate`. `evaluate_single_cause(idx, &effect)` runs one node. `evaluate_subgraph_from_cause`, `evaluate_shortest_path`, and similar methods drive the dynamic and adaptive reasoning modalities documented on the Dynamic causality page.

Reading the effect

A `PropagatingEffect`'s `value` field is an `EffectValue<T>` enum:

- `Value(T)`: the everyday case. A concrete output of type `T`.
- `None`: the explicit absence of a value.
- `ContextualLink(id1, id2)`: a deferred reference into the Context.
- `RelayTo(idx, effect)`: a dispatch command that routes the effect to a different node in the graph. This is what powers adaptive reasoning.
- `Map(parts)`: a labeled bundle of sub-effects for fan-out.

You typically pattern-match on the variant you expect:

```
match effect.value {
    deep_causality_core::EffectValue::Value(v) => println!("got {v}"),
    deep_causality_core::EffectValue::None => println!("no effect"),
    other => println!("unexpected: {other:?}"),
}
```

For the common case where you just want the inner value, `effect.value.into_value()` returns an `Option<T>`.

What's next

The single-input shape is enough to model many real workflows. When rules need to read an environment beyond their input, you move to a context-aware Causaloid via `Causaloid::new_with_context`, which threads a `Context` into every evaluation. The next page sets that up.

For a complete, runnable end-to-end example that walks Pearl's Ladder of Causation through `pure`, `bind`, and `intervene`, see examples/starter_example.

Hello, Context

The Causaloid in the previous page took an input and returned an effect. That covers a surprising amount of practical work. However, it stops the moment a rule needs to know something about the world beyond its input. The `Context` is the explicit place that world lives.

What a Context is

A Context is a typed weighted hypergraph whose nodes are `Contextoids`, each one carrying a typed payload: data, space, time, spacetime, or symbolic. The graph can be queried by id, walked along its edges, and mutated in place. Mutating it is the *dynamic* in dynamic causality: the same Causaloid evaluated against a new Context yields a new propagating effect. The Context concept page explains this further.

The `BaseContext` alias pins the seven generic parameters of `Context` to a sensible Euclidean default, which is what every example below uses. The base context lives in `deep_causality` and is built on top of the `ultragraph` hypergraph backend.

Build a Context

We will give a rule a tunable volume threshold by putting it in the Context as a Datoid.

```
use deep_causality::{
    BaseContext, ContextoidType, Contextuable, ContextuableGraph, Context,
Contextoid, Data,
};

fn build_context() -> BaseContext {
    let mut ctx: BaseContext = Context::with_capacity(1, "trading", 8);

    let threshold = Contextoid::new(
        10,
        ContextoidType::Datoid(Data::new(10, 1_500.0)),
    );
    let _idx = ctx.add_node(threshold);

    ctx
}
```

Three things to notice.

`Context::with_capacity(id, name, capacity)` takes an integer id, a name, and a pre-allocated capacity for the underlying hypergraph. The name shows up in logs; the capacity is a hint, not a hard cap.

`Data::new(id, value)` carries a numerical payload, and `Contextoid::new(id, ContextoidType::Datoid(data))` wraps it as a node. The other `ContextoidType` variants — `Spaceoid`, `Tempoid`, `SpaceTempoid`, `Symboid` — wrap the other four payload kinds.

`add_node` returns the node's index in the graph. The library also maintains an id-to-index map internally so queries by id stay O(1) regardless of insertion order.

A context-aware Causaloid

A context-aware Causaloid uses `Causaloid::new_with_context` and a slightly richer function signature than the stateless case. The context is shared, so the library wraps it in `Arc<RwLock<BaseContext>>`.

```rust
use deep_causality::{
    BaseCausaloid, BaseContext, Causaloid, ContextoidType, Contextuable,
MonadicCausable,
    PropagatingEffect, PropagatingProcess,
};
use deep_causality_core::EffectValue;
use std::sync::{Arc, RwLock};

fn volume_above_threshold(
    obs: EffectValue<f64>,
    _state: (),
    ctx: Option<Arc<RwLock<BaseContext>>>,
) -> PropagatingProcess<bool, (), Arc<RwLock<BaseContext>>> {
    let Some(volume) = obs.into_value() else {
        return PropagatingProcess::from_error(
            deep_causality::CausalityError::new(
                deep_causality_core::CausalityErrorEnum::Custom("missing
input".into()),
            ),
        );
    };
    let Some(ctx_arc) = ctx else {
        return PropagatingProcess::from_error(
            deep_causality::CausalityError::new(
                deep_causality_core::CausalityErrorEnum::Custom("missing
context".into()),
            ),
        );
    };

    let ctx_lock = ctx_arc.read().unwrap();
    let node = ctx_lock.get_node(0).expect("threshold contextoid missing");

    let threshold = match node.vertex_type() {
        ContextoidType::Datoid(d) => d.data(),
        _ => return PropagatingProcess::from_error(
            deep_causality::CausalityError::new(
                deep_causality_core::CausalityErrorEnum::Custom("wrong
contextoid kind".into()),
            ),
        ),
    };
```

```
    PropagatingProcess::pure(volume > threshold)
}

fn main() {
    let ctx = Arc::new(RwLock::new(build_context()));

    let causaloid: BaseCausaloid<f64, bool> = Causaloid::new_with_context(
        100,
        volume_above_threshold,
        ctx.clone(),
        "volume exceeds the threshold from context",
    );

    let effect = PropagatingEffect::pure(2_400.0_f64);
    let result = causaloid.evaluate(&effect);
    println!("{:?}", result.value);
}
```

Three things changed compared to the stateless Causaloid:

- The function signature is `fn(EffectValue<I>, S, Option<C>) → PropagatingProcess<O, S, C>`. The first argument is an `EffectValue`, not a bare `I`, because the upstream `PropagatingEffect` can carry richer variants like `RelayTo` or `None`. The second is the threaded state, ignored here because `BaseCausaloid` pins it to `()`. The third is the optional context.
- The return type is a `PropagatingProcess`, the stateful sibling of `PropagatingEffect`. For a stateless context-aware rule like this one the state stays `()`, but the type is shared with truly stateful reasoning so the trait machinery composes uniformly.
- The Context is wrapped in `Arc<RwLock<...>>` so it can be shared across Causaloids and mutated in place. Read access uses `ctx_lock.get_node(...)`; mutation uses `ctx_lock.update_node(...)`.

Mutating the Context

This is what makes the model dynamic. Replace the threshold and the same Causaloid produces a different effect on the same input:

```
use deep_causality::{ContextoidType, ContextuableGraph, Contextoid, Data};

let new_threshold = Contextoid::new(
    10,
    ContextoidType::Datoid(Data::new(10, 3_000.0)),
);

ctx.write().unwrap().update_node(10, new_threshold).unwrap();
```

The Causaloid never moved. The rule never moved. The world moved. The library treats that as a first-class case.

The Context concept page covers the related `Adjustable` trait, which exposes `update` and `adjust` methods on node payloads for the case where the structure stays fixed but incoming sensor data needs to be replaced or corrected for drift.

When to add to the Context

A rule of thumb: if the rule needs to see a value that changes during a run, a value that an operator might tune without rebuilding, or a value that a counterfactual scenario might want to replace, that value belongs in the Context. Also, all external data feeds usually go through the context first.

What's next

You now have the core moving parts: a Causal Monad, a Causaloid, a Context, and the propagating effect that flows between them. The next page shows how these compose into one chain, and how a non-Markovian `PropagatingEffect` lifts into a Markovian `PropagatingProcess` when a downstream step needs state. The concept pages cover each piece in more detail, and the examples show what the primitives let you build.

Hello, Effect Propagation

The earlier pages introduced the carrier effect (with its `pure` and `bind`), the Causaloid (which wraps a causal function), and the Context (which holds the world a rule reads from). This page is about what happens when you put them together.

The short version: a Causaloid's evaluation and a `bind` step both return the same carrier effect. That single shared return type is what lets one chain mix structural causal reasoning (Causaloids, Collections, Graphs) with sequential causal reasoning (monadic bind) without bridge code.

The shared return type

Every Causaloid evaluation returns a `PropagatingEffect<T>` (stateless) or a `PropagatingProcess<T, S, C>` (stateful). Every monadic `bind` returns the same. The Causaloid page covered the structural side; the Causal Monad page covered the sequential side. The type that flows between them is the same on both sides.

A practical consequence: you can evaluate a Causaloid, take its `PropagatingEffect`, and `.bind(|...|)` directly onto it. You can run a bind chain, take the result, and feed it into a Causaloid's `evaluate`. Both directions work because the carrier is uniform.

Two aliases, one underlying type

`PropagatingEffect<T>` and `PropagatingProcess<T, S, C>` are both type aliases over the same 5-arity container, `CausalEffectPropagationProcess<V, S, C, E, L>`:

```
pub type PropagatingEffect<T> =
    CausalEffectPropagationProcess<T, (), (), CausalityError, EffectLog>;

pub type PropagatingProcess<T, S, C> =
    CausalEffectPropagationProcess<T, S, C, CausalityError, EffectLog>;
```

The difference is the second and third type parameters. `PropagatingEffect` fixes state and context to the unit type `()`. `PropagatingProcess` keeps them generic.

Non-Markovian vs Markovian

The distinction maps directly onto two reasoning styles you will run into in practice:

- **`PropagatingEffect<T>` is non-Markovian**: each step depends only on its input value and the rules it runs. No state is carried across steps, no context is threaded through the chain. Treated as a Higher-Kinded Type, the witness is `HKT3` over `(T, E, L)`. Use this when the steps in your chain do not need to remember anything from earlier steps.
- **`PropagatingProcess<T, S, C>` is Markovian**: each step receives the previous value *plus* the threaded state `S` and context `C`. The state evolves as the chain runs; the context can be read from at any step. Treated as a Higher-Kinded Type, the witness is `HKT5` over `(T, S, C, E, L)`. Use this when the chain needs to accumulate, decay, or condition on what came before.

Both share the underlying container, so the `bind` operator works the same way for both. What differs is what each step has access to.

Lifting a `PropagatingEffect` into a `PropagatingProcess`

A common pattern: you start with simple non-Markovian rules, then realize one downstream step needs state. The library lets you lift without rewriting the upstream.

```
use deep_causality::{PropagatingEffect, PropagatingProcess};

#[derive(Clone, Default)]
struct RiskState { total_risk: f64 }

let stateless: PropagatingEffect<f64> = PropagatingEffect::pure(0.5);

let stateful: PropagatingProcess<f64, RiskState, ()> =
    PropagatingProcess::with_state(stateless, RiskState::default(), None);

let updated = stateful.bind(|val, mut state, _ctx| {
    let v = val.into_value().unwrap_or_default();
    state.total_risk += v;
    let mut next = PropagatingProcess::pure(v * 2.0);
    next.state = state;
    next
});
```

`PropagatingProcess::with_state(effect, initial_state, initial_context)` takes a stateless `PropagatingEffect`, an initial state, and an optional initial context. It returns a `PropagatingProcess` ready for stateful binds. The value, error, and log channels carry across unchanged. The state and context channels are now live.

The reverse direction is also fine: a stateful chain can ignore its state and context entirely when a step does not need them. The type-level distinction is real, but the boundary is one constructor call.

A bind-chain stands on its own

You can build a `pure/bind/intervene` chain on the carrier without Causaloids. The starter example does exactly that: `examples/starter_example` walks Pearl's Ladder of Causation as a pure `pure/bind/intervene` chain with no Causaloid in sight. The carrier's algebra is enough on its own when the reasoning is sequential and the rules are small enough to live as closures.

You can also use Causaloids without explicit monadic chaining. A single `causaloid.evaluate(&PropagatingEffect::pure(input))` is a complete program for the cases where structural composition (Singleton, Collection, Graph) is what you want.

How the bind-chain and Causaloid compose

Real systems need both. Sequential transforms belong in a bind-chain. Parallel aggregation belongs in a Causaloid Collection. Cross-influencing dependencies belong in a Causaloid Graph. Because every shape returns the same propagating-effect carrier, a single pipeline can mix all three. A Causaloid Graph emits a `PropagatingEffect`, which a `.bind` step consumes, which feeds another Causaloid evaluation, which feeds a `.bind`, and so on. State and audit log accumulate across every stage.

The flight envelope monitor example is the canonical demonstration. It runs a Causaloid Collection over five sensor-health checks, a three-step Causal Monad bind-chain for state estimation, and a Causaloid hypergraph of six envelope protections, all threading through one `PropagatingProcess<T, FlightState, AircraftConfig>` with state and audit log carried across every stage.

What this enables

Most of the advanced examples in the repository depend on this composition. The pattern is the same in each: a Causaloid (Singleton, Collection, or Graph) supplies the structural reasoning, and a Causal Monad bind-chain sequences the rest, both sharing the same propagating-effect carrier. The boundary between “structural” and “sequential” is fluid and you can move that boundary as the problem evolves. Start with a simple non-Markovian bind-chain. Add structure later with a Causaloid. Add state later into the Markovian part of the chain, and then combine all parts fluently.

The concept pages on Causaloid, Causal Monad, and Effect Propagation Process go deeper on the algebra. The examples put the composition to work.

Concepts

Reference pages for every primitive the library exposes. Start with **Dynamic causality** for the framing, then **Causaloid** and **Context** for the two structural units, then the **Effect Propagation Process** for what flows between them. The **Causal Monad** page covers the `pure`/`bind` algebra that carrier implements. The rest can be read on demand.

Foundations

- **The Axiom** — the single axiom the framework rests on, $m_2 = m_1 \gg\!= f$, and the properties it unlocks.
- **Dynamic causality** — the umbrella idea the rest of the library sits inside, and the philosophical commitment behind it.
- **Causaloid** — a self-contained unit of causality that composes into larger units of itself.
- **Context** — the explicit hypergraph that a dynamic causal rule reasons against.
- **Effect Ethos** — the deontic guardrail that intercepts every action the Causal State Machine proposes before it executes.

Propagation

- **Effect Propagation Process** — the carrier effect: the struct that carries a value, a state, a context, an error, and a log through a chain of Causaloids.
- **Causal Monad** — the pure/bind algebra the carrier implements, which makes effect propagation composable, auditable, and short-circuiting on error. A trait, not a separate type.
- **Causal Flow** — the fluent high-level DSL over the causal monad. Pipelines, loops, branches, and interventions as verbs, lowering to `pure` and `bind`.
- **Higher-Kinded Types** — how DeepCausality encodes the type constructors that Rust does not natively support.

Surfaces and tooling

- **Causal Discovery Language** — a typestate-builder DSL for going from raw observational data to an executable causal model.

- **Causal State Machine** — a registry of state-action pairs whose transitions are driven by causal evaluation rather than fixed thresholds.
- **Uncertainty** — a first-order type for uncertain values, plus a companion type for probabilistic presence.
- **Uniform Math** — one Functor/Monad/CoMonad surface across tensors, multivectors, manifolds, sparse matrices, and effect propagation.

Reference

- **Glossary** — canonical terminology for DeepCausality. The names land here once; everything else references this page.

Axiom

DeepCausality is the reference implementation of the *Effect Propagation Process* (EPP), which rests on a single axiom: every effect is derived from a prior effect by a causal function embedded into a monadic composition denoted as $m_2 = m_1 \gg\!= f$.

The EPP is an axiomatic, spacetime-agnostic foundation upon which domain-specific dynamic causal models are built and composed with one another, all derived from a single axiom. As we will see, this one axiom unlocks a number of unique properties:

- Static & dynamic causality processes
- Markovian & non-Markovian processes
- Euclidean & non-Euclidean spacetime
- Deterministic & probabilistic reasoning

Unpacking the axiom

The equation is compact; its power rests on two deliberate symmetries, and on what those symmetries make expressible. The seven steps below illustrate the higher-order effects resulting from the axiom.

01 — The Axiom

Read $m_2 = m_1 \gg\!= f$ as follows: a causal function f derives a new effect m_2 from a prior effect m_1. `bind` is that $\gg\!=$. It takes the value held in m_1, applies f, and returns the next effect. No value is unwrapped by hand between steps.

```rust
use deep_causality::PropagatingEffect;

// m₂ = m₁ >>= f — bind feeds m₁'s output straight into the causal function f.
let m1 = PropagatingEffect::pure(10);
let m2 = m1.bind(|v, _state, _ctx| {
    PropagatingEffect::pure(v.into_value().unwrap_or_default() + 1)
});

assert_eq!(m2.value.into_value(), Some(11));
```

02 — Two Unifications

How can the output of one step serve as the input to the next without translation? Two deliberate symmetries make it possible.

Lucian Hardy, in his foundational work, established that, absent a fixed spacetime, nothing distinguishes a cause from its effect, because the temporal order is missing. The Causaloid therefore folds *cause and effect* into a single entity, which lets causal relations operate independently of any specific spacetime. This establishes the first symmetry: cause and effect as one entity.

The first symmetry, in turn, requires a carrier that propagates information.

The propagating effect folds causal *input and output* into a single isomorphic carrier, so m_1 and m_2 share one type. This establishes the second symmetry: input and output as one entity.

```
cause  ⇄ effect    Causaloid · one entity
input  ⇄ output    propagating effect · one carrier

carrier: value · state · context · error · log
```

The function `f` operates on a carrier of five fields.

```
pub struct CausalEffectPropagationProcess<Value, State, Context, Error,
Log> {
    pub value:   EffectValue<Value>, // the propagating effect
    pub state:   State,              // Markovian state
    pub context: Option<Context>,    // the explicit world
    pub error:   Option<Error>,      // short-circuits the chain
    pub logs:    Log,                // append-only audit trail
}
```

The carrier specializes into two subtypes. The propagating effect encapsulates only the value, error, and log; the propagating process covers all five fields. This split is deliberate. By excluding state and context, the propagating effect applies to the large class of problems that need neither, without adding complexity, while more demanding problems, which require state, context, or both, use the propagating process. The two compose cleanly because they derive from the same carrier, so stateless and stateful processes can be combined as the problem requires. This design is also what resolves Russell's contradiction.

A physical law is *time-symmetric* when it runs equally well forward and backward; Newton's equations, Maxwell's equations, and the Schrödinger equation do not record which way time flows. Film an elastic collision, play it in reverse, and the reversed motion still satisfies the same equations. Causation insists on the opposite: a cause precedes its effect, and no effect runs backward into its cause. In 1912 Bertrand Russell pressed the contradiction to its logical conclusion. If fundamental physics is time-symmetric and causation is not, then causation and modern physics are mutually exclusive.

The contradiction dissolves once two asymmetries, long conflated under one name, are distinguished. The first belongs to the *law*: whether an equation is invariant under time reversal. The second belongs to *propagation*: the direction in which one effect gives rise to the next. Classical causality conflated the two. In a structural causal model the directed assignment `Y := f(X)` carries both the mechanism and its direction in one object, so a time-symmetric law has no place; the formalism imposes a direction on a law that has none.

The central axiom, $m_2 = m_1 \gg\!= f$, separates the asymmetry of direction from the asymmetry of time. The causal function `f` holds the law, which may be perfectly time-symmetric, while $\gg\!=$ holds the direction of propagation as a structural property of composition rather than of the law. A symmetric physical equation therefore composes inside a causal chain without contradiction, which is why a single algebra runs models of general relativity, electromagnetism, and quantum mechanics. Causality and modern physics no longer collide; in DeepCausality, they compose. The Dynamic causality page develops the argument from Russell through Whitehead, separating the notion of direction from the notion of time.

03 — Recovering the classical methods

Pearl's structural models, Granger's time-series test, and Rubin's potential outcomes are each the axiom with some of its freedom removed. DeepCausality generalizes the established methods of classical computational causality, and can therefore reconstruct any of them by specialization.

The base case is Pearl's biconditional, and it returns by lowering four independent parameters of the axiom to their floor. Setting the state and context channels to the unit type collapses the monad to a plain function. Restricting the value domain to two values, then the function to the identity on them, yields the biconditional itself. Reading the propagation direction as forward time restores temporal precedence.

The same reduction extends from the definition to the methods built upon it. Each established method is the axiom under a further restriction, and for the inference methods the operative

step is a single mechanism, contextual alternation: clone the context, change one part of it, and re-evaluate the same causal law, which is what replaces Pearl's abduction. The dynamic Bayesian network is the exception, recovered structurally rather than by alternation.

- **Structural causal models (Pearl).** Recovered as the axiom under the unit context with bivalent values, the identity mechanism, and forward-time propagation; the three rungs of the ladder map to evaluation of a causal function in the factual context, the do-operator, and a counterfactual obtained by pinning the factual context, alternating it, and re-evaluating the unchanged causal graph.
- **Potential outcomes (Rubin).** The outcomes `Y(1)` and `Y(0)` are the same causal graph evaluated against a treatment context and a control context, and the unit-level effect is their difference. The fundamental problem of causal inference is a constraint on the physical measurement of a single unit.
- **Conditional average treatment effect.** The potential-outcomes construction with the conditioning set `X = x` represented as a static context, giving `τ(x) = E[Y(1) - Y(0) | X = x]` as the averaged difference of the two evaluations.
- **Granger causality.** Each series is represented as a temporal context of time-indexed values, and the test compares a predictor's error under the full context against its error under an alternate context that masks one series' history. A reduction in error indicates a predictive, time-ordered dependence.
- **Dynamic Bayesian networks.** Recovered structurally: one static causal graph is evaluated over a temporal context whose nodes are time slices, with each variable's conditional-probability table realized as a node's causal function and within- and cross-slice dependencies as hyperedges over the relevant slices.

Each recovery is implemented twice. The first carries context, state, and audit on the propagating effect and alternates on the carrier through the `Alternatable` mechanism. The second holds the causal law in a Causaloid and alternates it by cloning an external context and re-evaluating. Both produce identical results across all five methods, as documented in the project repository.

04 — Intervention across three channels

Causal reasoning rests on a distinction between *seeing* and *doing*. To observe that two quantities move together measures correlation, which cannot separate a cause from a coincidence. An *intervention* settles the matter. It forces a variable to a chosen value, cuts it off from whatever ordinarily sets it, and lets the chain run. Only this isolates one factor as the driver

of another, and because the strength of a cause is the size of the change its intervention produces, the same operation measures how strongly causes act and ranks competing ones.

Judea Pearl gave this idea its formal machinery. His *do-operator*, written `do(x)`, together with the surrounding do-calculus, separated doing from seeing inside a rigorous algebra of causation, work for which he received the Turing Award. In the EPP that operator is a single method, `intervene`, applied at any point in a chain. The factual and counterfactual runs share one structure; a single call distinguishes them, and the substitution enters the log.

```rust
use deep_causality::{CausalFlow, PropagatingEffect};

// Chain: blood pressure → wall shear stress → arterial fatigue. The factual
run
// and each counterfactual share one structure; a single .intervene
distinguishes
// them and records do(...) in the audit log.
fn run_factual(baseline_bp: f64) -> PropagatingEffect<CycleSummary> {
    CausalFlow::value(baseline_bp)
        .map(shear_stress_stage)
        .map(fatigue_stage)
        .into_effect()
}

// do(BP = 120): a beta-blocker sets blood pressure before the chain runs.
fn run_medication(baseline_bp: f64, controlled_bp: f64) ->
PropagatingEffect<CycleSummary> {
    CausalFlow::value(baseline_bp)
        .intervene(controlled_bp)
        .map(shear_stress_stage)
        .map(fatigue_stage)
        .into_effect()
}

// do(WSS = 8): a surgical clip sets wall shear stress mid-chain.
fn run_surgical(baseline_bp: f64, clipped_wss: f64) ->
PropagatingEffect<CycleSummary> {
    CausalFlow::value(baseline_bp)
        .map(shear_stress_stage)
        .intervene(clipped_wss)
        .map(fatigue_stage)
        .into_effect()
}

// value is Pearl's do(x); context and state use alternate_context /
alternate_state.
```

In the example above, the factual run gives the outcome as it stands; each counterfactual gives the outcome that a different action would have produced. The causal effect of a treatment is the difference between the two arterial-fatigue summaries, and because the stages, the chain, and the model are identical across runs, that difference is attributable to the intervention alone. The

three runs walk Pearl's ladder in order: the factual run observes, the `.intervene` call does, and the comparison of their outcomes is the counterfactual. Setting the medication result beside the surgical one then ranks the two treatments by effect, and the audit log records each `do(...)`, so every verdict traces to the substitution that produced it.

Pearl's operator reaches exactly one thing: the value of a single variable. A full counterfactual then needs more apparatus. The model graph is surgically altered, and the unobserved background is recovered by abduction, a posterior over hidden noise reconstructed from what was seen. The reach is one value at a time, the world is inferred rather than held, and the surgery occurs at the level of the model.

DeepCausality widens the operation and removes the complexity. The carrier holds the world explicitly in its context and the history in its state, so an intervention rewrites a channel on the running chain instead of operating on the model, and abduction is no longer required. Three channels are open, each with its own operator and audit marker: *value*, *context*, and *state*.

The two added channels change the category of question that can be asked. A *context* intervention substitutes the world the chain reasons against, so a counterfactual can vary the entire environment rather than a single variable: what the model would have concluded under a different regime, or the outcome under treatment placed beside the outcome under control, with no model rebuilt. A *state* intervention forces the running memory of the process, so a counterfactual can vary the accumulated history: a simulator reset, a regime change that has already marked the counters, or a trajectory rewound to an earlier point. Pearl's `do(x)` asks what follows if one variable were set to `x`. The EPP also asks what follows if the world were different, and what follows if the history were different.

05 — From intervention to correction

Intervention so far has been analysis: a person poses a what-if question and reads the answer. The same operation becomes control once the chain performs it on itself. Because value, state, and context are externalized on the carrier, a step can read them mid-flow and branch on what it finds, and a branch that ends in an intervention is a correction. The chain watches its own trajectory and intervenes the moment an intervention becomes necessary.

The shape is the control loop, written in the DSL. Each tick advances the model, then branches on the carried value: inside the safe envelope the tick passes through untouched; outside it, the corrective arm fires `intervene`, and the next tick continues from the corrected value.

```rust
use deep_causality::CausalFlow;

// Lane keeping. Each tick advances the model; the chain then inspects its own
// value and corrects itself when the car drifts past the safe threshold.
CausalFlow::from(model::initial_process())
    .iterate_n(N_TICKS, |tick| {
        tick.bind(model::simulate_step).branch(
            // necessity condition detected?
            |offset| offset.abs() > cfg.anomaly_threshold,
            // if so, self-correct
            |hot|  hot.intervene_if(|_| true, |o| model::correction(o, &cfg)),
            // if not, no action
            |cold| cold,
        )
    })
    .into_process()
```

```
advance → inspect → correct if needed ↺ repeat each tick
```

Dynamics and control remain fully separate: the model knows nothing of correction, and the monitor knows nothing of the physics; the two meet only through the carried value. Every correction is logged like any other intervention, so a closed loop stays as auditable as the open run it replaces. As a direct consequence, simulating adverse conditions through a targeted intervention ahead of the correction becomes programmable. Where before we could only ask what would have happened if a value were different, we can now ask whether the fail-safe mechanism still holds under adverse conditions. We can also determine the range over which it is reliable in simulation and, just as important, where it stops.

06 — First-class uncertainty

Classical causal inference rests on statistics, yet it admits uncertainty only at the edges. It estimates a treatment effect and puts a confidence interval around it, while the quantities inside the model stay point estimates: a sensor reads `50.0`, a covariate is one number, a reading is either present or dropped. That breaks in two common cases. When the effect itself is probabilistic, a point estimate discards its shape. And when a value is missing, the missingness usually carries information: the patient who feels worse skips the daily report, and the sensor drops frames under the very vibration it watches. Impute a default for those gaps, and the bias runs the wrong way.

DeepCausality makes uncertainty a first-class type, after Bornholt and colleagues. `Uncertain<T>` carries a value together with the distribution that produced it; arithmetic and comparison build a computation graph rather than collapsing to a number, and evaluation samples lazily to a requested confidence.

`MaybeUncertain<T>` goes one step further, factoring a reading into two questions held apart: a presence probability, carried as an `Uncertain<bool>`, and the distribution the value follows when present. A reading present eighty percent of the time and one present five percent of the time are different objects, and the difference survives: `is_some()` returns the presence distribution itself, not a flag. Given presence, the original distribution remains, ready to sample.

```
use deep_causality_uncertain::{MaybeUncertain, Uncertain};

// One trial patient's daily pain-reduction reading: reported on ~70% of days,
// and, when reported, distributed Normal(4.0, 2.5). MaybeUncertain holds both.
let reading = MaybeUncertain::from_bernoulli_and_uncertain(0.7,
Uncertain::normal(4.0, 2.5));

// Presence is itself uncertain: is_some() returns the Bernoulli(0.7), not a
flag.
let present: Uncertain<bool> = reading.is_some();

// Commit to a value only when presence clears a confidence bar (here P > 0.5
at
// 95% confidence). Below it the gate fails and the chain short-circuits,
instead
// of imputing a number. This is the per-patient step of the clinical-trial
flow.
let score: Uncertain<f64> = reading.lift_to_uncertain(0.5, 0.95, 0.05, 1_000)?;
```

```
classical          → 50.0                       a point
Uncertain<T>       → Normal(50, 2.5)            a distribution
MaybeUncertain<T>  → P(present) × distribution  presence and distribution
```

The two channels propagate together. Add two such readings and the values combine as distributions, while the result counts as present only when both are; absence flows through the arithmetic with the right probability. A gate, `lift_to_uncertain`, commits to a plain `Uncertain<T>` only when presence clears a confidence bar. Below it the gate fails, and the chain short-circuits rather than inventing a number, the same way any failed step stops the

propagation. The model can therefore distinguish *the value is zero* from *I never observed the value*, and decline to answer when the evidence for presence is too thin.

07 — Expressive range: non-Euclidean, relativistic, quantum

The foundational axiom, as implemented in DeepCausality, supports a broad range of expression:

- Markovian & non-Markovian processes
- Euclidean & non-Euclidean spacetime
- Static & dynamic causality
- Deterministic & probabilistic reasoning

The Markovian property enables stateful processes, which cover a broad variety of advanced engineering and physics domains, as demonstrated in the avionics examples.

Non-Euclidean spacetime representation is common in advanced physics, for example when modeling general relativity for satellite navigation. Combined with the Markovian property, stateful physics simulations across Euclidean and non-Euclidean regimes compose natively, as demonstrated in the physics example.

Static and dynamic causal processes enable the combination of pre-existing background knowledge with current measurements, as demonstrated in the medicine examples.

Probabilistic reasoning paves the way for advanced applications in frontier fields that are inherently probabilistic, as demonstrated in the quantum examples and the uncertain examples.

Each property alone already enables a category of advanced science and engineering that was previously difficult to address with computational causality. Taken together, multiple fields of advanced science and technology compose, as demonstrated in the GRMHD example, which models magnetohydrodynamics under general relativity, or the event horizon example, which simulates a space probe descending into a black hole while transitioning seamlessly through Newtonian and relativistic regimes to compute its trajectory. Inverting general relativity itself is even possible, as demonstrated in the chronometric example, which recovers the GM constant to compute planetary mass from time dilation. Such complex, dynamic causal processes occur in countless domains, and accordingly the DeepCausality project provides over one hundred code examples to illustrate dynamic causality across them.

A single axiom gives rise to nearly limitless possibilities. The future is now.

Where to look next

Dynamic causality develops the framing the axiom sits inside. Causal Monad is the `pure/bind` algebra the axiom names, and Effect Propagation Process is the five-field carrier it operates over. Counterfactuals and Uncertainty cover the intervention and probabilistic channels in depth.

Dynamic causality

DeepCausality is a framework for **dynamic causality**. This page explains what the phrase commits to, where the commitment came from, and what it earns you in exchange.

The axiom

A causal relation is a **monadic functional dependency**. Formally:

```
m₂ = m₁ >>= f
```

Where m_1 and m_2 are propagating effects, f is a causal function, and $>>=$ is monadic bind. The EPP preprint calls this *the* axiom of causality. Every concept in this library is a specialization of that single equation.

Pearl SCMs, dynamic Bayesian networks, Granger causality, the Rubin causal model, and conditional average treatment effects are all parametric specializations of the same axiom. The classical causality examples walk each one through in code.

What “dynamic” means

Classical computational causality, from Pearl’s Structural Causal Models to Granger time-series analysis, assumes a fixed background spacetime and a static causal structure. At the frontiers of science and engineering, that assumption breaks. Regime shifts in financial markets, multi-scale feedback in autonomous vehicles, and the dynamic spacetime of general relativity all share a property: the causal rules themselves can evolve.

The EPP commits to making that evolution first-class, and the commitment is philosophical before it is technical. Chapter 3 of the preprint traces a line from Aristotle’s substance metaphysics, through Hume, Kant, and Reichenbach, to Russell’s 1912 critique. Russell observed that modern physics describes systems with time-symmetric equations, while classical causality demands temporal asymmetry. He concluded that “the law of causality is a relic of a bygone age.”

The EPP’s answer is the **Functional View**, drawn from Whitehead’s process philosophy. Reality is a process of becoming, not a collection of substances. Causality is not an external relation linking pre-existing things. It is the structure of effect propagation itself. Russell’s puzzle

dissolves once you notice that two distinct asymmetries had been conflated under one name: the asymmetry of laws (whether the equations are time-reversal invariant) and the asymmetry of propagation (the direction of flow from cause to effect). The monadic axiom separates the two. The function `f` inside `bind` can be a time-symmetric physics equation. The `bind` operator carries propagation asymmetry as a structural property of composition. Classical causality and modern physics stop fighting.

The four reasoning modalities

Section 4.9 of the EPP defines four modes of causal reasoning. Each is a setting of two knobs: is the graph fixed or constructed at runtime, and is the context fixed or evolving?

- **Static**: the causal graph is fixed and traversed along a pre-defined pathway. The classical case. A medical risk score driven by a fixed clinical protocol.
- **Dynamic**: the graph is fixed, but the pathway through it is selected at runtime. A subgraph or a shortest-path traversal answers a query. The context may itself be dynamic and feed into pathway selection. A trading model that queries a static rule library against a live market feed.
- **Adaptive**: the Causaloid itself dispatches to the next step based on its own internal logic and the current context. A clinical model that hands off to a “normal blood pressure” or “high blood pressure” subgraph depending on the latest reading. The set of possible pathways is closed and known at design time.
- **Emergent**: the graph is constructed and modified at runtime in response to the context, often combined with adaptive reasoning inside the new graph. New Causaloids and new edges are introduced by a generative process. The set of possible pathways is no longer closed.

Static, Dynamic, and Adaptive all remain deterministic. Their state space is bounded, their dispatch rules are known up front, and formal verification is feasible. Emergent reasoning is different.

Adaptability vs verifiability

Section 4.9.5 of the EPP states the trade-off plainly:

- **Adaptive reasoning** is deterministic dynamics. The system adapts within a pre-defined and pre-validated set of behaviors. It is verifiable.

- **Emergence** is non-deterministic dynamics. The system can construct causal structures that were not foreseen by the designer. It is not statically verifiable.

The non-verifiability of emergent reasoning is not a flaw in the formalism. It is a property of the world the system is coupled to. When a causal graph evolves in response to a sensor stream, the system reads from an open environment. The generative function cannot be proven deterministic in the abstract, because it consumes data from a world that is not itself bounded.

The EPP responds with an architectural answer: the **Effect Ethos**, an operational guardrail that uses a Defeasible Deontic Inheritance Calculus (DDIC, after Olson, Salas-Damian, and Forbus). The Effect Ethos sits above the Causal State Machine and intercepts every proposed action. It evaluates the action against a graph of Teloids, each one a computable norm. Conflict between norms is resolved by three principles:

- **Lex Posterior**: the more recent norm wins.
- **Lex Specialis**: the more specific norm wins.
- **Lex Superior**: the higher-priority norm wins.

Verifiability is restored at the action layer rather than the reasoning layer. The reasoning graph may evolve in ways no static proof can foresee, yet every action that leaves the system has been checked against an immutable ethos. The result is a discipline for managing emergence via programmable ethics.

The operational pieces

Four primitives operationalize the axiom:

- **Causaloid**: a self-contained unit of causality. Wraps a function from input (and optionally context) to a `PropagatingEffect`. Composes isomorphic-recursively into Collections and hypergraphs that share the same trait surface.
- **Context**: an explicit hypergraph encoding the environment. Nodes are typed Contextoids carrying data, space, time, spacetime, or symbolic payloads. The Context is what makes a Causaloid evaluation context-relative without committing to a fixed background spacetime.
- **Propagating Effect**: the shared carrier every Causaloid emits and consumes. Realized as a 5-arity container `CausalEffectPropagationProcess<V, S, C, E, L>` in `deep_causality_core`, exposed through two aliases: the non-Markovian `PropagatingEffect<T>` with state and context fixed to the unit type, and the Markovian

`PropagatingProcess<T, S, C>` that keeps both generic. Lifting from one to the other is one constructor call. The carrier implements the **Causal Monad** trait, the `pure/bind/intervene` algebra that gives structural reasoning (a Causaloid) and sequential reasoning (a bind-chain) a single boundary type, which is why they compose without bridge code. `intervene` implements Pearl's `do()` operator mid-chain, making counterfactual analysis a first-class operation rather than a separate engine.

- **Effect Ethos**: the deontic verification layer described above. Required wherever emergent reasoning is in play. Optional otherwise.

What this earns you

- **Composition without identity loss**: two Causaloids combine into a third with the same type. Tooling that worked on one works on the composition.
- **Audit trail by construction**: every chain accumulates an `EffectLog`. The trail survives errors and is the system's answer to "which rule fired, on what inputs, in what order?"
- **Counterfactuals as a first-class operation**: `intervene` rewrites the value at any point in the chain. The same machinery handles factual and counterfactual evaluation.
- **Four modalities under one type**: a single Causaloid expression can be promoted from static to dynamic to adaptive without rewriting the calling code. Emergence is reachable through the generative process documented in chapter 8 of the preprint.
- **A formal grounding**: the Rust types correspond to definitions in the EPP preprint and its companion volumes, not to convention.

Where to look next

The three primitive pages walk through the actual types: Causaloid, Context, Effect Ethos. The Effect Propagation Process page covers the 5-arity container. The Causal Monad and HKT pages show how the algebra is implemented in Rust without runtime overhead.

For the formal treatment, start with the EPP preprint. Chapter 3 covers the philosophical move. Chapter 4 defines the axiom and the four reasoning modalities. Chapter 7 covers the Effect Ethos and the non-determinism solution. Chapter 8 covers the ontology of emergence. The Formalization preprint carries the math.

Causaloid

A `Causaloid` is the fundamental unit of causality in DeepCausality. Three properties define it.

1. It wraps a function that takes input, optionally consults a context, and returns a `PropagatingEffect`.
2. It carries enough metadata (id, name, description) to remain identifiable when it shows up in a log.
3. It composes isomorphic-recursively. A Causaloid, a collection of Causaloids, and a graph of Causaloids all implement the same `Causable` + `MonadicCausable` trait surface, so each one stands in for any other and they nest into each other without limit.

The third property is the load-bearing one. It is borrowed from physicist Lucian Hardy's work on quantum gravity, where a *causaloid* folds cause and effect into a single object so that causal structure can be discussed without assuming a fixed temporal order.

The type

The Rust definition lives in deep_causality/src/types/causal_types/causaloid/mod.rs:

```
pub struct Causaloid<I, O, STATE, CTX>
where
    I: Default,
    O: Default + Debug,
    STATE: Default + Clone,
    CTX: Clone,
{
    id: IdentificationValue,
    causal_type: CausaloidType,
    causal_fn: Option<CausalFn<I, O>>,
    coll_aggregate_logic: Option<AggregateLogic>,
    coll_threshold_value: Option<NumericalValue>,
    context_causal_fn: Option<ContextualCausalFn<I, O, STATE, CTX>>,
    context: Option<CTX>,
    causal_coll: Option<Arc<Vec<Self>>>,
    causal_graph: Option<Arc<CausaloidGraph<Self>>>,
    description: String,
    _phantom: PhantomData<(I, O, STATE, CTX)>,
}
```

Four generic parameters do real work. `I` is the input type; `O` is the output effect's value type. `STATE` carries any per-evaluation state the rule wants to thread through. `CTX` is the context type. For the common case where you do not need state or context, the `BaseCausaloid<I>` alias pins them to `()` and `BaseContext`.

The structure

A Causaloid is one of three shapes, recorded in `CausaloidType`:

- **Singleton**: a single causal function. The atomic case.
- **Collection**: a native Rust collection of Causaloids evaluated together under an `AggregateLogic` (conjunction, disjunction, threshold). The "many rules, one decision" case. Slices, `VecDeque`, `HashMap`, and `BTreeMap` all pick up the `MonadicCausableCollection` blanket impl, so any of them works.
- **Graph**: a proper `CausaloidGraph<Self>` hypergraph (backed by `ultragraph`). The "rules with structure" case. Order and reachability of evaluation matter, and a single hyperedge can connect more than two Causaloids at once.

The three shapes are *isomorphic-recursive*. A Singleton, a Collection, and a Graph are distinct concrete structures, yet each one implements the same `Causable` and `MonadicCausable` trait surface. As far as the rest of the library is concerned, each one *is* a Causaloid. That uniformity is what makes them composable into each other. A Causaloid wrapping a Graph can be a node in another Graph, an entry in a Collection, or the operand of a `bind` step. The structure nests to arbitrary depth without the calling code changing shape.

This is the central representational move. Classical causality frameworks force you to pick a structure up front. A Pearl SCM is a graph. A Granger model is a set. A Bayesian network is a graph with a specific edge semantics. Changing the structure means rewriting the model. DeepCausality lets you choose your structure for any specific problem, combine different structures for complex cases, and encapsulate sub-modules into single Causaloids to make larger models manageable and composable.

Construction

```rust
use deep_causality::{
    AggregateLogic, BaseCausaloid, BaseContext, Causaloid, CausaloidGraph,
PropagatingEffect,
};
use std::sync::Arc;

// 1. Stateless, no context. The default for simple rules.
let above_zero: BaseCausaloid<f64> = Causaloid::from_causal_fn(
    1,
    "above_zero",
    "value is greater than zero",
    |x: &f64| Ok(PropagatingEffect::Deterministic(*x > 0.0)),
);

// 2. Contextual. The Causaloid captures an Arc<Context>; the closure
//    receives both the input and the captured context on every call.
let with_ctx: BaseCausaloid<Tick> = Causaloid::from_contextual_causal_fn(
    2, "name", "description", ctx,
    |t: &Tick, c: &Arc<BaseContext>|
Ok(PropagatingEffect::Deterministic(true)),
);

// 3. Collection. Aggregates a set of Causaloids under one rule.
let any_of: BaseCausaloid<f64> = Causaloid::from_causal_collection(
    3, "any_of", "any predicate fires",
    vec![above_zero.clone(), other_rule.clone()],
    AggregateLogic::Or,
    None,
);

// 4. Graph. Edges encode dependencies between rules.
let mut g = CausaloidGraph::new();
let root = g.add_root_causaloid(stage_one);
let next = g.add_causaloid(stage_two);
g.add_edge(root, next)?;
let pipeline: BaseCausaloid<Tick> = Causaloid::from_causal_graph(
    4, "pipeline", "two-stage signal", g,
);
```

Each constructor returns the same `Causaloid` type. They differ only in which of the four optional fields are populated. The discriminant is `causal_type`.

Evaluation

`evaluate` runs the Causaloid against an input and returns a `PropagatingEffect`:

```
let effect = pipeline.evaluate(&tick)?;
```

For a Singleton the result is the function's return value. For a Collection the per-element effects are reduced under `AggregateLogic`. For a Graph the children are evaluated in topological order against the parent's effect, and the final node's effect is returned.

Errors short-circuit the chain. The `EffectLog` accumulates regardless, so a failed run still produces an audit trail of where it failed.

Where to look next

Context is the structure a contextual Causaloid reads from. Effect Propagation Process is the carrier effect that flows through a Causaloid chain. Causal Monad is the `pure`/`bind` algebra that carrier implements, so chains compose without losing their properties.

Context

A `Context` is the explicit environment a Causaloid reasons against. It is the half of dynamic causality that holds the world while the rules hold the structure.

Concretely, a Context is a typed weighted hypergraph whose nodes are `Contextoids`. Most production work uses one Context per system, mutated in place across the lifetime of a run.

The type

The Rust definition lives in deep_causality/src/types/context_types/context_graph/mod.rs:

```
pub struct Context<D, S, T, ST, SYM, VS, VT>
where
    D: Datable + Clone,
    S: Spatial<VS> + Clone,
    T: Temporal<VT> + Clone,
    ST: SpaceTemporal<VS, VT> + Clone,
    SYM: Symbolic + Clone,
    VS: Clone,
    VT: Clone,
{
    id: ContextId,
    name: String,
    base_context: UltraGraphWeighted<Contextoid<D, S, T, ST, SYM, VS, VT>,
u64>,
    id_to_index_map: HashMap<ContextoidId, usize>,
    extra_contexts: Option<ExtraContextMap<…>>,
    number_of_extra_contexts: u64,
    extra_context_id: u64,
    current_data_map: HashMap<usize, usize>,
    previous_data_map: HashMap<usize, usize>,
    current_index_map: HashMap<usize, usize>,
    previous_index_map: HashMap<usize, usize>,
}
```

Seven generic parameters look intimidating; they are how the library remains polymorphic across Euclidean, non-Euclidean, temporal, and abstract relational settings. The `BaseContext` alias pins all seven to sensible defaults, and most code reaches for the alias.

Contextoids

A `Contextoid` is the atomic unit of context. It carries an id and a typed payload:

- **`Datoid`**: arbitrary data with a name and value. The everyday case for tunable thresholds, model parameters, current state snapshots.
- **`Spaceoid`**: a spatial position or region in the chosen space type.
- **`Tempoid`**: a temporal position or interval.
- **`SpaceTempoid`**: a combined spacetime point or extent.
- **`Symboid`**: a symbolic entity (a label, a category, an external reference).

Contextoids are *not* recursive. A Contextoid cannot contain another Contextoid. The monograph treats this as a deliberate guard against self-referential paradox; the engineering payoff is that walking the graph stays predictable.

Adding nodes and edges

```
use deep_causality::{BaseContext, Context, Contextoid, ContextoidType,
Data, Time};
use std::time::SystemTime;

let mut ctx: BaseContext = Context::with_capacity(1, "trading", 64);

let threshold = Contextoid::new(
    10,
    ContextoidType::Datoid(Data::new(10, "volume_threshold".into(), 1_500.0)),
);
let now = Contextoid::new(
    20,
    ContextoidType::Tempoid(Time::new(20, "now".into(), SystemTime::now())),
);

let i_thresh = ctx.add_node(threshold);
let i_now = ctx.add_node(now);
ctx.add_edge(i_thresh, i_now, /* weight = */ 1)?;
```

Nodes are addressed by id (`u64`) at the public surface; the library maintains a private id→index map so queries stay O(1) regardless of insertion order. Edges carry a weight; the `u64` default suits most use cases and can be lifted by reaching past the `BaseContext` alias.

Mutating in place

This is what makes the model dynamic.

```
let updated = Contextoid::new(
    10,
    ContextoidType::Datoid(Data::new(10, "volume_threshold".into(), 3_000.0)),
);
ctx.update_node(10, updated)?;
```

The Causaloids that read from this Context do not need to be rebuilt. They evaluate against whatever the Context currently holds. The `previous_data_map` field on `Context` preserves a one-step history, so a rule can compare *now* against *just-before-now* when the change itself is the relevant signal.

Adjusting nodes in place

Mutation by full replacement is the coarse case. Production systems often need something finer: the Context graph is structurally fixed, but the incoming sensor data is irregular. A feed drops packets. A reading drifts. A reading arrives but is known to be biased. You want to correct what is already in the node rather than rebuild the node.

The `Adjustable` trait is the seam for that. Each Contextoid payload that admits correction implements two methods:

- `update`: replace the stored value outright with a value supplied from an `ArrayGrid`. The default implementation under `context_node_types` sanity-checks the incoming value and rejects it if it is the type's default (a zero sentinel), preventing accidental wipes. Use this when you have detected the stored value is invalid and you want to overwrite it.
- `adjust`: apply a correction relative to the stored value. The default implementation adds a delta from the supplied `ArrayGrid` and rejects the result if it would go negative. Use this when the stored value is approximately right but a drift has been observed and needs correcting.

Both methods are const-generic over the grid dimensions (`WIDTH`, `HEIGHT`, `DEPTH`, `TIME`), so the correction data can be 1D for a scalar, 2D for a spatial frame, 3D for a volumetric field, or 4D for a spacetime patch. The trait default does nothing, so a node type that should never be touched at runtime is correct by default.

A companion trait, `UncertainAdjustable`, covers nodes whose payload is an `Uncertain<T>` rather than a fixed value. It takes a typed `Data` argument instead of an `ArrayGrid` and is the right hook when the correction itself carries uncertainty.

The split between *update* and *adjust* is deliberate. Replacement is destructive and asymmetric. Adjustment is incremental and preserves whatever calibration was already in the node. Mixing them at the same call site would obscure intent, so the trait surfaces them as two separate methods and lets the caller pick by name.

Counterfactuals via extra contexts

The `extra_contexts` field carries parallel hypothetical contexts. Build a counterfactual the same way you build the primary Context, register it under an `extra_context_id`, and evaluate the same Causaloid against it.

```
let alt_id = ctx.add_extra_context();
ctx.with_extra(alt_id, |alt| {
    alt.update_node(10, /* counterfactual threshold */)?;
    Ok(())
})?;
let alt_effect = signal.evaluate_with_extra(&tick, alt_id)?;
```

Nothing on the primary Context is disturbed. The library treats counterfactual reasoning as a configuration of the same machinery rather than as a separate engine.

When to add to the Context

A value belongs in the Context when one of these is true:

- The value changes during a run, and the rules need to see the change.
- The value is set externally and tunable by an operator.
- The value is something a counterfactual run might want to replace.
- The value is a shared piece of state that more than one Causaloid reads from.

Values that fail every test stay in the closure. The Context is the structured shared state for a causal model and is designed to be accessed from multiple Causaloids. It is also a foundational pillar of dynamic causality.

Where to look next

Causaloid is the rule that reads the Context. Effect Propagation Process is what the rule produces. Effect Ethos is what verifies the rule's output before it commits.

Effect Ethos

A Causaloid or Causal Monad determines what the system *infers*. The Causal State Machine (CSM) translates that inference into a proposed action. The Effect Ethos says whether that proposed action is *allowed to execute*. The three layers encode different responsibilities.

The three-layer flow

DeepCausality separates reasoning from action by design:

1. **Reasoning**: a Causaloid (Singleton, Collection, or hypergraph) and the Causal Monad produce a `PropagatingEffect`. This is the inference layer. It answers "what is the case?"
2. **Action proposal**: the CSM reads the propagating effect, evaluates which of its registered causal states have become active, and constructs a `ProposedAction` for each active state. This is the bridge between inference and the outside world. It answers "what to do next?"
3. **Action verification**: the Effect Ethos intercepts every proposed action before it fires. It evaluates the proposal against a graph of Teloids and returns a `Verdict`. This is the guardrail layer. It answers "is the system *allowed* to do that here, now, under these rules?"

The Effect Ethos is what safeguards the CSM. An action that the CSM would otherwise fire does not fire if the Ethos returns an impermissible verdict.

The Origin of the Effect Ethos

The Effect Ethos exists because dynamic, emergent causality is intrinsically non-deterministic. The Dynamic causality page lays out the four reasoning modalities and shows where the determinism boundary breaks: static, dynamic, and adaptive reasoning are formally verifiable, but emergent reasoning is not. The causal graph can rewire itself at runtime in ways no upfront proof can cover. Once that capability is on the table, "the reasoning is correct" stops being a property you can establish in advance. Something else has to absorb the verification burden.

The realization was that every action the system is about to take needs an independent check, expressed in rules that do not change while the reasoning evolves. The check has to handle conflicting rules, audit trails, and overrides without losing determinism. That is a deontic problem, not a causal one, and it had already been studied.

The answer came from Olson, Salas-Damian, and Forbus at Northwestern University in *A Defeasible Deontic Calculus for Resolving Norm Conflicts*. The paper introduces the Defeasible Deontic Inheritance Calculus (DDIC): a formalism for resolving a continuous stream of possibly conflicting norms. It defines the three deontic modalities (Obligatory, Optional, Impermissible), characterizes the three conflict types (direct, indirect, intersecting), and proves that three resolution heuristics (Lex Specialis, Lex Posterior, Lex Superior) are sufficient to axiomatize conflict resolution under deontic inheritance. The paper goes further and shows that one widely used multi-agent strategy is a red herring once defeasance is modeled correctly.

The Effect Ethos is an implementation of DDIC inside DeepCausality. A `Teloid` is one norm tuple from the calculus. The `TeloidGraph` carries the inheritance and defeasance edges DDIC requires. `evaluate_action` runs the activation, conflict detection, and resolution steps from the paper, in the order DDIC prescribes, and returns a `Verdict` whose justification field is the audit trail the formalism implies. The mapping is intentional. DDIC gave a theoretically justified axiomatization of norm conflict detection and resolution; the Effect Ethos makes it runnable, embeds it in a typed context, and wires it to the Causal State Machine so that every proposed action passes through the calculus before it executes.

The problem the Ethos solves

The Effect Ethos pulls scattered checks into one structured object that can answer the thorny questions. The CSM hands every proposed action to the Ethos before execution. The Ethos returns a verdict carrying both the outcome and the chain of Teloids that justified it.

What it is

The `EffectEthos` lives in the `deep_causality_ethos` crate. The shape:

```
pub struct EffectEthos<D, S, T, ST, SYM, VS, VT>
where
    /* … same context bounds as Context … */
{
    teloid_store: TeloidStore<D, S, T, ST, SYM, VS, VT>,
    tag_index: TagIndex,
    teloid_graph: TeloidGraph,
    id_to_index_map: HashMap<TeloidID, usize>,
    is_verified: bool,
}
```

Two parts to read. The `teloid_store` holds the active rules. The `teloid_graph` and `tag_index` make the rules navigable, both by id and by category.

A `Teloid` is the atom inside the store. It is a computable unit of purpose, and it instantiates one norm tuple from the DDIC calculus described in the origin section above. Concretely a Teloid carries:

- A **deontic modality**: `Obligatory`, `Impermissible`, or `Optional(cost)`.
- A **condition** evaluated against the Context (either a deterministic Causaloid or an `Uncertain` predicate).
- A **scope tag** used by the CSM to filter which Teloids apply to a given proposed action.
- An **id** that survives logging and shows up in every audit trail.

The Teleology preprint introduces Teloids as the answer to the question, “What stops an emergent system from inferring its way into a state you cannot let it act on?”

Building an Ethos

Norms are added through `add_deterministic_norm` (for deterministic Causaloid conditions) or `add_uncertain_norm` (for `Uncertain` predicates). Inheritance and defeasance edges between Teloids are wired up with `link_inheritance` and `link_defeasance`. Before the Ethos can be queried by a CSM, it must be `freeze()`’d. Freezing finalizes the Teloid graph the same way it finalizes a Causaloid graph, switching the underlying `ultragraph` backend to its query-optimized CSR form.

```rust
use deep_causality_ethos::EffectEthos;

let mut ethos: EffectEthos<_, _, _, _, _, _, _> = EffectEthos::new();
// ethos.add_deterministic_norm(...);
// ethos.link_inheritance(general_id, specific_id);
ethos.freeze();
```

The full API is on docs.rs.

Evaluating a proposed action

The CSM hands a `ProposedAction` to the Ethos along with the current Context and the scope tags that apply to the proposal. The Ethos returns a `Verdict`:

```rust
use deep_causality_ethos::{DeonticInferable, TeloidModal};

let verdict = ethos.evaluate_action(&proposed_action, &context, &tags)?;

match verdict.outcome() {
    TeloidModal::Obligatory      ⇒ csm.fire(proposed_action)?,
    TeloidModal::Impermissible   ⇒ csm.reject(verdict.justification())?,
    TeloidModal::Optional(cost) ⇒ csm.fire_if_within_budget(proposed_action,
*cost)?,
}
```

The verdict carries both the outcome and a justification: the ordered list of Teloid ids that produced it. The CSM uses the outcome to decide whether to fire; the justification goes into the audit log so any decision the system makes can be replayed later with the same Context and the same Ethos.

Conflict resolution

Real rule sets contradict each other. Two requirements both apply, one says obligatory, the other says impermissible. The Effect Ethos resolves the contradiction with three principles:

- **Lex Posterior**: the later-issued rule wins over the earlier one.
- **Lex Specialis**: the more specific rule wins over the more general one.
- **Lex Superior**: the higher-priority rule wins over the lower-priority one.

These run in a fixed order when the Ethos is asked to reconcile a conflict. The combination is enough to handle most rule-set evolution in practice without giving up determinism. The proof that these three heuristics are sufficient, and the analysis of which common resolution strategies they subsume, comes from the DDIC paper cited above.

Why this is the right place to guardrail

The Effect Ethos can disagree with the inference layer, and the disagreement is the point.

A Causaloid graph reasons forward from inputs to a propagating effect. It is concerned with *what is inferable*. The CSM translates that inference into a *proposed action*. The Ethos reasons against the proposal from operational constraints. It is concerned with *what is permissible*. The two answers disagree often enough to be worth modelling separately. When they agree, the CSM fires. When they disagree, the rejection is structured and explainable.

This is what restores verifiability under emergent reasoning. The Dynamic causality page breaks the four reasoning modalities at the determinism boundary: static, dynamic, and adaptive reasoning are all formally verifiable, while emergent reasoning is not. The reasoning graph may evolve in ways no static proof can foresee. The Effect Ethos accepts that reality and moves the verifiability checkpoint to the one place where it stays feasible: the action layer. Every action the CSM proposes is checked against an immutable Ethos before it leaves the system. The reasoning is free to be emergent; the actions are not.

Where to look next

Causal State Machine is the layer that proposes the actions the Ethos checks. Causaloid is the inference layer the CSM reads from. The API reference is on docs.rs at `deep_causality_ethos`.

Two papers ground the design:

- Olson, T., Salas-Damian, R., and Forbus, K. D. *A Defeasible Deontic Calculus for Resolving Norm Conflicts*, Northwestern University. The DDIC formalism. This is where the Effect Ethos comes from.
- The Teleology preprint shows how DDIC is embedded into the Effect Propagation Process and the DeepCausality runtime.

Effect Propagation Process

The Effect Propagation Process (EPP) is the load-bearing abstraction of DeepCausality. It is the carrier effect: the value that flows between every other piece of the library. The Causaloid, the Context, the Causal State Machine, and the Effect Ethos all exchange work through one type, and that type implements the Causal Monad trait so it composes:

```
pub struct CausalEffectPropagationProcess<Value, State, Context, Error,
Log> {
    pub value:   EffectValue<Value>,
    pub state:   State,
    pub context: Option<Context>,
    pub error:   Option<Error>,
    pub logs:    Log,
}
```

This is the runtime realization of the theory described in the Effect Propagation Process preprint. The paper reframes causality as a spacetime-agnostic functional dependency between an input and a propagated effect. The struct above is that dependency made concrete and runnable.

What the EPP contributes

Most causal libraries split reasoning across several incompatible vocabularies. Structural causal models live in one type. Sequential probabilistic chains live in another. State is held in a host struct. Errors propagate through `Result`. Logs end up in a tracing subscriber. Context is encoded implicitly. The EPP collapses that fragmentation. One container carries everything a chain needs to know about itself:

1. **Unified carrier for heterogeneous reasoning.** Structural reasoning (a Causaloid Singleton, Collection, or Graph) and sequential reasoning (a Causal Monad bind-chain) both return an EPP. The two reasoning styles share a single boundary type, which is what makes them composable without bridge code. A graph emits an EPP, a `bind` consumes it, another graph evaluates against the result. Nothing translates between worlds.

2. **Non-Markovian and Markovian under one type.** The same struct represents both. `PropagatingEffect<T>` fixes `State = ()` and `Context = ()`; each step depends only on its input, and the chain stays Markov-free. `PropagatingProcess<T, S, C>` keeps the

state and context generic; each step receives the threaded state and context, and the chain becomes Markovian. Lifting between the two is a single constructor call. The boundary is real, but it is movable.

3. **Audit and replay.** Because the EPP carries the log inline with the value, every step appends to the same record, and a chain can be replayed off disk with no missing context. There is no separate tracing infrastructure to align, no out-of-band state to reconstruct.

The EffectValue Type

`value`: the propagating effect's payload, wrapped in an `EffectValue<T>` enum:

```
pub enum EffectValue<T> {
    None,
    Value(T),
    ContextualLink(ContextoidId, ContextoidId),
    RelayTo(usize, Box<PropagatingEffect<T>>),
    #[cfg(feature = "std")]
    Map(HashMap<IdentificationValue, Box<PropagatingEffect<T>>>),
}
```

`None` is an explicit *no effect*. `Value(T)` is the everyday case. `ContextualLink` says "the value is whatever the Context says it is at these two ids" and defers the fetch. `RelayTo` is a dispatch command: route this effect to the rule at index N. `Map` carries a labeled bundle of sub-effects.

A Causaloid's wrapped function returns a `PropagatingEffect<T>` whose `value` is one of those variants. The richer variants exist so that downstream rules can do work for the upstream rule without losing the audit trail in between.

`state`: caller-supplied state threaded through the chain. For the stateless case, `State = ()` and the field carries no information.

`context`: an optional `Context` value. When a contextual Causaloid runs it threads the Context through here; when a stateless rule runs it stays `None`.

`error`: `Option<Error>`. The chain short-circuits when this is `Some`. The presence of an error does not stop the log from accumulating; the failure point is recorded with everything else.

`logs`: an append-only `EffectLog`. Every Causaloid that runs adds an entry. The log is the audit trail.

The aliases

You will rarely instantiate the five-parameter form by hand. The library ships two pinned aliases.

```
// Stateless, contextless. The everyday case.
pub type PropagatingEffect<T> =
    CausalEffectPropagationProcess<T, (), (), CausalityError, EffectLog>;
```

```
// Stateful, with a typed context. The dynamic case.
type CausalProcess<T, S, C> =
    CausalEffectPropagationProcess<T, S, C, CausalityError, EffectLog>;
```

`PropagatingEffect<T>` is what a `Causaloid::from_causal_fn` closure returns. `CausalProcess<T, S, C>` is the stateful form a chain operates over when it threads state and context through `bind`.

How the process moves

A single Causaloid call takes an input, produces a `PropagatingEffect`, and returns. The "process" emerges when Causaloids compose. Each rule in the chain consumes the upstream effect, performs its own computation, and produces a new effect. The chain accumulates:

- The latest `value`.
- The threaded `state` (updated in place when stateful).
- The shared `context` (mutable or readonly depending on the configuration).
- The first encountered `error`, after which propagation stops.
- The growing `logs`, regardless of error state.

The composition is provided by the Causal Monad and its `bind` operation. Conceptually:

```
m₁ >>= f   →   m₂
```

m_1 is the upstream `CausalEffectPropagationProcess`. `f` is the next Causaloid's function. m_2 is the new process: the new value sits in m_2`.value`, the threaded state in m_2`.state`, the merged logs in m_2`.logs`, and any error surfaces in m_2`.error`.

Inspecting an effect

A consumer typically pattern-matches on `EffectValue`:

```
match effect.value {
    EffectValue::Value(v)                 ⇒ commit(v)?,
    EffectValue::None                     ⇒ skip(),
    EffectValue::ContextualLink(a, b)     ⇒ resolve_link(&ctx, a, b)?,
    EffectValue::RelayTo(idx, sub)        ⇒ dispatch(idx, *sub)?,
    EffectValue::Map(parts)               ⇒ fan_out(parts)?,
}
```

The `error` field is checked before this match. The `logs` field is appended to the persistent audit log on every emission regardless of outcome.

Why a five-field record

The five fields are the irreducible set. Drop any one and a contribution from the list above collapses:

- Without **`value`** there is nothing to propagate.
- Without **`state`** the Markovian case cannot be expressed without a separate type, and the unification falls apart.
- Without **`context`** spatial, temporal, and symbolic conditioning leak out into ambient state.
- Without **`error`** the chain cannot short-circuit cleanly, and partial-failure replay becomes guesswork.
- Without **`logs`** audit and replay stop being intrinsic and become an external concern again.

DeepCausality keeps the five together to enable verifiable end-to-end reasoning. A test that replays an effect off disk has everything. A debugger that wants to step backward through a propagation has everything for fine-grained diagnostics.

Where to look next

Causal Monad is the algebra that composes processes. HKT is how the algebra is encoded in Rust's type system. Causaloid is what produces the processes in the first place. Effect Ethos is what verifies the actions an EPP chain ultimately proposes. The Effect Propagation Process preprint is the formal treatment of the model this page implements.

Causal Monad

The Causal Monad is the algebra the carrier effect already carries encoded as a Rust trait. `PropagatingEffect<T>` and `PropagatingProcess<T, S, C>` implement the `CausalMonad` trait, and that trait is what lets a chain of Causaloids compose without losing their properties. Each Causaloid is a step. The trait is the law for how the steps combine.

The whole algebra is two operations: `pure` and `bind`. The rest follows.

The axiom

From the EPP preprint:

> A causal relation is a monadic dependency, in which one propagating effect is obtained from another by composition with a causal function in a monadic context of the causal process.

The equation:

```
m₂ = m₁ >>= f
```

Here `>>=` is `bind`, `m₁` and `m₂` are propagating effects, and `f` is a causal function. The carrier effect implements that operator over the five-field `CausalEffectPropagationProcess` struct.

A trait, not a primitive

`CausalMonad` is a trait. There is no `CausalMonad` value, no struct to instantiate, no third primitive sitting beside the Causaloid and the Context. The carrier effect *is* the monad:

```
pub trait CausalMonad: Sized {
    type Value;
    type State;
    type Context;

    fn pure(value: Self::Value) → Self;

    fn bind<NewValue, F>(self, f: F)
        → CausalEffectPropagationProcess<NewValue, Self::State, Self::Context,
CausalityError, EffectLog>
    where
        F: FnOnce(
            EffectValue<Self::Value>,
            Self::State,
            Option<Self::Context>,
        ) → CausalEffectPropagationProcess<NewValue, Self::State,
Self::Context, CausalityError, EffectLog>;
}
```

It is implemented once, for `CausalEffectPropagationProcess<Value, State, Context, CausalityError, EffectLog>`. That single impl covers both aliases: the stateless `PropagatingEffect<T>` (where `State = Context = ()`) and the stateful `PropagatingProcess<T, S, C>`. There is exactly one `bind`, and it threads state.

The same two operations are also exposed as inherent methods on the carrier, so everyday code writes `PropagatingEffect::pure(x)` and `effect.bind(...)` without importing the trait. The trait exists so generic code can bind against the contract, and so the type signature states the intent: this is a state-threading monad.

pure

`pure` lifts a plain value into the carrier. The returned process has:

- `value = EffectValue::Value(value)`
- `state = State::default()`
- `context = None`
- `error = None`
- `logs = EffectLog::default()`

This is the seed for a chain. Most chains start with `PropagatingEffect::pure(input)` and immediately bind.

bind

`bind` chains the next step. Its continuation receives three things: the upstream value wrapped in `EffectValue`, the threaded state, and the optional context. It returns the next process, and that process's state and context carry forward.

```
let next = effect.bind(|value, state, context| {
    // inspect value, read context, evolve state, return the next process
    ...
});
```

`bind` does these things, in order:

1. If the upstream `error` is `Some`, short-circuit. Return a process with the same error, the carried state and context, and the existing logs; the value becomes `EffectValue::None`. No fabricated default value is invented.
2. Otherwise call the continuation with the value, state, and context.
3. Merge the upstream logs into the next process's logs via `LogAppend::append`. The audit trail grows; entries do not vanish across binds.
4. Keep the state and context of the process the continuation returned. This is what makes the chain Markovian when it needs to be: a step can read the running state, update it, and the update survives into the next step.

The earlier design split this into two binds: a value-only effect-system bind that could not thread state, and a separate state-threading one. The value-only form froze the Markovian state, so it was removed. The trait now is the contract, and there is one `bind` that threads state correctly for both the stateless and the stateful carrier.

fmap

When a step only transforms the value and has no reason to touch state, context, or error, `fmap` is the lighter operation:

```
let doubled = effect.fmap(|x| x * 2);
```

`fmap` maps the value and passes state, context, and logs through unchanged. It short-circuits on error like `bind`, so it never panics on an errored or empty carrier. Reach for `bind` when a step needs the state or context; reach for `fmap` when it does not.

A minimal example

```
use deep_causality::PropagatingEffect;

let final_process = PropagatingEffect::pure(10)
    .bind(|value, _state, _context| {
        let n = value.into_value().unwrap_or_default();
        let mut next = PropagatingEffect::pure(n + 1);
        next.logs.add_entry("step 1");
        next
    });

assert_eq!(final_process.value.into_value(), Some(11));
assert_eq!(final_process.logs.len(), 1);
```

Two binds and you have a chain. Five binds and you have a pipeline. Five hundred and you have a system.

The monad laws

A monad earns the name by satisfying three identities. The carrier satisfies them.

Left identity. `pure(a).bind(f)` is equal to `f(a)`. Wrapping a value and immediately binding is the same as just calling the function.

Right identity. `m.bind(pure)` is equal to `m`. Binding `pure` at the end is a no-op.

Associativity. `m.bind(f).bind(g)` is equal to `m.bind(|v, s, c| f(v, s, c).bind(g))`. Grouping does not change the result.

The library's test suite covers these explicitly. The point of the laws in practice: you can refactor a chain freely, pull a step out, inline a step in, regroup, and the meaning does not change.

Stateless and stateful, one algebra

`PropagatingEffect<T>` pins `State` and `Context` to `()`, so its bind threads the unit state trivially; the chain stays Markov-free. `PropagatingProcess<T, S, C>` keeps both generic, so its bind threads real state and context. They are the same algebra over the same struct. Lifting from the stateless to the stateful form is a single constructor call, `with_state`.

```
pub type PropagatingEffect<T> =
    CausalEffectPropagationProcess<T, (), (), CausalityError, EffectLog>;
```

A Causaloid returning a `PropagatingEffect` is returning a value that already implements this trait.

Why this matters

Three concrete payoffs.

Short-circuiting on error costs nothing. The first failed step turns into an `error.is_some()` on the carried process, and every subsequent `bind` is a no-op that preserves the logs. You do not write `?` propagation by hand inside the chain.

Logs accumulate without instrumentation. `LogAppend::append` runs inside every bind. A consumer that wants to print or persist the trace gets the full ordered sequence with no side channel.

Refactoring stays safe. The laws guarantee that breaking a long chain into helper functions, or composing several chains into a larger one, does not change the result. You get the refactoring confidence that pure functional code usually offers.

Where to look next

Effect Propagation Process is the carrier this trait operates over. HKT explains how the signature stays generic across the five parameters without runtime cost. Causaloid is what produces the values that flow through.

Causal Flow

`CausalFlow` is the fluent API over the Causal Monad. The monad is the algebra: `pure` and `bind` over the carrier effect. Written out by hand, a monadic pipeline exhibits real complexity. You wrap values in `EffectValue`, call `pure` and `with_state`, unwrap with `into_value().unwrap_or_default()`, and check the error channel between steps. `CausalFlow` hides all of that behind a much simplified fluent API.

```rust
use deep_causality::CausalFlow;

let outcome = CausalFlow::value(2_i64)
    .try_step(|x| Ok(x + 3))
    .map(|x| x * 10)
    .finish();

assert_eq!(outcome, Ok(50));
```

That is the same chain you would write with `PropagatingEffect::pure(2).bind(...).bind(...)`, with all the wrapping removed.

Causal Monad, Simplified

Every `CausalFlow` verb lowers to an existing monad operation. `value` lowers to `pure`. `map` lowers to `fmap`. `try_step`, `branch`, `iterate_n`, and the rest lower to `bind`. Under the hood, everything is still the causal monad with all its expressiveness. It is just the interface that has been simplified.

Two consequences follow. First, the monad laws still hold, so a flow refactors as freely as the chain underneath it. Second, you can drop in and out of the DSL at any point: `from(process)` lifts an existing process into a flow, and `into_process()` / `into_effect()` drop back to the concrete carrier for code that expects it.

The Fluent API

The surface groups into six families.

Seed. `value(v)` starts a stateless flow carrying a value. `process(s)` starts a stateful flow seeding the state channel. `effect()` seeds the unit value. `from(process)` lifts an existing

carrier, and `.context(c)` attaches a read-only context.

Step. `map(|v| u)` transforms the value. `try_step(|v| Result)` runs a fallible step. `and_then` / `next` compose a whole sub-pipeline (`Value → CausalFlow<U>`). `guard(|&v| Result)` validates without changing the value. `update_state` / `update_context` evolve a single channel. `bind` is the raw monad passthrough for an existing stage signature.

Branch. `branch(cond, on_true, on_false)` routes by a test on the value. `branch_with` tests the value, state, and context together. `either` routes a flow whose value is `Either<L, R>`. Each arm is itself a flow, so a branch arm can be a full sub-pipeline.

Loop. `iterate_n(n, step)` runs a step a fixed number of times. `iterate_until(pred, max, step)` runs until a predicate holds. `iterate_to_fixpoint(max, step)` runs until the value stops changing. The two open-ended forms take a step bound and fail with a `MaxStepsExceeded` error rather than spinning forever.

Intervene. `intervene(v)` force-substitutes the value, Pearl's `do(v)`, and records the override in the audit log. `intervene_if(cond, f)` does it only when a test holds.

Finish. `finish()` returns `Result<Value, CausalityError>`. `run(on_ok, on_err)` dispatches to handlers. `is_err()` peeks at the error channel. `into_process()` / `into_effect()` return the concrete carrier.

A control loop

`iterate_n` and `branch` together express a bounded loop whose body decides what to do each tick. The arms are flows, so the branch reads as two small pipelines.

```rust
use deep_causality::CausalFlow;

let total = CausalFlow::value(0_i64)
    .iterate_n(5, |tick| {
        tick.branch(
            |n| n % 2 == 0,              // is the value even?
            |even| even.map(|n| n + 10), // yes: add 10
            |odd| odd.map(|n| n + 1),    // no: add 1
        )
    })
    .finish();

assert_eq!(total, Ok(50));
```

This is the shape the avionics and corrective-control examples take: one `iterate_n` for the loop, `next` to wire per-tick stages, and a `branch` for the conditional intervention.

Factual and counterfactual

```rust
use deep_causality::CausalFlow;

// factual: the reading is 8, the pipeline scales it
let factual = CausalFlow::value(8_i64).map(|x| x * 2).finish();

// counterfactual: what if the reading had been clamped to 0?
let counterfactual = CausalFlow::value(8_i64)
    .intervene(0) // do(reading = 0)
    .map(|x| x * 2)
    .finish();

assert_eq!(factual, Ok(16));
assert_eq!(counterfactual, Ok(0));
```

Named stages compose

`next` composes a sub-pipeline, a function `Value → CausalFlow<U>`. Pulling each step into a named function keeps a long pipeline readable, and the laws guarantee the meaning does not change.

```rust
use deep_causality::CausalFlow;

fn scale(x: i64) -> CausalFlow<i64> {
    CausalFlow::value(x * 10)
}

let out = CausalFlow::value(5_i64).next(scale).map(|x| x + 1).finish();
assert_eq!(out, Ok(51));
```

The error channel is automatic

A failing step moves the flow into its error channel, and every later verb becomes a no-op that carries the error and the accumulated log forward. You do not write `?` between steps.

```rust
use deep_causality::{CausalFlow, CausalityError, CausalityErrorEnum};

let outcome = CausalFlow::value(-1_i64)
    .try_step(|n| {
        if n >= 0 {
            Ok(n)
        } else {
            Err(CausalityError::new(CausalityErrorEnum::Custom("negative
input".into())))
        }
    })
    .map(|n| n * 2) // skipped
    .finish();

assert!(outcome.is_err());
```

Stateless and stateful

`CausalFlow<Value>` defaults its state and context to `()`, the stateless form that lowers to `PropagatingEffect`. Seed it with `value(v)`. When a pipeline needs memory, seed it with `process(s)` and evolve the state; the flow now lowers to `PropagatingProcess` and threads state Markovian-style.

```rust
use deep_causality::CausalFlow;

let final_process = CausalFlow::process(0_i64) // state = 0
    .update_state(|state, _value| state + 1)
    .into_process();

assert_eq!(final_process.state, 1);
```

The verbs are identical across both forms. The type parameters decide whether the chain carries memory, exactly as they do for the monad underneath.

Where to look next

Causal Monad is the `pure`/`bind` algebra this DSL is sugar over. Effect Propagation Process is the carrier both operate on. Counterfactuals covers `intervene` and Pearl's Ladder in full. For the hands-on introduction, start with Hello, Causal Flow.

Higher-Kinded Types

A higher-kinded type is a type that takes another type as a parameter and produces a type. For historical reasons, the Rust team decided against including higher-kinded types into the Rust programming language. However, with the introduction of the causal discovery language, monadic composition becomes a viable alternative, and that also enabled the causal monad and effect propagation process. Therefore, the Deep Causality Project decided to include a higher-kinded type implementation in a dedicated crate that uses the witness pattern and a trait hierarchy to establish arity-five higher-kinded types and the corresponding effect.

The encoding

The crate deep_causality_haft defines the trait hierarchy:

```
pub trait HKT {
    type Constraint: ?Sized;
    type Type<T>
    where
        T: Satisfies<Self::Constraint>;
}

pub trait HKT2<F> {
    type Type<T>;
}


pub trait HKT3<Fixed1, Fixed2> {
    type Type<T>;
}

pub trait HKT4<F1, F2, F3> {
    type Type<T>;
}

pub trait HKT5<Fixed1, Fixed2, Fixed3, Fixed4> {
    type Type<T>;
}
```

Each variant fixes some parameters and varies the rest. `HKT5` is the one the Causal Monad uses: four slots are fixed (state, context, error, log) and the fifth (the value type) varies through `Type<T>`.

The witness pattern looks like this:

```
pub struct CausalEffectPropagationProcessWitness<S, C, E, L>(
    Placeholder,
    PhantomData<S>,
    PhantomData<C>,
    PhantomData<E>,
    PhantomData<L>,
);

impl<S, C, E, L> HKT5<S, C, E, L> for CausalEffectPropagationProcessWitness<S,
C, E, L> {
    type Type<Value> = CausalEffectPropagationProcess<Value, S, C, E, L>;
}
```

The witness type is zero-sized at runtime; the body of the impl is the trick. `Type<Value>` is the type-level function that produces the concrete process given a value type.

A second witness, `PropagatingEffectWitness<E, L>`, fixes the state and context to `()` and `()`. It is what the stateless `PropagatingEffect` alias uses.

What you actually write

Almost nothing. The witnesses live behind aliases and are not part of the day-to-day API surface. Most code looks like:

```
use deep_causality::PropagatingEffect;

let m: PropagatingEffect<i32> = PropagatingEffect::pure(10);
let n = m.bind(|value, _state, _context| {
    let x = value.into_value().unwrap_or_default();
    PropagatingEffect::pure(x + 1)
});
```

There is no `HKT5::Type<…>` in sight. The compiler resolves it. The encoding earns its keep because the *library author* could write the monad's `bind` once, generically over the witness, and

have it work for every concrete instantiation.

Why this matters

The HKT machinery is what lets the Causal Monad satisfy its laws *generically*, then specialize to a hundred concrete shapes (stateless, stateful, with this context, with that error, with that log) without rewriting the laws each time. The runtime cost is the cost of monomorphization: the compiler emits one specialized version per concrete witness, no virtual calls, no boxed type erasure.

Where to look next

Causal Monad is the user of the HKT encoding. Effect Propagation Process is the type the encoding parameterizes. The formal definitions for `HKT`, `HKT3`, and `HKT5` are on docs.rs/deep_causality_haft.

Causal Discovery Language

The Causal Discovery Language (CDL) is the DSL that bridges raw observational data and an executable causal model. It lives in the `deep_causality_discovery` crate and uses Rust's typestate pattern to encode the pipeline stages in the type system.

The library's other concepts assume you already have a Causaloid. The CDL is for the case where you do not.

Two algorithms, two lineages

CDL hosts two discovery algorithms as **compile-time-isolated lineages** that converge on a shared analyze/finalize tail:

- **SURD** (Synergistic, Unique, Redundant Decomposition): from a single dataset, decomposes how a set of source variables drive a target. It is the shipped implementation of SURD-states.
- **BRCD** (Bayesian Root-Cause Discovery): given a *normal* and an *anomalous* dataset over the same variables plus a causal graph, ranks which variable's conditional mechanism changed — the root cause of a regime shift. The graph is supplied as a CPDAG, or learned from the normal data via BOSS.

The lineages share no typestate until the converged analysis state, so calling a BRCD stage on a SURD pipeline (or the reverse) is a compile error, not a runtime one.

The problem it solves

Discovering causal structure from data is not one operation. It is a pipeline: load the data, prepare it, run a discovery algorithm, analyze the result, and finalize a report that informs how the causal model is constructed. Each stage has its own configuration, its own failure modes, and its own outputs that the next stage depends on. Doing this by hand in a notebook ends in fragile glue code; doing this with a generic pipeline framework loses the type safety that makes Rust worth using here.

The CDL keeps the type safety. Each stage returns a new typestate, and a stage method only exists on the typestate that precedes it. You cannot run discovery before preparing the data, and you cannot finalize an incomplete pipeline. The compiler refuses.

The pipeline

A run config built by `CdlConfigBuilder` is the **single source of truth**. It is a staged typestate builder: each required field is its own stage, so `build()` is reachable only once every required field is set (omitting one is a compile error), and `build()` then verifies that the referenced data files exist.

```
// SURD config: dataset path, target, MRMR feature count, max order,
thresholds.
let cfg = CdlConfigBuilder::build_surd_config::<f64>()
    .with_path("./data.csv")
    .with_target_index(3)
    .with_num_features(3)
    .with_max_order(MaxOrder::Max)
    .with_analyze(SurdAnalyzeConfig::new(0.01, 0.01, 0.01))
    .build()?;

// BRCD config: two datasets + the algorithm config; CPDAG optional (None ⇒
BOSS).
let cfg = CdlConfigBuilder::build_brcd_config()
    .with_normal_path("./normal.csv")
    .with_anomalous_path("./anomalous.csv")
    .with_brcd_config(BrcdConfig::<f64>::continuous(0))
    .with_cpdag_path("./cpdag.csv")
    .build()?;
```

`CdlBuilder::build_surd(&cfg)` / `build_brcd(&cfg)` seed the pipeline with the config and fix the precision. The stages then read their parameters from the config, so the chain is parameterless:

- **SURD**: `surd_load_input → clean_data → feature_select → surd_discover → surd_analyze → finalize`
- **BRCD**: `brcd_load_input → brcd_discover → brcd_analyze → finalize`

The final stage emits a `CdlReport` carrying a `CdlDiscoveryOutcome` (the `Surd` or `Brcd` result); its `Display` renders the matching section with edge-construction recommendations (for example, “Strong unique influence: Recommended Direct edge in `CausaloidGraph`”, or a ranked list of root-cause candidates). The report is where the pipeline ends and the model-construction workflow begins.

What the code looks like

The pipeline is a monadic sequence over `CdlEffect<T>`. Each stage is a method on the effect, so the chain reads top to bottom with no per-line wrapper; under the hood an `FnOnce` `and_then` threads the value and merges warnings, short-circuiting on the first error. The full runnable version is in [examples/causal_discovery_examples/cdl/surd_discovery/main.rs]:

```rust
use deep_causality_discovery::*;

let config = CdlConfigBuilder::build_surd_config::<f64>()
    .with_path(&file_path)
    .with_target_index(3)
    .with_num_features(3)
    .with_max_order(MaxOrder::Max)
    .with_analyze(SurdAnalyzeConfig::new(0.01, 0.01, 0.01))
    .build()?;

let result_effect = CdlBuilder::build_surd(&config)
    .surd_load_input()
    .clean_data(OptionNoneDataCleaner)
    .feature_select()
    .surd_discover()
    .surd_analyze()
    .finalize();

result_effect.print_results();
```

[CdlEffect<T>] is:

```rust
pub struct CdlEffect<T> {
    pub inner: Result<T, CdlError>,
    pub warnings: CdlWarningLog,
}
```

It carries either the next-stage `CDL< ... >` typestate or a `CdlError`, plus accumulated warnings. The HKT witness `CdlEffectWitness<CdlError, CdlWarningLog>` implements `Functor`, `Pure`, `Applicative`, and `Monad` from [deep_causality_haft]; `CdlBuilder` plugs into the `Effect3` machinery and fixes the error and warning channels.

Two layers of safety run at once. The outer `CdlEffect` monad sequences and short-circuits. The inner `CDL<State>` typestate enforces stage order and algorithm isolation: `surd_discover` exists only on the SURD-features state and `brcd_discover` only on the BRCD-loaded state, so crossing the lineages — or running discovery before the data is ready — is a compile error.

When to reach for it

You want the CDL when one of these is true:

- The causal structure is not known up front. You have data and you want the library to find the structure (SURD), or to localize a fault across a known graph (BRCD).
- You want a reproducible, type-safe pipeline rather than an exploratory notebook.
- You want one explicit, compile-checked config to drive the run.

You want to write Causaloids directly when one of these is true:

- The causal structure is known. You are encoding domain expertise, not discovering it.
- The rules need to do something a discovery algorithm cannot produce (custom conditionals, side-effecting actions, calls into other libraries).
- Performance constraints rule out the discovery phase.

Most production systems use both. The CDL produces an initial discovery report from historical data; the operator constructs the `CausaloidGraph` from those recommendations and adds rules the data does not justify on its own.

The relationship to other concepts

The CDL is a *producer of recommendations* for Causaloid graphs. SURD's unique/synergistic/redundant findings guide which edges and `AggregateLogic` to wire into the graph; BRCD's ranking points at the node whose mechanism changed, the root cause to act on. The constructed model uses the same types as a hand-written one and feeds the rest of the framework directly.

A `Context` is the engineer's job: assemble the Contextoids the discovered Causaloids should evaluate against and hand them in. The pipeline produces the recommendations; you supply the world they read from.

Where to look next

The runnable walkthroughs are in `examples/causal_discovery_examples`: `cdl/surd_discovery` for SURD, and `cdl/brcd_discovery` / `cdl/brcd_boss_discovery` for BRCD on a real Sock Shop case. The API reference lives on docs.rs at `deep_causality_discovery`. The underlying MRMR, SURD, and BRCD primitives are in `deep_causality_algorithms`, usable directly when the full pipeline is more than you need.

Causal State Machine

The Causal State Machine (CSM) is the connector between a Causaloid's verdict and an effect on the outside world. It lives in [deep_causality/src/types/csm_types]() and is built around two ideas: a state is "active" when its Causaloid evaluates to an active effect; an action is a function that runs when its paired state is active.

What it is

A CSM holds a thread-safe map of `(CausalState, CausalAction)` pairs:

```
pub struct CSM<I, O, C>
where
    I: Default + Clone,
    O: CsmEvaluable + Default + Debug + Clone,
    C: Clone,
{
    state_actions: Arc<RwLock<CSMMap<I, O, C>>>,
}
```

The constructor takes a slice of pairs. The map sits behind `Arc<RwLock< ... >>`, so the CSM is shareable across threads and can grow or shrink at runtime through `add_state`, `remove_state`, and `update_state`, each implemented in its own submodule.

States and actions

A `CausalState<I, O, C>` is small. It carries:

- an integer `id`;
- a `version` for tracking iterations of the same logical state;
- a `PropagatingEffect<I>` of pre-bound data;
- the `Causaloid` that decides whether this state is active;
- an optional `UncertainParameter` for states whose Causaloid emits an uncertain effect.

A `CausalAction` is smaller still:

```
pub struct CausalAction {
    action: fn() -> Result<(), ActionError>,
    description: &'static str,
    version: usize,
}
```

`action.fire()` invokes the function pointer. The action surface is kept minimal on purpose: the conditional logic lives in the Causaloid, not in the action.

Evaluation

Two entry points are exposed:

- `eval_single_state(id, data)` runs the Causaloid for one state against caller-supplied data.
- `eval_all_states()` walks every registered state and runs its Causaloid against the data already bound to the state.

In both cases the effect returned by the Causaloid is inspected:

```
let is_active = match &effect.value {
    EffectValue::Value(val) => val
        .is_active(state.uncertain_parameter().as_ref())
        .map_err(CsmError::Causal)?,
    // Other variants (RelayTo, Error, etc.) are inactive for action firing.
    _ => false,
};

if is_active {
    self.fire_action_with_ethos_check(state, action, effect)?;
}
```

`is_active` is delegated to the `CsmEvaluable` trait. A deterministic effect resolves to `true` or `false` directly; an uncertain effect runs its hypothesis test against the state's `UncertainParameter`. Anything that is not a value (relay, error, none) is treated as inactive and fires nothing.

When to reach for it

A CSM fits whenever the question “should this fire?” has to follow a causal verdict rather than a fixed threshold. Common shapes:

- sensor monitoring with thresholds that depend on context, not on the latest reading alone;
- alert routing where the alert depends on a pattern across signals;
- control loops in which an action should run only when an uncertain condition clears a confidence bar.

The CSM does not own a scheduler. The host application decides when to call `eval_*`. That keeps it cheap to embed in async runtimes, batch jobs, or hard real-time loops.

See also

- Example: sensor monitoring with CSM, backed by `examples/csm_examples/csm_basic`.
- Concept: Causaloid, the unit that supplies the verdict each state acts on.
- Concept: Effect Ethos, for policy checks that should gate an action before it fires.
- Concept: Uncertainty, for the hypothesis tests applied when a state’s Causaloid emits an uncertain effect.

Uncertainty

DeepCausality treats uncertainty as a first-class type. Two related types ship in the `deep_causality_uncertain` crate. Both follow the design in Bornholt, Mytkowicz, and McKinley, "Uncertain‹T›: A First-Order Type for Uncertain Data" (ASPLOS '14).

The uncertainty bug

Most engineering code treats a noisy estimate as an exact value. A single `f64` represents a sensor reading whose real distribution might be `Normal(50.0, 2.5)`. Compound a few of these and the final number carries no record of where it came from or how confident it is. Conditionals on such values silently produce false positives and false negatives.

Uncertain<T>

`Uncertain<T>` wraps a value along with the distribution that produced it:

```
use deep_causality_uncertain::Uncertain;

let noisy = Uncertain::<f64>::normal(50.0, 2.5);  // mean 50, std-dev 2.5
let range = Uncertain::<f64>::uniform(0.0, 100.0);
let flip  = Uncertain::<bool>::bernoulli(0.5);
let exact = Uncertain::<f64>::point(10.0);        // a known value, lifted in
```

Arithmetic, comparison, and logical operators are overloaded. They build an implicit computation graph rather than collapsing to a number:

```
let a = Uncertain::<f64>::normal(10.0, 1.0);
let b = Uncertain::<f64>::normal(5.0, 0.5);
let total = a + b;  // still Uncertain<f64>; the distribution is preserved
```

Evaluation is lazy and sampling-based. `expected_value(n)` and `standard_deviation(n)` draw samples; `estimate_probability(n)` runs the chain to a target precision; `to_bool(confidence)` and `probability_exceeds(threshold, confidence, samples)` use the Sequential Probability Ratio Test so they stop sampling as soon as the verdict reaches the requested confidence. A thread-local cache memoizes draws so repeated reads of the same graph node do not redo work.

Conditionals that respect the distribution

Branching on an uncertain boolean returns an uncertain value rather than collapsing it:

```
let traffic_heavy = Uncertain::<bool>::bernoulli(0.7);
let via_main = Uncertain::<f64>::normal(30.0, 5.0);
let via_back = Uncertain::<f64>::normal(45.0, 2.0);

let eta = Uncertain::conditional(traffic_heavy, via_back, via_main);
```

`Uncertain::conditional` is the controlled exit from the uncertainty world; it produces a single uncertain estimate that mixes both branches in proportion to the condition. `implicit_conditional()` is the convenience for "more likely than not" decisions; `to_bool(confidence)` is the explicit form when a specific confidence bar is required.

MaybeUncertain<T>: probabilistic presence

Some values may not exist at all. A sensor misses a frame; a clinical-trial subject does not report on a given day. `MaybeUncertain<T>` separates two questions: is the value present, and if it is, what is its distribution.

```
use deep_causality_uncertain::{MaybeUncertain, Uncertain};

let always_known   = MaybeUncertain::
<f64>::from_uncertain(Uncertain::normal(10.0, 2.0));
let always_missing = MaybeUncertain::<f64>::always_none();
let sometimes      = MaybeUncertain::<f64>::from_bernoulli_and_uncertain(
    0.7,
    Uncertain::normal(5.0, 1.0),
);
```

Arithmetic propagates absence. Adding `sometimes + always_missing` returns absence with the probability the operand had of being missing. `is_some()` returns an `Uncertain<bool>` the rest of the framework can reason about. `lift_to_uncertain(presence_threshold, confidence, ...)` collapses the type back into a plain `Uncertain<T>` once there is enough evidence the value is actually there.

Where it shows up in the framework

A `CausalState` carries an optional `UncertainParameter`, so a Causaloid that emits an uncertain effect can be tested against state-specific confidence and sample-budget settings at fire time. The CDL pipeline can feed uncertain features into discovery without flattening them to point estimates. The Effect Ethos can gate actions on uncertain conditions with explicit confidence bars rather than thresholds on means.

See also

- Crate README: `deep_causality_uncertain`.
- Examples: `gps`, `sensor`, and `clinical_trial` cover route choice under noise, sensor fusion with anomaly detection, and trial data with probabilistic presence.
- Concept: Causal State Machine, which uses `UncertainParameter` to gate action firing.
- Background: Bornholt, J., Mytkowicz, T., McKinley, K. S. “Uncertain⟨T⟩: A First-Order Type for Uncertain Data.” ASPLOS ‘14.

Uniform Math

DeepCausality treats algebra, geometry, topology, and effect propagation as a single mathematical surface. The same `Functor`, `Monad`, and `CoMonad` operations run over tensors, multivectors, manifolds, sparse matrices, and propagating effects. Two crates do the work: `deep_causality_haft` provides the Higher-Kinded Type machinery; `deep_causality_num` provides the algebraic trait floor that the math containers stand on.

Why Uniform Mathematics matters

Scientific code typically pays a hidden tax: every time the math crosses a domain (a mesh walk to a tensor contraction, a tensor to a rotor, a rotor back to a scalar field), the developer writes bridge code. Indices get repacked. Loops get rewritten. The contraction lives in one library, the rotation in another, the per-vertex traversal in a third. Each crossing is a place where bugs hide and where one library's conventions clash with another's.

A unified mathematical surface removes the tax. When a tensor, a multivector, and a manifold all implement the same `Functor` and `Monad` operations, they compose in the same way they would on paper. A walk over a mesh, a contraction on a per-vertex tensor, a rotation by a Clifford rotor, and an audit-logged monadic step are five operations from the same algebraic vocabulary, not five separate libraries that need translating.

The practical consequences are real:

- **One composition law applies everywhere.** `fmap`, `bind`, `extend`, and `extract` mean the same thing on a tensor, a multivector, a manifold, or a propagating effect.
- **Cross-domain pipelines stay readable.** Mesh walk and tensor algebra and rotor application can share a single closure. The structure of the code matches the structure of the math.
- **Numerical precision becomes a parameter, not an assumption.** Because the algebraic floor is generic, every container honors the same `RealField` / `Field` / `Ring` traits, and the float type can be changed in one place.
- **Algebraic laws are compile-time guarantees.** A type that is not associative cannot be passed where associativity is required. The compiler enforces what mathematicians prove by hand.

A concrete example: GRMHD

The grmhd example (General Relativistic Magnetohydrodynamics) is the sharpest demonstration of what the unification enables. It couples a general-relativity solver to a plasma physics solver, picks a metric signature dynamically based on local spacetime curvature, computes the Lorentz force in the selected geometry, and feeds the electromagnetic stress-energy back into the spacetime metric. Every one of those steps lives in a different mathematical regime. They are composed by a `bind` chain over the Causal Monad:

```
let result: PropagatingEffect<GrmhdState> =
PropagatingEffect::pure(GrmhdState::new(&config))
    .bind(|state, _, _| {
        // [Step 1] GR Solver — tensor algebra.
        // Builds the Schwarzschild metric g_uv and Ricci tensor,
        // contracts them into the Einstein tensor G_uv = R_uv - ½ R g_uv.
        model::calculate_curvature(state.into_value().unwrap_or_default())
    })
    .bind(|state, _, _| {
        // [Step 2] Causal coupling — metric-signature selection.
        // Branches on curvature intensity and picks Metric::Minkowski(4)
        // (relativistic regime) or Metric::Euclidean(3) (classical regime).
        model::select_metric(state.into_value().unwrap_or_default())
    })
    .bind(|state, _, _| {
        // [Step 3] MHD Solver — Clifford Algebra.
        // Wraps the current J and magnetic field B as CausalMultiVector<f64>
        // in the metric chosen above, then computes F = J ∧ B as a bivector
        // through the Clifford geometric product.
        model::calculate_lorentz_force(state.into_value().unwrap_or_default())
    })
    .bind(|state, _, _| {
        // [Step 4] GRMHD coupling — back to tensor algebra.
        // Builds the EM field strength tensor F^uv (rank-2 CausalTensor)
        // and contracts it with the spacetime metric g_uv from Step 1
        // to produce the EM stress-energy tensor T^uv.

model::calculate_energy_momentum(state.into_value().unwrap_or_default())
    })
    .bind(|state, _, _| {
        // [Step 5] Stability analysis
        // Scalar branching on the bivector intensity.
        model::analyze_stability(state.into_value().unwrap_or_default())
    });
```

Look at what the chain crosses. Step 1 is pure tensor algebra in `deep_causality_tensor`. Step 2 makes a runtime decision in `deep_causality_metric` that changes the geometry of the next step. Step 3 leaves tensor algebra entirely and computes in Clifford / geometric algebra through `deep_causality_multivector`. Step 4 returns to tensor algebra, but now coupled to the same spacetime metric produced in Step 1, which is what closes the GRMHD feedback loop. Step 5 is ordinary Rust.

With the uniform surface, `CausalTensor` and `CausalMultiVector` and `Metric` are all Functor / Monad instances over the same `PropagatingEffect` carrier. The Causal Monad's `bind` is the only composition operator. Each stage consumes a `GrmhdState` and returns an updated `GrmhdState` wrapped in a `PropagatingEffect`, regardless of which mathematical regime it works in. The pipeline reads like the physics: curvature → metric selection → Lorentz force → stress-energy → stability.

This is what "uniform mathematical foundation" means in practice. It is the ability to write a five-stage GR-plus-plasma simulation as a five-line monadic chain, with the type system enforcing dimensional consistency and the algebraic floor guaranteeing that every step uses the same scalar field at the same precision.

The `mathematics_examples` tree extends the same composition further: a Kalman predict-correct chain mixing tensor and rotor steps, a heat equation alternating `extend` for the spatial Laplacian with `bind` for the time step, and the `capstone_spinor_minkowski` example parallel-transporting a unit timelike spinor through `Cl(3,1)` along a discretized worldline. Same composition law in every case.

The uniform mathematical foundation is enabled by two distinct capabilities on the Deep Causality Project: Higher kinder types and an algebraic trait hierarchy.

HKT in Rust via the witness pattern

Rust has no native Higher-Kinded Types. HAFT adds the abstraction with a witness pattern: a zero-sized struct that implements the `HKT` trait and stands in for the type constructor. Code generic over the witness picks up any container that implements the same functional trait:

```rust
fn double_value<F>(m_a: F::Type<i32>) -> F::Type<i32>
where
    F: Functor<F> + HKT,
{
    F::fmap(m_a, |x| x * 2)
}
```

`double_value::<OptionWitness>(Some(5))`, `double_value::<VecWitness>(vec![1, 2, 3])`, and `double_value::<ResultWitness<i32>>(Ok(5))` all type-check, all run, and none allocate for the witness itself. Default witness implementations ship for `Option`, `Result`, `Box`, `Vec` (full `Functor/Applicative/Monad/Foldable`) and for `BTreeMap`, `HashMap`, `VecDeque` (`Functor` and `Foldable` only).

Higher-arity traits (`HKT2` through `HKT5`) handle type constructors with more than one parameter. Unbound variants exist for `Bifunctor`, `Profunctor`, `Adjunction`, `ParametricMonad`, `Promonad`, `RiemannMap`, and `CyberneticLoop`. Each one is the right tool for a specific shape of dynamics; the HAFT README documents the intended use case for each.

A type-encoded effect system sits on top of the multi-arity HKTs. `Effect3/Effect4/Effect5` plus `MonadEffect3/MonadEffect4/MonadEffect5` let an effect type carry a primary value plus several fixed channels (error, log, trace) through `pure` and `bind`. The compiler checks that effects are handled or propagated; nothing leaks implicitly.

The same pattern lifts the math containers into the same surface:

Domain	Type	Witness	Role
Mechanics	`CausalTensor<T>`	`CausalTensorWitness`	Data container
Algebra	`CausalMultiVector<T>`	`CausalMultiVectorWitness`	Transformations
Topology	`Manifold<T>`	`ManifoldWitness`	Space and context
Structure	`CsrMatrix<T>`	`CsrMatrixWitness`	Sparse relations
Causality	`PropagatingEffect<T>`	`Effect5Witness`	Time and flow

Each one implements `Functor` and `Monad` through HAFT. `Manifold` additionally implements `CoMonad`, which is what enables `extend` (apply a function to every local neighborhood) and

`extract` (read the value at the current point). Code written against `fmap`, `bind`, `extend`, and `extract` runs over any of these without rewriting the call site.

Adjunctions (`BoundedAdjunction`) formally link Geometry (Multivectors) and Topology (Manifolds), so a problem stated in one category can be translated to the other along a typed bridge rather than ad-hoc glue.

The algebraic trait hierarchy

The numeric layer underneath ships an explicit algebraic hierarchy in [deep_causality_num]. The trait names follow standard abstract algebra:

```
Magma → Semigroup → Monoid → Group → AbelianGroup
                                        ↓
                                      Ring → CommutativeRing → Field →
RealField → ComplexField<R>
```

with `Module<R>`, `Algebra<R>`, `AssociativeAlgebra<R>`, `DivisionAlgebra<R>`, and `EuclideanDomain` for vector and ring-with-division structures. Marker traits (`Associative`, `Commutative`, `Distributive`) make algebraic laws compile-time promises rather than convention. Concrete classifications:

Type	Primary traits
`f32`, `f64`	`RealField`, `Field`, `DivisionAlgebra<Self>`
`Complex<T>`	`Field`, `DivisionAlgebra<T>`, `Rotation<T>`
`Quaternion<T>`	`AssociativeRing`, `DivisionAlgebra<T>`, `Rotation<T>`
`Octonion<T>`	`DivisionAlgebra<T>` (non-associative)
`i8…i128`, `isize`	`Ring`, `Integer`, `SignedInt`
`u8…u128`, `usize`	`Ring`, `Integer`, `UnsignedInt`

The library will not let an algorithm that requires associativity use a non-associative type. The type system rejects it before the program runs.

The math layers

The witness table earlier in this page lists each math container in one row. This section gives each one a bit more shape: what is in the box, and when to reach for it.

Tensors: `deep_causality_tensor`

N-dimensional arrays with stride-based memory layout, broadcasting for element-wise operations, and **Einstein summation** for matrix products and tensor contractions. The crate ships `Functor`, `Applicative`, `Monad`, and `CoMonad` instances, so a contraction or a per-cell map composes through the same `fmap`/`bind` surface the rest of the stack uses. Reach for tensors wherever the math is rectangular data: relativistic field tensors, Kalman state and covariance, linear algebra mid-pipeline.

Multivectors and Geometric Algebra: `deep_causality_multivector`

Clifford algebras over the dynamic signature space, with pre-configured constructors for **Pauli (Cl(3,0))**, **Spacetime Algebra (STA)**, **Conformal Geometric Algebra (CGA)**, **Projective Geometric Algebra in 3D (PGA3D)**, the **Dixon algebra** used in Standard Model particle physics, and the **Grand Unified Algebra hosting the full Spin(10) gauge symmetry**. `Functor`, `Applicative`, and `Monad` instances are implemented; `bind` realizes the **tensor product of algebras**, so dimension-changing compositions are monadic rather than ad-hoc.

The payoff is concrete. The Maxwell example expresses the electromagnetic field `F = ∇A` as a single multivector and recovers `E` and `B` as its bivector grades. Six scalar components collapse to four, a ~50% compute reduction, without giving up correctness. Where vector calculus needs separate cross products, exterior derivatives, and Hodge stars to express the same physics, the geometric product handles it as one operation on one object.

Topology and Differential Geometry: `deep_causality_topology`

Graphs, hypergraphs, **simplicial complexes**, **manifolds**, and point clouds, with first-class **exterior calculus operators** (exterior derivative `d`, Hodge star `⋆`, codifferential `δ`, Hodge-Laplacian) and a **lattice gauge field framework** supporting **U(1), SU(2), SU(3), and Lorentz** gauge groups. The lattice gauge implementation is verified against **24 reference results from**

Creutz's *Quarks, Gluons and Lattices*, a stable bedrock for anyone composing physical simulations from this stack.

`Manifold` is the layer's `CoMonad`; that is what makes `extend` (apply a function to every local neighborhood) and `extract` (read the value at the current point) first-class on geometric data. Graph convolutions and cellular automata become one comonadic walk over a typed neighborhood.

Sparse matrices: `deep_causality_sparse`

CSR (Compressed Sparse Row) sparse matrices with `Functor`, `Applicative`, and `Monad` instances, used wherever the data is large and mostly zero: causal-graph adjacency, big covariance structures, transition matrices. The same `fmap/bind` surface applies.

Metric signatures: `deep_causality_metric`

A single horizontal crate that defines metric signatures **once** and shares them across every layer above: the **East Coast convention** used in general relativity, the **West Coast convention** used in particle physics, and the broader Clifford signature space **Cl(p, q, r)** that the multivector and topology crates build on. A GR calculation in the tensor layer and a particle-physics calculation in the multivector layer can share sign conventions automatically;

Precision as a parameter

The algebraic trait floor and the witness-based composition together produce a property that is hard to get any other way: numerical precision becomes a single line of the program. Every example in [examples/mathematics_examples] exposes one type alias near the top of `main.rs`:

```
pub type FloatType = Float106;  // or f64, or f32
```

That alias flows through every tensor contraction, multivector rotation, manifold extension, and monadic step. Edit the line; the program re-runs at the new precision. There is no parallel implementation, no `#ifdef`, no second copy of the math at a different precision. The compiler instantiates the generic code at whichever scalar field the alias resolves to.

The capstone (`capstone_spinor_minkowski`) parallel-transports a unit timelike spinor along a discretized Minkowski worldline through four boost steps, then compares the composed result against `cosh(θ), sinh(θ)` for the summed rapidity:

Precision	Composition drift
`f64`	~1.1×10^{-16}
`Float106`	~1.7×10^{-31}

Fifteen orders of magnitude, recovered by editing one line. The algorithm, topology, Clifford rotor, and monadic chain are unchanged. Chained transcendental composition (Lie-group accumulation, long Kalman cascades, repeated rotor application, parallel transport) is where the precision pays off the most.

What the unification actually enables

The HKT machinery, the witness table, and the algebraic trait floor are three layers of one architecture. Read in isolation, each one looks like an implementation detail. Together they give the library a property that conventional scientific stacks struggle to deliver:

A single closure can walk a mesh, contract a tensor, apply a Clifford rotor, accumulate a state, append to an audit log, and short-circuit on error, in any order, with no glue between the layers. The composition law that makes the closure work is the same law that composes a stateless `bind` chain, a contextual Causaloid graph, a manifold extension, and a `PropagatingEffect` returned from the Effect Propagation Process. The math, the data structures, and the runtime all speak one language.

See also

- Reference READMEs: `deep_causality_haft`, `deep_causality_num`.
- Examples: `examples/mathematics_examples` covers HKT composition (`tensor_x_algebra_rotation_field`, `tensor_x_topology_laplacian`, `triple_hkt_stress_field`), causal-monad composition (`effect_kalman_predict_correct`, `effect_diffusion_on_manifold`, `effect_tensor_algebra_roundtrip`), and the `Cl(3,1)` spinor capstone.
- Concept: Higher-Kinded Types, Causal Monad, Effect Propagation Process.

Counterfactuals

Counterfactual reasoning is first-class in DeepCausality. The same machinery that runs factual evaluation runs counterfactual evaluation. The mechanism is the `Alternatable` family of traits: one trait per substitutable channel on the carrier, plus a marker super-trait that bundles all three. The classic causal-inference operator `intervene` survives as a thin vocabulary alias atop value alternation.

Pearl's Ladder of Causation

Pearl distinguishes three rungs of causal reasoning, each strictly stronger than the one below:

Rung	Question	Operator	EPP expression
1. Association	"If I see X, what do I expect about Y?"	`P(Y | X)`	`pure(x).bind(f)`; read-only composition
2. Intervention	"If I *do* X, what happens to Y?"	`P(Y | do(X))`	`pure(x).bind(f).intervene(new)`; overrides a value mid-chain
3. Counterfactual	"Given the world as it is, what *would* have happened if X had been different?"	`P(Y_x | X', Y')`	the same chain run twice, factually and with an alternation, then compared

The first rung is a `bind`. The second adds value alternation (`intervene` / `alternate_value`). The third rung runs rung two against a held factual reference and compares the two outcomes. The architecture is the same in every case: a chain whose value, state, context, error, and log are the only thing being threaded.

The Alternatable family: three channels, three traits

The carrier struct (`CausalEffectPropagationProcess`) has five fields: `value`, `state`, `context`, `error`, `logs`. Three are legitimately substitutable mid-chain; two are not. Substituting `error` would silently paper over a failure upstream, and `logs` is append-only by design. That leaves three alternable channels.

DeepCausality exposes one single-method trait per channel:

```
pub trait AlternatableValue<V>   { fn alternate_value(self,   new: V) →
Self; }
pub trait AlternatableContext<C> { fn alternate_context(self, new: C) → Self;
}
pub trait AlternatableState<S>   { fn alternate_state(self,   new: S) → Self;
}
```

Plus a marker super-trait, auto-implemented via blanket impl, for generic code that needs all three at once:

```
pub trait Alternatable<V, C, S>:
    AlternatableValue<V> + AlternatableContext<C> + AlternatableState<S> {}

impl<T, V, C, S> Alternatable<V, C, S> for T
where T: AlternatableValue<V> + AlternatableContext<C> + AlternatableState<S>
{}
```

The familiar `Intervenable<V>` trait survives as a thin vocabulary alias atop `AlternatableValue<V>`, so existing code that calls `effect.intervene(x)` keeps working unchanged:

```
pub trait Intervenable<V>: AlternatableValue<V> {
    fn intervene(self, new_value: V) → Self where Self: Sized {
        self.alternate_value(new_value)
    }
}
impl<T, V> Intervenable<V> for T where T: AlternatableValue<V> {}
```

`intervene` is the causal-inference word for value substitution (Pearl's `do(...)`). `alternate_value` is the same operation under the EPP's substitution vocabulary. Same method body, two surfaces.

The shared contract

Every method in the family follows the same three rules:

- **Error short-circuit.** If the upstream chain has already errored, the call is a no-op. An alternation cannot fix a broken chain.
- **Channel isolation.** Only the named channel is rewritten; the other two alternable channels, plus the error and log, continue unchanged.
- **Automatic audit entry.** A distinctive marker is appended to the log: `!!ValueAlternation!!: <old> replaced with <new>`, `!!ContextAlternation!!: context replaced`, or `!!StateAlternation!!: state replaced`.

The three operators compose freely. On a `PropagatingProcess` you can intervene on the value, alternate the context, and reset the state in any order within the same chain; each emits its own audit entry, and downstream `bind` steps see the substituted channels.

Which channels carry real information

The trait family is implemented uniformly on the underlying struct, so both pinned aliases satisfy it. The channels that carry information depend on which alias you use:

- **`PropagatingEffect<T>`** pins `State` and `Context` to `()`. Only value alternation does observable work on this alias; `alternate_context(())` and `alternate_state(())` are well-typed but only append a log entry.
- **`PropagatingProcess<T, S, C>`** keeps state and context generic. All three operators are meaningful.

Use `PropagatingEffect` for Pearl-style stateless chains. Use `PropagatingProcess` whenever a chain needs to thread Markovian state or a typed world context.

Walking the Ladder in code

The intervention example from the project README walks all three rungs in roughly a dozen lines on a stateless `PropagatingEffect`:

```rust
use deep_causality_core::{Intervenable, PropagatingEffect};

// Causal chain: Dose → Absorption → Metabolism → Response (numeric
outcome).

// Rung 1: Association. Run the chain factually.
let observed = PropagatingEffect::pure(10.0_f64)
    .bind(|dose,  _, _|
PropagatingEffect::pure(dose.into_value().unwrap_or_default() * 0.8))   //
Absorption: 8.0
    .bind(|level, _, _|
PropagatingEffect::pure(level.into_value().unwrap_or_default() - 2.0))  //
Metabolism: 6.0
    .bind(|level, _, _|
PropagatingEffect::pure(level.into_value().unwrap_or_default()));       //
Response:   6.0

// Rung 2: Intervention. do(BloodLevel := 3.0) mid-chain.
let intervened = PropagatingEffect::pure(10.0_f64)
    .bind(|dose, _, _|
PropagatingEffect::pure(dose.into_value().unwrap_or_default() * 0.8))   //
Absorption: 8.0
    .intervene(3.0)
// do(BloodLevel := 3.0)
    .bind(|level, _, _|
PropagatingEffect::pure(level.into_value().unwrap_or_default() - 2.0)) //
Metabolism: 1.0
    .bind(|level, _, _|
PropagatingEffect::pure(level.into_value().unwrap_or_default()));      //
Response:   1.0

// Rung 3: Counterfactual. The causal-effect estimate is the difference between
// the intervened outcome and the observed outcome (individual treatment
effect):
//     ITE = Y(do(X)) - Y(X_observed)
let y_obs = observed.value.into_value().unwrap_or_default();
let y_int = intervened.value.into_value().unwrap_or_default();
let causal_effect = y_int - y_obs;

println!("Observed Y      = {y_obs:.2}");
println!("Intervened Y    = {y_int:.2}");
```

```
println!("Causal effect Δ = {causal_effect:+.2}"); // -5.00: the intervention lowered the response by five units.
```

The two runs share their structure and their composition law. The only difference is the `.intervene(3.0)` call. The causal-effect estimate is the **difference** `Y(do(X)) - Y(X_observed)`; in this run that is `1.00 - 6.00 = -5.00`. The log on `intervened` records the original blood level, the substituted value, and the marker that an intervention occurred; the run stays replayable and auditable.

Beyond the value channel

Pearl's `do(...)` operator only swaps **one** thing: the value of a single endogenous variable, with the exogenous noise held fixed by the abduced posterior. The `Alternatable` family widens the substitution surface to three independent channels and removes the inference step entirely:

- **Value alternation** (`alternate_value` / `intervene`): Pearl's `do(...)`, expressed as one method call.
- **Context alternation** (`alternate_context`): swap the entire world (or any structured piece of it) without rebuilding the chain. The classical SCM example uses this for Pearl's rung 3; the RCM example uses it to compute potential outcomes across treatment and control.
- **State alternation** (`alternate_state`): force the running Markov state to a new value. Useful for simulator resets, regime changes that touch accumulated counters, and test fixtures. The DBN example shows context alternation alongside state threading: a mid-stream climate-regime change leaves the day counter and umbrella count intact.

Abduction never enters this picture, because `PropagatingProcess` already carries the world state explicitly. The trade is concrete: the world state must be represented as a Context up front, instead of inferred from observations.

Why mid-chain matters

Most counterfactual frameworks require structural manipulation (mutilating the SCM, rebuilding the graph, re-evaluating from the root). That works at the model level but is expensive and obscures the trace. The `Alternatable` family operates at the **value, context, or state level** along the existing chain. The causal law, the Causaloid graph, and the audit log are unchanged; the rewritten channel flows through the remaining steps as if it had been produced upstream.

For interactive what-if analysis, sensitivity testing, and Pearl-style do-calculus over a running pipeline, that is the cheap and honest operation to have.

These traits are not a substitute for structural intervention when the question genuinely changes the model (deleting an edge, removing a Causaloid). Those are graph-level edits, and the EPP expresses them by composing a different Causaloid graph against the same Context. The `Alternatable` family is the channel-level rung; structural surgery is its model-level counterpart.

What this means

- **Counterfactuals as a one-line API.** A counterfactual is one alternation call between `start(ctx)` and the binds.
- **Three independent substitution channels.** Value, context, and state can be alternated independently or in any combination, each emitting its own distinctive audit marker.
- **Replayable counterfactual analysis.** Every alternation is recorded in the same log as the factual run, so a downstream consumer can reproduce both.
- **No abduction step.** The world state lives in the Context explicitly, so "the world as it actually was" is a value, not an inferred posterior over hidden noise.
- **Composable with the rest of the algebra.** A counterfactual chain returns a `PropagatingEffect` or `PropagatingProcess`, so it composes with Causaloid evaluations, downstream `bind` steps, and the Effect Ethos check just like any factual chain would.

See also

- Causal Monad: the `pure/bind` algebra that the `Alternatable` family plugs into.
- Effect Propagation Process: the carrier whose channels the family rewrites.
- Causaloid: for structural (graph-level) counterfactuals.
- Classical Causality Examples: five textbook methods (CATE, DBN, Granger, RCM, SCM) implemented twice, once with the Causaloid + Contextual Alternation pattern and once with `PropagatingProcess` + the `Alternatable` family. Includes a practitioner decision guide.
- `examples/starter_example`: walks Pearl's Ladder end to end on `PropagatingEffect`.

Causal Discovery Algorithms

The `deep_causality_algorithms` crate ships the two algorithms the Causal Discovery Language wires together to turn raw observational data into a discovery report that informs the construction of a causal model. They are independent and individually useful; they cover complementary halves of the discovery problem: **which variables matter (MRMR)** and **how they interact (SURD)**.

MRMR: Maximum Relevance, Minimum Redundancy

MRMR is a filter-based feature-selection method from Zhao, Anand, and Wang (Uber, 2019), originally developed for industrial-scale machine-learning pipelines but well suited to causal discovery as a pre-step. Given a target variable and many candidate features, MRMR ranks features by two competing pressures:

- **Maximum relevance**: pick features that are individually informative about the target.
- **Minimum redundancy**: penalize features that mostly repeat what already-selected features tell you.

The result is a compact, informative feature subset that retains most of the signal a richer set carries, at a fraction of the discovery cost. As a filter method, relative to wrappers and embedded methods, MRMR is fast, model-agnostic, and reusable across pipelines.

When to reach for it: the data has more candidate features than the discovery step can reasonably search, and you want the discovered model to be both tractable and interpretable. MRMR is the CDL pipeline's standard pre-discovery stage.

Reference paper: arXiv:1908.05376, *Maximum Relevance and Minimum Redundancy Feature Selection Methods for a Marketing Machine Learning Platform*.

SURD: State-and-interaction-type causality

SURD (Synergistic / Unique / Redundant Decomposition) is the discovery method from Martínez-Sánchez and Lozano-Durán. It quantifies causality as a function of **system state and interaction type**, decomposing the influence one variable has on another into three components:

- **Synergistic**: the joint contribution of multiple sources that only appears when they act together.
- **Unique**: the part each source contributes that no other source can replace.
- **Redundant**: the part that multiple sources carry equivalently.

Traditional methods (Convergent Cross Mapping, PCMCI, Granger causality, conditional independence tests) report **a single average causal strength** across all states of the system. SURD breaks that single number into the three components and tracks them *per state*; two variables that look similar on average but differ in their synergistic content are no longer mistaken for the same relation. The state-conditional view is the natural match for the EPP's dynamic-causality model: causal structure that varies with the state of the system is exactly what the framework is built to represent.

When to reach for it: the system is dynamic, the relationships are likely non-linear or state-dependent, and you want to distinguish redundant signals from genuinely additive (synergistic) ones rather than averaging them away.

Reference paper: arXiv:2505.10878, *Observational causality by states and interaction type for scientific discovery*.

How they compose in the CDL pipeline

The Causal Discovery Language is the typestate-builder pipeline that orchestrates the two: data load → clean → **MRMR feature selection** → **SURD causal discovery** → analysis → finalize. Each stage advances the typestate so misuse is rejected at compile time. The output is a `CdlReport` whose recommendations (for example, "Strong unique influence: Recommended Direct edge in `CausaloidGraph`") tell you which Causaloids to wire and how; you then construct the `CausaloidGraph` from those findings using the rest of the framework.

Either algorithm is usable on its own. MRMR is independently useful as a general feature-selection primitive. SURD is independently useful wherever you want a state-conditional decomposition of causal interactions rather than a scalar score.

See also

- Causal Discovery Language: the typestate pipeline that wires both algorithms.
- Causaloid: the output of the discovery step, the unit the rest of the framework composes over.

- Uncertainty: the `Uncertain<T>` type for downstream propagation under noise.
- Reference: `deep_causality_algorithms` crate.

Deployment

A DeepCausality model may run in a normal binary, on a background thread, or across a pool of Tokio worker threads.

This page works through the deployment story using the `tokio_example` crate.

A runtime-agnostic core

A Causaloid is evaluated through `evaluate(&self, ...)`. The call borrows the Causaloid; it does not consume or mutate it. The rule itself is a function pointer, so evaluation allocates nothing on the heap and dispatches nothing through a vtable. One value goes in, one `PropagatingEffect` comes out.

Two consequences follow. A single Causaloid can be evaluated any number of times. And because evaluation is a shared read, it can be evaluated from many places at once, which makes it thread-safe.

Asynchronous serving with Tokio

For a service that ingests a stream of events, pair the model with Tokio and run inference on a task. The example's entry point is just twelve lines:

```
#[tokio::main]
async fn main() -> Result<(), Box<dyn std::error::Error>> {
    let event_handler = EventHandler::new(build_causal_model());

    tokio::spawn(async move {
        if let Err(e) = event_handler.run_background_inference().await {
            eprintln!("inference error: {e}");
        }
    })
    .await
    .expect("Failed to spawn async background task");

    Ok(())
}
```

The model is built once, the handler takes ownership, and inference runs on a spawned task. Here the main task awaits that task, so the program runs to completion and exits cleanly. In a real service the task is long-lived: it reads events off a channel, the main task is free to do other work, and the `.await` becomes a graceful-shutdown handle.

The inference inside the task is synchronous. There is no `.await` in the evaluation path, no async Causaloid, no futures threaded through the causal logic. The asynchrony sits at the runtime boundary. This keeps the causal code simple and keeps the hot path free of executor overhead.

Multi-threaded serving

The model is shareable across threads, and the lock that makes it shareable does not serialize the work. The handler holds the model behind an `Arc<RwLock< ... >>`, and the Causaloid inside it is itself an `Arc`:

```
pub struct EventHandler {
    model: Arc<RwLock<BaseModelTokio>>,
}

pub async fn run_background_inference(&self) → Result<(), Box<dyn Error +
Send>> {
    let causaloid = {
        let model = self.model.read().unwrap();
        Arc::clone(model.causaloid())
        // read guard dropped here
    };

    for d in data.into_iter() {
        self.handle_inference(d, &causaloid)?;
    }
    Ok(())
}
```

The pattern is deliberate. A worker takes the read lock, clones the inner `Arc<Causaloid>`, and drops the guard immediately. The lock is held for the few microseconds it takes to copy a pointer. After that, the worker evaluates against its own `Arc` clone with no lock at all.

That is what allows concurrent serving. An `RwLock` admits many readers at once, so any number of Tokio worker threads can clone the pointer in parallel; none of them blocks on inference

because none of them holds the lock while inferring. The same model answers one request or a million, and the Causaloid's `evaluate(&self)` signature guarantees the shared access is safe. A writer takes the lock only when the model or its Context must be updated, and only then do readers wait.

Run the example

```
git clone https://github.com/deepcausality-rs/deep_causality
cd deep_causality
cargo run --release -p tokio_example
```

The program builds the handler, starts the background task, and prints one line per inference before exiting cleanly.

Glossary

This page is the single source of truth for terminology. The other concept pages link here. The EPP preprint and its companion volumes are the formal authority; this page is the operational synopsis.

Core terms

Causaloid: A self-contained unit of causality. Wraps a causal function and metadata. Composes recursively into Singleton, Collection, and Graph forms that share the same type. See Causaloid.

Causaloid Graph: A directed graph whose nodes are Causaloids and whose edges express the order of evaluation. The result of evaluating the graph is the effect produced at its terminal node(s).

Causal Discovery Language (CDL): A typestate-builder pipeline that ingests observational data and produces a discovery report whose recommendations inform the construction of a `CausaloidGraph`. Lives in `deep_causality_discovery`. See CDL.

Causal Function: The pure function a Causaloid wraps. Two signatures: stateless (`CausalFn<I, O>`) takes the input alone; contextual (`ContextualCausalFn<I, O, STATE, CTX>`) also receives a Context.

Causal Monad: The `pure`/`bind` algebra that composes Causal Effect Propagation Processes. A trait (`CausalMonad`) implemented by the carrier effect, not a separate type. The axiom $m_2 = m_1 \gg\!= f$ is its state-threading `bind` operation. See Causal Monad.

Causal Reasoning: The act of running one or more Causaloids against a Context and consuming the resulting propagating effect.

Causal State Machine (CSM): A higher-level construct that links recognized causal states to deterministic actions. Used in the Effect Ethos for separating inference from action. Introduced in the Teleology preprint.

Context: An explicit hypergraph encoding the environment in which Causaloids operate. Nodes are Contextoids; edges are typed relations. See Context.

Contextoid: The atomic node of a Context. Carries a typed payload: `Datoid`, `Spaceoid`, `Tempoid`, `SpaceTempoid`, or `Symboid`. Non-recursive by design.

Contextual Fabric: The monograph's name for the union of Contexts and Context Hypergraphs in which effects propagate. The Rust code exposes this through `BaseContext` and its generic relatives.

Dynamic Causality: The library's framing of causality as a process whose structure (or context, or both) can evolve. The default operating mode. See Dynamic Causality.

Effect Ethos: The verification layer that checks every causal effect against a set of named Teloids. Implements defeasible deontic inference with Lex Posterior, Lex Specialis, and Lex Superior. Lives in `deep_causality_ethos`. See Effect Ethos.

Effect Log: An append-only audit log carried by every `CausalEffectPropagationProcess`. Every Causaloid invocation contributes one entry; `bind` merges logs across the chain.

Effect Propagation Process (EPP): Both a concept and a literal type in code. The concept: the directed flow of effects through a Causaloid chain. The type: `CausalEffectPropagationProcess<Value, State, Context, Error, Log>` in `deep_causality_core`. See Effect Propagation Process.

Effect Value: The enum that holds the payload of a propagating effect: `None`, `Value(T)`, `ContextualLink`, `RelayTo`, or `Map`.

Evidence: A unit of factual data in the monograph's ontology. In code, evidence enters the system as Contextoids of type `Datoid`.

Higher-Kinded Types (HKT): Type-level functions that take types as arguments and return types. The library encodes them via the witness pattern (`HKT3`, `HKT5`) defined in `deep_causality_haft`. See HKT.

Propagating Effect: The stateless carrier alias `PropagatingEffect<T> = CausalEffectPropagationProcess<T, (), (), CausalityError, EffectLog>`. The everyday return type of a Causaloid's function; it implements the Causal Monad trait. Its stateful sibling is `PropagatingProcess<T, S, C>`.

Teloid: The atomic deontic rule inside an Effect Ethos. Encodes a modality (obligatory, impermissible, optional), a condition, and a Context query. Defined in the Teleology preprint.