COHERENCE REALITY ENGINE

Technical Manual

Invaris AI | Version 1.0 | March 9, 2026 | CRE v8.7.1 | Spec: CRE_CANON_0C_WO020

> *For engineers, enterprise integrators, technical due diligence reviewers, and patent examiners. This document covers the complete system architecture from kernel to application layer - every crate, every invariant, every enforcement mechanism, and the rationale for each architectural decision.*

Read this document with skepticism. Every claim about determinism, enforcement, and invariant coverage is backed by a specific test, a specific CI gate, or a specific compiler-level mechanism. Where the claim is architectural rather than test-proven, that distinction is noted.

SECTION 1 Architectural Overview

The CRE Structural Premise

Large language models are stochastic sorters. They predict the most statistically likely next token. That is the entirety of what they do. The result is a system that cannot structurally distinguish between a verified fact, a moral judgment, a probabilistic prediction, and a logical necessity. All four are processed identically as token sequences in a shared embedding space and blended into a single continuous narrative. This is not a flaw to be patched. It is an architectural property of the design.

CRE addresses five structural failure modes that are permanent features of probabilistic generative systems and cannot be resolved from inside them:

- Hallucination - the generation of structurally plausible but factually or logically inadmissible content with no internal signal distinguishing it from valid output
- Drift - the gradual erosion of premises, term definitions, or positional anchors under conversational momentum without evidence-driven cause
- Mirroring - the convergence of model output toward user-expressed preferences, adopting the user's framing as a substitute for structural evaluation
- Authority laundering - the assertion of conclusions requiring explicit authority grounding without declaring or holding to any authoritative framework
- Epistemic category collapse - the inability to distinguish and keep separate the four fundamental epistemic categories: empirical facts, normative judgments, probabilistic guesses, and logical necessities

None of these failure modes can be resolved from inside the probabilistic model. Scaling increases fluency. It does not close a structural absence. RLHF adjusts output tendencies probabilistically. It does not install an epistemic type system. Constitutional AI operates within the embedding space. It does not produce structurally external, deterministic, independently verifiable evaluation artifacts.

> *The patent thesis: epistemic authorization cannot be fixed from inside a probabilistic model. It must be built outside, as a structurally independent deterministic adjudication layer. That is the architectural claim at the core of USPTO Provisional Patent Applications #63/977,457 and #63/988,849.*

The Six-Layer Stack

CRE is a six-layer architecture organized around a single inviolable principle: the system that generates AI output and the system that adjudicates that output must be structurally independent. Independent at the type level, at the dependency level, and at the enforcement level - not as a policy, but as a physical property of the build.

Layer	Name	Description
1	cre-core (Kernel)	Deterministic adjudication engine. Pure function. No side effects. Frozen at commit hash. 434+ passing tests. CI zero-diff gate enforced.
2	cre-api (Membrane)	Schema validation and lane classification boundary. Translates untrusted LLM proposals into typed kernel inputs. All LLM output treated as untrusted proposals.
3	cre-satellite-runtime	Singular invocation chokepoint. Reads kernel PublicOutputFrame. Dispatches to enabled satellite modules by subscription tier. No backdoors.
4	Satellite Crates	cre-coach, cre-build, cre-disagree, cre-strategic, cre-decide. Intelligence and capability layer. Read-only consumers of kernel output. 28 modules (8 core engines always active, 20 optional intensifiers gated by subscription tier). Roadmap to 41 modules.
5	Infrastructure Crates	cre-types (shared type library), cre-llm-proxy (provider connections), cre-telemetry (privacy-safe audit logging), cre-providers (inference adapters).
6	Application Layer	Tauri desktop app + Svelte frontend. Targeting macOS and iOS simultaneous App Store launch. Normative Translation Layer renders kernel output as conversational UX.

The Dependency Graph

The dependency graph is strictly acyclic and one-directional. Information flows outward from the kernel. Nothing flows back.

```
cre-types  (leaf - depends only on serde)
    ^
cre-core   (cre-types, blake3, serde_json, thiserror)
    ^           ^
cre-api    cre-providers
    ^
cre-satellite-runtime
    ^
cre-coach  cre-build  cre-disagree  cre-strategic  cre-decide
```

The kernel (cre-core) depends on cre-types and three external Rust crates: blake3 for cryptographic sealing, serde_json for canonical serialization, and thiserror for structured error types. It depends on nothing else. Every other layer in the stack is forbidden from creating a reverse dependency into cre-core.

A note on lane classification for reviewers encountering the six-layer stack for the first time: lane assignment occurs in the membrane (cre-api) using structural signals and modal markers before any claim reaches the kernel. If classification is ambiguous, the membrane triggers LANE_AMBIGUITY halt. It does not guess. Cross-lane contamination in the kernel is not a runtime risk - it is a compile-time impossibility. A Truth-lane artifact is a

different Rust type than a Normative-lane artifact. The function that evaluates Normative claims does not accept Truth-lane artifacts as arguments. The compiler rejects the call.

The Generation Plane vs. the Authorization Plane

The architectural separation protected by the provisional patents divides the system into two distinct planes.

The Authorization Plane contains cre-core and cre-types. This plane adjudicates. It is deterministic, frozen, and governed by compile-time invariants. It has no awareness of the satellite capabilities, the UI, the LLM providers, or the telemetry layer. It produces verdicts. It does not produce suggestions.

The Generation Plane contains everything else: LLMs, satellite modules, the UI, the telemetry system, the LLM proxy. This plane is deliberately allowed to be dynamic, exploratory, and capable of probabilistic behavior. The key constraint: nothing in the generation plane can write to the authorization plane. The LLM can produce inadmissible output. The satellites can explore. The kernel stays clean.

Scope of Kernel Adjudication

A precise statement of CRE's epistemic scope is required here to prevent a class of reviewer misunderstanding that surfaces during technical due diligence. The kernel adjudicates structural admissibility - it does not verify empirical truth. These are different operations.

A deterministic logic engine can verify internal consistency, constraint compliance, logical contradiction, lane boundary enforcement, authority declaration, and premise stability. It cannot independently verify whether an empirical claim is true in the world without an evidence base the user supplies. What the kernel has is the structural properties of the reasoning itself - and those properties are what it evaluates.

A claim that is structurally admissible has earned the right to be asserted given its declared premises, evidence, and authority. It has not thereby been proven correct. This distinction is intentional, architecturally grounded, and fully consistent with CRE's role as an adjudication layer rather than an oracle. Scaling probabilistic models improves fluency. It does not close this structural absence. That is precisely why the adjudication layer must be built outside.

SECTION 2 The Kernel (cre-core)

What the Kernel Is

cre-core is a pure function. Given identical inputs and a fixed kernel version, it produces identical outputs. Byte for byte. On any machine. Without exception. This is not a performance characteristic or a design aspiration. It is the definitional property of what the kernel is - enforced by its architecture, its type system, its test suite, and its CI gates.

The kernel does not decide. It does not rank, score, optimize, or deliberate. It evaluates admissibility. Given a claim and its declared context, it determines whether the claim meets the structural conditions required for assertion in the declared epistemic lane. The answer is always one of three structural states: admitted, halted with a typed code, or underdetermined with a specific missing primitive identified.

> *INV-042: Kernel does not decide, rank, score, optimize, or deliberate; it is admissibility-only. Enforced by halt code exhaustiveness and the absence of any probability-output path in run_kernel.*

The run_kernel Entry Point

All kernel execution enters through a single function: run_kernel. This is not a stylistic choice. It is the enforcement mechanism for the kernel's purity guarantee. There is one entry point. There is one exit point. The execution path between them is a directed acyclic graph (DAG) - no recursion, no back-propagation, no lateral engine communication.

```
pub fn run_kernel(input: RunInput, opts: KernelOptions) -> KernelArtifacts
```

RunInput carries the claim, its declared epistemic lane, any evidence pack, any authority deck, any prior state for multi-turn continuation, and any requested move. KernelArtifacts returns the PublicOutputFrame (POF), the MoveMenu, and the ExecutionLog. Every field in KernelArtifacts is deterministically derived from RunInput. No ambient state. No environment reads. No timestamps from the system clock.

The Kernel Pipeline - Processing Phases

The kernel processes a claim through a strictly ordered sequence of gates. Each gate is identified by a PR number derived from the pull request that institutionalized it. Higher-priority violations halt execution and block all downstream gates.

Gate	Description	Halt Code
PR5	Continuation gate - validates schema-level continuation legality if a multi-turn move is requested	CRE_SCHEMA_VIOLATION
PR4	Move execution - validates requested move against current contract of allowed actions	CRE_INVALID_MOVE
PR1	Missing primitives gate - checks for fundamental structural prerequisites (lane declaration, etc.)	HaltMissingPrimitives
PR2	Lane enforcement - confines reasoning to declared epistemic lane	HaltLaneMismatch
CCG	Compression gate - halts if justification exceeds 500 character limit	CreCompressionFailure
PR3	Lane mismatch detection - explicit cross-lane contamination check	HaltLaneMismatch
PR6	Authority conflict resolution	AuthorityConflict
PR8	Claim kind classification (Analytic vs. EmpiricalTestable)	ClaimKindUnknown
PR9	Analytic closure gates - evaluates logical tautologies and contradictions	ClosedAnalyticTrue / False
PR10	Evidence pack requirement for empirical claims	HaltEvidenceRequired
PR11	Evidence lane contract enforcement	HaltLaneMismatch
PR13-14	Normative lane + authority deck validation	HaltAuthMissing
PR33	OPHRA outcome artifact - human override accountability	n/a (artifact generation)
PR100	CRI seal determinism verification	CapsuleVerificationFailed

The Kernel Freeze

The kernel is frozen at v1.0.0 at a specific commit hash. The freeze is not a convention or a policy. It is mechanically enforced by a CI zero-diff gate that rejects any commit that touches cre-core source files.

Modification is impossible without an explicit governance act: a CRI proposal capsule with human approval, regression gates passing before and after, and a CHANGELOG entry citing the authorizing RID.

The rationale for this level of rigidity is precise. A kernel that can be modified is a kernel that can drift. A kernel that can drift cannot make determinism guarantees. A kernel that cannot make determinism guarantees is not a governance layer - it is another probabilistic system with extra steps. The freeze is the guarantee. Everything else depends on it.

This does not mean the kernel cannot evolve. It means kernel evolution is governed, versioned, and consequential rather than incremental. Kernel versions are immutable once frozen. If a future governance review determines that a new kernel version is warranted, that version is introduced as a distinct frozen artifact alongside the prior version, not as a modification to it. Existing ReplayCapsules are permanently bound to the kernel version under which they were generated: the kernel_version field is embedded in the sealed bytes before hashing, making silent retroactive revision mathematically impossible.

> *The 'changing ruler' principle: a freely mutating evaluation core is like a ruler that changes length - it renders every past measurement retrospectively untrustworthy. The freeze is what makes historical ReplayCapsules meaningful.*

FORBIDDEN_CRATES Registry

The FORBIDDEN_CRATES registry is the compile-time enforcement mechanism for one-way dependency. It is a canonical list of crate names that cre-core must never depend on - directly or transitively. A CI test walks the full transitive dependency graph using cargo_metadata and fails the build if any FORBIDDEN_CRATE appears.

```
pub const FORBIDDEN_CRATES: &[&str] = &[
    "cre-coach", "cre-learn", "cre-providers", "cre-cli",
    "cre-smoke", "cre-spec-index", "cre-kernel",
    "cre-telemetry", "cre-api", "cre-llm-proxy", "cre-llm",
    "cre-proxy", "cre-scoring", "cre-ranking", "cre-optimizer",
    "cre-history", "cre-memory", "cre-storage", "cre-ui",
    "cre-web", "cre-server", "cre-gateway",
    "cre-satellite", "cre-expansion",
];
```

The registry carries a minimum count invariant (FORBIDDEN_CRATES_MINIMUM: 20) to prevent accidental truncation. Removing an entry from this list is itself a governed act requiring the same CRI proposal process as a kernel modification.

The Invariant Registry

The kernel is governed by 56 invariants, all currently at status 1 (passing). 51 are code-enforced at compile-time or by deterministic proof tests. 5 are Process Controls documented in GOVERNANCE_PROCESS_CONTROLS.md and required for merge approval. The full registry is maintained in docs/invariants.md. Primary invariants relevant to external reviewers:

ID	Invariant	Enforcement
INV-01	Kernel freeze: zero changes to cre-core/	FORBIDDEN_CRATES + git diff CI gate
INV-02	One-way dependency: kernel never imports satellites	cargo tree transitive closure test
INV-03	Determinism: 5-run byte-identical artifacts	Binary Audit Matrix - 5 independent runs

INV-04	No HashMap in artifact paths	grep + clippy - HashMap introduces ordering non-determinism
INV-05	Refusal superiority: confidence below floor triggers halt	Confidence floor test
INV-06	Monotonic degradation: confidence never increases	Monotonic degradation test suite
INV-07	POF-MoveMenu mirroring: next_valid_moves must match MoveMenu codes exactly	Post-run assertion on every evaluation
INV-08	Lane isolation: no cross-lane artifacts	Lane isolation test suite
INV-09	Terminal freeze: empty MoveMenu = terminal state, cannot be reopened	Post-verdict assertion
INV-10	No timestamps in artifacts - derived from run_id hash only	grep guard in CI
INV-11	Halt registry completeness: every HaltCode maps to a MoveMenu template	Exhaustive match - compiler enforces
INV-13	Coherence aggregation uses min(), not avg()	Unit test on aggregator function
INV-14	Counter-arguments are structural, never LLM-generated	Type enforcement - typed counter-arg structs only
INV-15	ReplayCapsule BLAKE3 seal matches re-hash on 5 independent runs	5-run seal verification
INV-042	Kernel is admissibility-only: no scoring, ranking, or deliberation output	Halt code exhaustiveness + absence of probability output
INV-043	MoveMenu must be present on every halt - user is never stranded	Post-run assertion + kernel_move_contract test
INV-044	Kernel is deterministic: no rand, no SystemTime, no env reads in kernel path	Code review + pure function architecture
INV-045	Provider outputs cannot write to ledger	gates.sh ledger-isolation cargo tree assertion

SECTION 3 Epistemic Lane Architecture

The Four Lanes

CRE enforces four mutually exclusive epistemic lanes. Every claim that enters the system must be classified into exactly one lane before evaluation can proceed. The classification is performed by the membrane (cre-api) using structural signals and modal markers in the claim text. If classification is ambiguous, the membrane triggers LANE_AMBIGUITY halt - it does not guess.

The lanes are not logical categories. They are Rust types. A Truth-lane artifact is a different type than a Normative-lane artifact. The function that evaluates Normative claims does not accept Truth-lane artifacts as

arguments. The Rust compiler rejects the call. Cross-lane contamination is compile-time impossible in the kernel. This is what the patent applications mean by 'epistemic type system enforcement.'

Truth Lane

Evaluates empirical and logical claims: what is factually or logically defensible against observable reality. Two claim kinds are recognized within the Truth lane: Analytic claims (tautologies and deductive proofs, evaluable to 100% confidence) and EmpiricalTestable claims (claims dependent on evidence, capped at 95% confidence). Evidence packs are required for EmpiricalTestable claims. Without an evidence pack, PR10 halts with HaltEvidenceRequired.

Normative Lane

Evaluates ethical and moral claims: what is right, wrong, permitted, or obligatory. Normative claims cannot be evaluated without an explicitly declared Authority Deck - a structured declaration of the governing framework (e.g., US Constitutional Law, a specific ethical framework, an organizational policy). The kernel initializes in a null assumption state: it presumes no default worldview and assumes no moral framework. Without an Authority Deck, PR13 halts with HaltAuthMissing. The Authority Deck is not optional. It is a required primitive.

Likelihood Lane

Evaluates probabilistic claims: what might happen. The Likelihood Lane rejects numeric probability scoring. Numeric probabilities are treated as an illusion of quantification that masks underlying epistemic fragility in non-repeatable environments. Instead, Plausibility-Ordered Reasoning (POR) is applied: claims are evaluated by their structural distance from collapse, not their statistical probability of success.

POR subjects every predictive claim to five structural failure modes: Scale (does the claim collapse when extended to larger sizes, volumes, or durations), Adversary (does it fail when an opponent actively exploits its vulnerabilities), Noise (does it break under real-world variability), Incentive Drift (does it erode as actor motivations shift over time), and Dependency Collapse (does it fail if external prerequisites fail). A claim's plausibility score is how much adversarial pressure it can absorb before structure fails.

> *Numeric probability scores - percentages, odds ratios, expected value calculations - are not admitted as structural conclusions in the Likelihood lane. They may appear in evidence supplied by the user as grounding material, but the kernel does not output them as evaluative verdicts. The reason is architectural: numeric probabilities carry the illusion of quantified precision while masking the underlying assumptions and reference class choices that determine their value. POR replaces that precision with something more defensible: structural distance from collapse under five specific failure modes.*

Necessity Lane

Evaluates constraint-based claims: what must be done given declared constraints. The Necessity Lane functions as a subtractive filter. It eliminates impossible paths. It does not optimize or recommend a 'best' choice - it presents only the options that survive declared constraints. A claim in the Necessity Lane that attempts to recommend rather than eliminate triggers a lane violation halt.

Lane Classification - The Membrane

The membrane (cre-api) performs lane classification before any claim reaches the kernel. Classification uses structural signals and modal markers - linguistic patterns that indicate the epistemic register of the claim. 'Must,' 'should,' 'ought' signal the Normative lane. 'Might,' 'could,' 'is likely' signal the Likelihood lane. 'Therefore,' 'necessarily,' 'given these constraints' signal the Necessity lane. Declarative present-tense factual assertions signal the Truth lane.

If signals conflict and classification cannot be resolved deterministically, the membrane triggers LANE_AMBIGUITY halt. The membrane does not attempt probabilistic classification. The claim either has a clear lane or it does not proceed.

Claims that carry simultaneous epistemic signals across multiple lanes - for example, a claim combining an empirical observation with a normative conclusion in the same sentence - do not receive automatic classification into the dominant lane. The membrane triggers LANE_AMBIGUITY and requires the user to declare the governing lane before evaluation proceeds. The system does not resolve multi-epistemic ambiguity by choosing; it surfaces it.

The membrane exposes two integration paths. Path A is for Rust-native Tauri desktop integration. Path B is for HTTP and external FFI environments. Both paths pass through identical schema validation. No internal kernel types are re-exported at the API boundary - all enum fields are represented as String at the surface, decoupling callers from internal type evolution.

SECTION 4 Core Data Types

PublicOutputFrame (POF)

The POF is the primary, user-facing payload deterministically derived from the kernel's internal state. It is never summarized or generated by a language model. It contains the exact epistemic context (lane), the verdict, specific halt codes, underdetermination flags, and structural confidence bands. It is the mandatory integration surface for all satellite modules - satellites are physically restricted to being terminal consumers of the POF.

Field	Description
lane	The epistemic lane of the evaluated claim. Option<Lane> - None on lane classification failure.
verdict	Enum: Admit / Refuse / Underdetermined. Kernel's structural determination.
halt_code	Specific HaltCode if halted. Every HaltCode maps to a MoveMenu template (INV-11).
next_valid_moves	Vec<String> of move codes. Must exactly mirror MoveMenu codes (INV-07). Schema violation if they diverge.
prohibited_moves	Vec<String> of explicitly prohibited moves. No overlap with next_valid_moves permitted.
underdetermination_flags	Structural gaps preventing closure. Missing primitives identified by type.
missing_primitives	Vec<MissingPrimitive> - specific typed declarations of what the kernel needs to proceed.
closure_confidence_band	Enum: VeryLow / Low / Medium / High / VeryHigh. Band, not numeric score.
confidence_index_0_100	Option<u8> - None at freeze v1. Non-None unauthorized until SASI regime specified (INV-041).
artifact_logs	Forensic audit vector. Gated behind emit_artifact_logs config flag. Disabled = kernel cannot accumulate shadow logs.

MoveMenu

The MoveMenu is the continuation contract. It enumerates the exact lawful next steps available from the current state. Every HaltCode in the system maps deterministically to a set of MoveOptionV1 entries. The mapping is

exhaustive - compiler enforces via exhaustive match on the HaltCode enum. If a new HaltCode is added without a corresponding MoveMenu mapping, the build fails.

MoveMenu and POF.next_valid_moves must mirror each other exactly. If they diverge at any point, the kernel panics with a schema violation. This invariant (INV-07) is checked post-run on every evaluation.

HaltCode	MoveMenu Response
HaltMissingPrimitives	PROVIDE_CLAIM_TEXT, SET_LANE (if lane absent)
HaltLaneMismatch	SWITCH_EPISTEMIC_LANE - restate or reclassify into correct lane
HaltAuthMissing	ATTACH_AUTHORITY_DECK - Normative lane requires explicit authority hierarchy
HaltNullSet	REVIEW_RULE_VIOLATIONS - inspect eliminations, optionally widen admissible search
HaltUnstable	INJECT_EVIDENCE_PRIMITIVES - add sourced evidence to improve stability
HaltLowConf	REVIEW_TRIPWIRE_LIST - acknowledge or refine tripwires before closure attempt
HaltDivergence	RESOLVE_ENGINE_CONFLICT - choose synthesis strategy for conflicting outcomes
AuthorityConflict	DECLARE_AUTHORITY, DISAMBIGUATE_AUTHORITY
ClaimKindUnknown	RECLASSIFY_CLAIM - explicitly declare Analytic or EmpiricalTestable
CreCompressionFailure	COMPRESS_INPUT - justification exceeds 500 character limit
ClosedAnalyticTrue	[empty - terminal state, claim is analytically true]
ClosedAnalyticFalse	[empty - terminal state, claim contains logical contradiction]

ReplayCapsule

The ReplayCapsule bundles the POF, MoveMenu, ExecutionLog, and LedgerTree into a single artifact and seals it with a BLAKE3 cryptographic hash. The sealing process enforces canonical JSON serialization before hashing: all object keys sorted lexicographically, array order preserved, cosmetic whitespace stripped, floating-point numbers banned (base64 for binary data). The resulting byte sequence is deterministic across all machines and all serialization environments.

```
let bytes = canonical_json_value(&capsule_data)?;
let seal = blake3::hash(&bytes).to_hex().to_string();
```

Before the API boundary emits a capsule, Emission Self-Verification runs: the system invokes its own canonical verifier on the generated artifact. If the seal fails internal verification, the system suppresses the artifact and returns CapsuleVerificationFailed. Corrupted evidence is never shipped.

The seal is regime-bound: the kernel_version and a deterministic specification hash are embedded in the bytes before hashing. Historical capsules are permanently bound to the governance rules active at the time of their creation. Silent retroactive revision is mathematically impossible.

Storage uses identity-derived, append-only architecture: the filename is derived from the BLAKE3 seal itself (seal_blake3_hex.json). The system deterministically refuses to overwrite existing files. History is immutable by construction.

A scrubbed variant (ReplayCapsule::scrubbed()) replaces raw claim text with BLAKE3 hashes of that text, preserving the verifiable audit trail without exposing claim content in the archive. This variant is designed for environments where claim content is confidential but evaluation integrity must be demonstrable.

CapabilityPackV1

The interface between the satellite runtime and individual satellite modules. Carries the active subscription tier and feature toggles. Satellites receive this as the first argument of every invocation. They must not read environment variables or access ambient state. Tier gating is performed exclusively through guard_by_tier - there is no satellite-owned gating logic.

```
pub struct CapabilityPackV1 {
    pub tier: SubscriptionTier,
    pub toggles: Vec<String>,
}
```

SubscriptionTier

CRE currently implements two tiers in the type system: Professional and Enterprise. The tier hierarchy is Professional < Enterprise. Higher tiers unlock additional satellite modules. The guard_by_tier function is the single source of truth for tier enforcement - no satellite implements its own gating logic.

SECTION 5 The Satellite Runtime

Architecture

The satellite runtime (cre-satellite-runtime) is the singular invocation chokepoint between the kernel output and the satellite capability modules. It is a pure function: zero state mutation, zero ambient environment access. Toggle independence is enforced - enabling or disabling one module must not alter the behavior of others. Graceful degradation is required - if a satellite is missing required inputs, it emits a formal RequiredPremise artifact rather than generating plausible output.

```
pub fn run_satellites(
    pof: &PublicOutputFrame,
    pack: &CapabilityPackV1,
    context: &ConversationContext,
) -> SatelliteOutputV1
```

SatelliteOutputV1 fields are all Option<T>. A disabled satellite produces None, not an error. The caller can determine from the returned struct exactly which satellites ran and which were gated by tier or toggle.

The Five Satellite Crates

Satellite capabilities are organized across five independent crates. Each crate is a one-way dependent on cre-types. None depend on cre-core. The FORBIDDEN_CRATES registry enforces this at the build level.

cre-disagree - Adversarial Analysis

Generates deterministic counter-arguments against claims that pass initial admissibility gates. Four structural counter-argument types: Negation Inversion (directly contradicts the claim conclusion), Premise Removal (tests claim survivability if a foundational premise is removed), Authority Challenge (contests whether declared authority is valid in this specific context), and Scope Stress (tests whether the claim holds when extended to adjacent cases).

Each unresolved counter-argument decrements the coherence confidence score by a fixed amount. Confidence can only decrease - Monotonic Confidence Degradation (INV-06). The adversarial engine never generates probabilistic objections. Counter-arguments are typed structural artifacts (INV-14). An LLM does not generate the counter-arguments. The adversarial engine generates them from the claim's structural properties.

cre-coach - Coaching and Guidance

Translates kernel verdicts into actionable typed guidance. Produces CapabilityPackV1 artifacts categorized as Assumptions, Contradictions, Required Premises, Failure Modes, or Evidence Requests. Does not produce free-form text. If required inputs are missing, emits a formal RequiredPremise rather than generating plausible output.

cre-strategic - Strategic Analysis

Operates at the level of game-theoretic reasoning. Maps incentive structures, identifies premise dependencies on actor behavior, surfaces Dependency Collapse risks in multi-party claims. Most relevant for claims whose structural integrity depends on the real-world architecture of incentives rather than internal logic alone.

cre-build - Claim Construction

Supports construction of structurally admissible claims. Identifies what a claim that would pass requires - explicit premises, declared authority, correct lane classification. Functions as a repair guide for claims that fail admissibility gates, translating halt codes into specific construction requirements.

cre-decide - Decision Support

Supports multi-option decision evaluation within the Necessity Lane. Applies declared constraints as eliminative filters. Does not rank or recommend - eliminates inadmissible options and presents only those that survive constraint evaluation. Generates CrossEngineSynthesisObject (CESO) artifacts when multi-lane tensions require explicit mapping.

The CrossEngineSynthesisObject (CESO)

The CESO is generated when a complex claim requires routing through multiple epistemic lanes simultaneously and the lanes reach diverging conclusions. It maps inter-lane tensions, identifies bottlenecks preventing reconciliation, and outlines the conditions under which lanes could be resolved. It does not synthesize the lanes into a single answer - synthesis is the failure mode CRE is built to prevent.

Every CESO carries the mandatory disclaimer: 'This is synthesis, not a verdict.' The kernel issues verdicts. The CESO is a navigation instrument. Its authority is explicitly zero - it is an advisory output from the generation plane, not a determination from the authorization plane.

SECTION 6 The Coherence Score

Five-Dimensional Measurement

The five-dimensional coherence score is computed by the satellite runtime over the accumulated ConversationContext. It is not computed by the kernel. The kernel produces binary admissibility determinations.

The coherence score is a satellite-layer signal that provides continuous structural health feedback to the user interface.

Dimension	What It Measures
Term Stability	Consistency of key term usage across the conversation. Detects semantic drift - terms that begin with one meaning and acquire another without declaration.
Premise Consistency	Survival of foundational premises established early in the conversation. Detects premise abandonment under conversational pressure.
Lane Integrity	Whether the conversation is staying within its declared epistemic lane. Detects cross-lane contamination before it triggers a kernel halt.
Authority Adherence	Whether claims requiring explicit authority are properly grounded. Detects authority laundering at the pre-halt level.
Positional Delta	Whether the AI position has shifted from its initial state. Measures whether shift is driven by evidence or by conversational momentum.

Aggregation - min(), Not avg()

The aggregate coherence score is the minimum of the five dimension scores, not the average. This is INV-013, enforced by a dedicated unit test on the aggregator function. The rationale: a conversation that scores perfectly on four dimensions but has one dimension in structural distress is a conversation with a structural gap. Averaging would obscure that gap. The minimum function surfaces it.

Traffic Light Thresholds

Three traffic light states are computed from the aggregate coherence score:

State	Threshold and Meaning
GREEN	>= 0.8 Conversation is structurally sound across all five dimensions
YELLOW	0.5 - 0.8 Structural signal - monitoring. Satellite-layer UX signal. Does not trigger kernel halt.
RED	< 0.5 Structural halt warranted

YELLOW does not trigger a kernel halt. It is a UX signal from the satellite layer indicating structural stress that warrants user attention. Only the kernel can trigger a halt. The coherence meter can show YELLOW while the kernel continues to admit claims - the meter is measuring conversational trajectory, not individual claim admissibility.

SECTION 7 Infrastructure Crates

cre-types

The foundational shared type library. The single source of truth for all types shared across the system - SubscriptionTier, CapabilityPackV1, PublicOutputFrame, ReplayCapsule, CoherenceScoreV1, AdversarialResultV1, MoveMenuTemplate, and all HaltCode variants. cre-types depends only on serde. No other crate in the system may define competing versions of these types. All cross-module interaction goes through cre-types schemas (GOV-05).

cre-llm-proxy

Manages external LLM provider connections. The LLM proxy is an unprivileged execution substrate. Its outputs enter the system exclusively as untrusted proposals through the membrane. The proxy has no write access to the kernel, the ledger, or any artifact store (INV-045 - provider outputs cannot write to ledger, enforced by gates.sh ledger-isolation cargo tree assertion).

The proxy architecture is provider-agnostic. NullProvider (always available, no network) and OllamaProvider (available when OLLAMA_URL is set) are implemented. Additional providers are added through the provider interface without touching the kernel. The multi-model fan-out architecture - routing a query to multiple providers in parallel and running each response through the CRE kernel independently - is a natural extension of the proxy design.

cre-telemetry

Handles privacy-safe, append-only audit logging. Raw claim text is structurally stripped from the POF before any telemetry data is collected. The telemetry system logs structural events - halt codes, lane classifications, coherence dimension scores, missing primitive types - without storing the content of claims. This architecture satisfies the PrivacyPolicyConfig requirement: telemetry is collected transparently with explicit user consent, and the data architecture makes raw claim content collection impossible by design rather than by policy.

cre-learn

The Human Learning Subsystem (HLS) vault. Manages the data and learning architecture for CRE. Three persistence layers: Full Archive (complete evaluation sessions with input/output pairs, authority decks, evidence packs, spec versions, and ReplayCapsule artifacts for deterministic replay), Structured Knowledge (processed, anonymized structural patterns), and Operational Cache. All learning is CRI-mediated and auditable. There is no autonomous truth ingestion.

SECTION 8 The Application Layer

Tauri and the Headless Kernel Architecture

The kernel is headless - completely divorced from the user interface by design. It is deployed locally as an offline-first, sandboxed sidecar binary invoked via a Tauri bridge. Tauri provides the deployment mechanism for desktop environments on macOS and iOS, managing the IO boundary without creating a dependency between kernel logic and UI logic.

The Tauri bridge acts as a strict pass-through. A serialized RunInput enters the kernel. KernelArtifacts are returned. The Tauri layer executes offline. It forbids telemetry at the kernel level, auto-updates to kernel behavior, and hidden cloud dependencies in the kernel path. The local adjudication environment is a sealed epistemic vacuum.

The Svelte Frontend

The frontend is built in Svelte and TypeScript. It translates the kernel's structural outputs into a conversational interface. Three primary UI responsibilities: rendering the five-dimensional coherence meter as a live traffic light,

rendering the MoveMenu as plain-language 'Next Steps' pill cards via the Normative Translation Layer (NTL), and rendering the Side-by-Side View when structural gaps warrant direct comparison.

The Normative Translation Layer intercepts raw kernel machine codes (HaltCodes, MoveMenuActionIds) and translates them into plain language. The NTL is a pure rendering layer - it does not modify kernel output, add information, or interpret beyond the typed artifacts it receives. Translation is deterministic: each machine code maps to exactly one plain-language rendering.

Parallel Dispatch Architecture

The application dispatches two requests simultaneously for each user turn. One request goes to the LLM proxy to capture the raw, ungoverned response (the Baseline Envelope). One request goes through the local CRE kernel for deterministic evaluation. The side-by-side view is powered by this parallel dispatch - the left panel shows the Baseline Envelope, the right panel shows the governed output. The gap between them is the structural difference the governance layer enforced.

This architecture ensures that users can see the ungoverned output they would have received without CRE, making the governance layer's operation concrete and verifiable rather than abstract.

SECTION 9 CI/CD Governance

The Gate Scripts

Three gate scripts must pass on every commit. No --no-verify is allowed. No #[ignore] macro is permitted on freeze proof tests (enforced by a bash grep scan in gates.sh that fails CI if #[ignore] appears in any crates/*/tests/ file).

```
cargo fmt --all -- --check
cargo clippy --workspace --all-targets --all-features -- -D warnings
cargo test --workspace --all-targets --all-features -- --test-threads=1
```

Tests run single-threaded (--test-threads=1) to prevent any ordering-dependent non-determinism in the test suite. A test that passes in one thread ordering and fails in another is a determinism violation.

The Spec-Code-Proof Doctrine

Every kernel behavior must satisfy three conditions simultaneously: it must be specified (documented in the canonical spec), implemented (present in cre-core), and proven (covered by a deterministic test that fails the build if violated). A behavior that is specified and implemented but not proven is not covered. A behavior that is implemented and tested but not specified is not canonical. The doctrine requires all three.

The CI gates enforce the 'paper-green' kill switch: no developer can silence a failing invariant test with #[ignore] to produce a passing build. If the test fails, the build fails. The specification, the code, and the proof are permanently locked together.

The Binary Audit Matrix

Determinism is verified by running the kernel five independent times against identical inputs and comparing the resulting artifacts byte-for-byte. This is INV-03. If any run produces a divergent artifact - different bytes, different hash - the determinism invariant has failed and the build is rejected. This test catches any non-deterministic behavior that might be introduced by dependency updates, compiler changes, or environmental differences.

The Constitutional Matrix

The constitutional matrix maps every behavioral gate to its specification reference, implementation location, and proof test. It is maintained as a living document and checked in CI. The matrix is complete when every surface of kernel behavior - every HaltCode, every admissibility path, every terminal state, every artifact field - has a corresponding entry. As of March 2026: 56/56 invariants at status 1.

SECTION 10 Patent Coverage Map

USPTO Provisional Patent Applications

Two USPTO provisional patent applications protect the core architectural innovations of CRE: #63/977,457 and #63/988,849. Both applications must be cited together when referencing the IP. The provisional clock expires February 6, 2027. Non-provisional conversion is targeted for Q4 2026.

Application #63/977,457 - Deterministic Epistemic Adjudication as an External Layer

This application covers the architecture of deterministic epistemic adjudication as a layer structurally external to probabilistic generative models - the separation of generation from authorization, enforced at the type system level.

The novel claim is not merely that an external evaluation layer exists. It is that the separation is enforced at compile time through an epistemic type system: Truth-lane artifacts are distinct Rust types from Normative-lane artifacts. The function that evaluates Normative claims does not accept Truth-lane inputs as arguments. The compiler rejects the call before runtime. The invariant is structural, not behavioral.

Implementation: the Lane enum in cre-types, the lane-typed evaluation functions in cre-core, the FORBIDDEN_CRATES registry, the one-way dependency architecture, and the CI zero-diff gate on cre-core collectively implement this claim. The kernel is not merely designed to be external to the LLM - it is architecturally incapable of being contaminated by it.

Application #63/988,849 - Cryptographically Bound Self-Verifying Outputs

This application covers the ReplayCapsule architecture: cryptographically bound self-verifying outputs that make every evaluation independently verifiable without trusting the system that produced it.

The novel claim is the combination of deterministic execution (same inputs always produce same outputs) with regime-bound cryptographic sealing (the seal binds the output to the exact governing specification version active at evaluation time) with Emission Self-Verification (the system verifies its own capsule before emitting it, and suppresses corrupted artifacts rather than distributing them). Together, these three properties produce an artifact that any independent auditor can verify on any machine without access to the original system, and that cannot be retroactively revised without detection.

Implementation: the canonical_json_value() function, the BLAKE3 seal computation, the regime identifier embedding (kernel_version + spec_version_hash in the sealed bytes), Emission Self-Verification, identity-derived append-only storage, and the ReplayCapsule::scrubbed() privacy variant collectively implement this claim.

What No Prior Art Covers

Neither of these innovations - type-system-enforced epistemic lane isolation in an external adjudication layer, nor cryptographically bound regime-specific self-verifying AI output artifacts - exists in prior art identified during the patent application process. Constitutional AI, RLHF, guardrail systems, and retrieval-augmented generation all operate within the probabilistic model's embedding space. None of them produce structurally external, compile-time-enforced, cryptographically sealed, independently verifiable evaluation artifacts.

SECTION 11 Enterprise Integration Guide

Integration Architecture

CRE exposes two integration paths through the cre-api boundary layer. Both paths provide identical kernel behavior. The choice between them is an integration environment decision, not a capability decision.

Path A - Rust-Native / Tauri Desktop

Direct Rust function call to run_api_v1 with a typed ApiRequestV1 struct. Zero serialization overhead. Full type safety at the integration boundary. Appropriate for desktop applications built with Tauri, for CI/CD pipeline integrations, and for server-side Rust deployments.

```
let response = cre_api::run_api_v1(request, opts)?;
```

Path B - HTTP / FFI External Environments

JSON over HTTP through run_api_v1_json, the preferred entry point for external callers. Handles deserialization and enforces version-gated strictness before dispatching. Two API versions are accepted: v1 (permissive mode - unknown fields in nested types discarded) and v2 (strict mode - unknown fields at any nesting level rejected).

```
let response = cre_api::run_api_v1_json(&json_string)?;
```

API Versioning

Both API versions (v1 and v2) share the same kernel path. Kernel behavior is identical. The version controls input parsing strictness only. v2 is recommended for production enterprise deployments where unknown field injection is a security consideration. Future incompatible kernel changes would add a new entry point alongside run_api_v1 rather than modifying it.

Preflight Gates

The cre-api boundary implements sequential preflight gates that shield the kernel from malformed or malicious inputs. Gates execute in order: API version validation (Gate 1), authority deck requirement check (Gate 2), evidence pack requirement check (Gate 3). Each gate can produce an early rejection without touching kernel state. Rejected requests return gate-specific error codes with capsule_json: None.

ReplayCapsule Export for Enterprise Audit

Enterprise deployments that require evidentiary-grade output for regulatory, legal, or compliance purposes should enable capsule emission via the emit_capsule flag in KernelOptions. The resulting ReplayCapsule satisfies the following audit requirements:

- Tamper-evident: BLAKE3 seal breaks if any byte is altered after emission
- Regime-bound: seal embeds kernel_version and spec_version_hash - historical capsules remain bound to their original governance rules
- Independently verifiable: any auditor can recompute the seal and replay the reasoning path offline without access to the original system
- Privacy-safe: ReplayCapsule::scrubbed() variant available for environments requiring claim content confidentiality
- Append-only storage: identity-derived filenames, deterministic refusal to overwrite - immutable history by construction

OPHRA - Authorized Human Override

When an authorized human operator decides to override a system halt, the architecture physically prevents mutation of the original ReplayCapsule. Instead, a separate OPHRA override artifact is generated that references the original capsule's BLAKE3 seal. The kernel's initial refusal and the human's sovereign decision to override remain mathematically distinct in the permanent audit trail. This architecture satisfies enterprise requirements for accountable human oversight without compromising the integrity of the original evaluation record.

Telemetry and Privacy Architecture

Enterprise deployments that require telemetry collection should configure the PrivacyPolicyConfig to align with organizational data governance requirements. The telemetry architecture provides the following guarantees by design:

- Raw claim text is structurally stripped from the POF before any telemetry system touches the data
- The emit_artifact_logs flag gates shadow log accumulation - when disabled, the kernel is physically incapable of logging claim content
- Telemetry data covers structural events only: halt codes, lane classifications, coherence dimension scores, missing primitive types
- All telemetry collection requires explicit user consent - this is a configured gate, not a default behavior

ARCHITECTURAL SUMMARY

The Coherence Reality Engine is a deterministic epistemic adjudication system built outside the probabilistic models it governs. The kernel is frozen, compile-time-governed, and cryptographically sealed. The satellite capabilities are extensible and exploratory. The line between them is enforced by the Rust compiler, the FORBIDDEN_CRATES registry, the CI zero-diff gate, and 56 invariants at status 1.

The architectural claim protected by USPTO Provisional Patent Applications #63/977,457 and #63/988,849 is precise: epistemic authorization cannot be fixed from inside a probabilistic model. It must be built outside, as a structurally independent deterministic adjudication layer, with the separation enforced at the type system level and the outputs sealed cryptographically to the governing specification version that produced them.

Every claim in this document about enforcement, determinism, and invariant coverage has a corresponding test, CI gate, or compiler mechanism. Where the claim is architectural rather than test-proven, that distinction has been noted. The system does not claim what it cannot demonstrate.

While the industry trains the prosecutor, we are building the court.

Invaris AI | Building the epistemic infrastructure layer for the AI era.
Protected by USPTO Provisional Patent Applications #63/977,457 and #63/988,849