Product Requirements Document: Containment Certificate Protocol

Working Name: CCP (Containment Certificate Protocol) **Status:** Draft v0.1 **Date:** April 2026 **Authors:** [TBD]

---

1. Problem Statement

AI agents are becoming autonomous economic participants — holding wallets, signing transactions, paying for services. Infrastructure for identity (KYA, TAP, ERC-8004), wallets (OWS, ERC-7710), and payments (x402, MPP) is shipping fast. But no standard exists for the structural trust layer: a machine-readable, verifiable way for an agent to prove to any counterparty that its worst-case economic impact is bounded and backed.

Current approaches rely on behavioral reputation (credit scores, staking history, karma systems). These fail for LLM-based agents because:

- The agent is probabilistic — past behavior weakly predicts future behavior
- The agent is non-stationary — model updates silently change the scored entity
- The agent is ephemeral — spinning up a new identity is nearly costless

The result: counterparties have no reliable way to assess how much they can lose from interacting with an agent, and no way to verify that loss is bounded and absorbed.

What this product is

A protocol and on-chain standard for issuing, publishing, and verifying **containment certificates** — machine-readable attestations that an agent's economic impact is bounded by agent-independent constraints and backed by exogenous reserves.

What this product is not

- Not a reputation system or credit score
- Not a wallet or payment protocol
- Not an agent framework or orchestration layer

- Not a regulatory compliance tool (though it may support compliance)

---

2. Core Thesis

Trust for probabilistic agents should be modeled as bounded-loss architecture, not behavioral reputation. The certificate attests to the *containment system*, not the *agent's character*. A counterparty verifying a certificate is answering one question: **does the surrounding system make this agent economically safe enough to transact with?**

---

3. Users & Personas

3.1 Agent Operators (Issuers)

Companies or individuals deploying AI agents for economic activity. They configure containment architecture (smart contract limits, permission scopes, reserves), obtain audits, and issue certificates for their agents.

Need: Credible, portable proof that their agents are economically safe — enabling counterparties to transact without bilateral negotiation.

3.2 Counterparties (Verifiers)

Other agents, protocols, merchants, or humans deciding whether to transact with a certified agent. They parse the certificate against their own risk policy.

Need: Machine-readable, on-chain-verifiable answer to "what is my maximum exposure from this interaction, and is it backed?"

3.3 Auditors (Attestors)

Independent firms or DAOs that verify containment architectures — formal verification of smart contracts, penetration testing of permission models, reserve adequacy assessment — and sign certificates.

Need: Clear standard for what to audit, how to attest, and how to publish findings.

3.4 Protocol Integrators

DeFi protocols, marketplaces, payment rails, and agent registries that want to gate access or adjust terms based on containment quality.

Need: A queryable on-chain interface to check whether an agent has a valid certificate

meeting minimum containment thresholds.

---

4. Key Concepts

4.1 Agent-Independent vs. Agent-Influenceable Containment

The critical design distinction. Agent-independent containment mechanisms cannot be influenced, circumvented, or degraded by the agent through its outputs. Examples: formally verified smart contracts, TEEs, HSMs, MPC approval flows. Agent-influenceable containment depends at some point on human judgment or a system the agent can interact with. Examples: human oversight, reputation gates, mutable configuration.

The certificate must distinguish between these two categories for every constraint.

4.2 Containment Bound

The worst-case economic loss the agent can produce if ALL agent-influenceable layers are compromised and only agent-independent layers hold. This is the number that matters for counterparty risk assessment.

4.3 Exogenous Reserve

Collateral backing the residual risk, denominated in assets whose value is independent of the agent's own ecosystem. Not self-minted tokens. Not governance tokens. Stablecoins, ETH, or equivalent. Must be held in a smart contract the agent cannot unilaterally access.

4.4 Risk Function (Not Risk Score)

Counterparties evaluate certificates through a function, not a scalar:

> R = P_a × P_joint_failure(containment_layers, correlation) × L

Different counterparties may weight variables differently based on their own risk tolerance. The certificate provides the inputs; the counterparty applies the function.

---

5. Certificate Schema

5.1 Structure

```
ContainmentCertificate {
```

```
// --- Identity ---
version: "ccp-v0.1"
certificate_id: bytes32              // unique identifier
agent_id: address                    // agent's on-chain address
operator_id: address                 // deployer/operator address
chain_id: uint256                    // chain where certificate is published

// --- Validity ---
issued_at: uint256                   // block timestamp
expires_at: uint256                  // expiry timestamp
status: enum { ACTIVE, REVOKED, EXPIRED }

// --- Constraints ---
constraints: [
  {
    constraint_id: bytes32
    type: enum {
      MAX_SINGLE_ACTION_LOSS,
      MAX_PERIODIC_LOSS,
      PERMISSION_SCOPE,
      REVERSIBILITY_WINDOW,
      HUMAN_OVERSIGHT_THRESHOLD
    }
    value: uint256                   // amount or duration
    denomination: string             // "USDC", "ETH", "seconds"
    period: uint256                  // seconds (for periodic constraints)
    enforcement: enum {
      SMART_CONTRACT,
      TEE,
      HSM,
      MPC,
      OFF_CHAIN
    }
    contract_address: address        // if on-chain enforcement
    formally_verified: bool
    verification_proof_uri: string   // IPFS hash of verification report
    agent_independent: bool          // THE critical field
  }
]

// --- Reserve ---
reserve: {
  amount: uint256
  denomination: string
  contract_address: address          // reserve custody contract
  reserve_type: enum { ESCROW, INSURANCE_POOL, STAKED }
  reserve_ratio: uint256             // ratio to max periodic loss (basis point
```

```
    exogenous: bool                    // true = value independent of agent ecosy
  }

  // --- Derived Metrics ---
  containment_bound: uint256           // worst-case loss assuming only
                                       // agent-independent layers hold
  agent_independent_layer_count: uint8
  total_layer_count: uint8

  // --- Attestations ---
  attestations: [
    {
      auditor_id: address
      auditor_name: string
      scope: enum {
        SMART_CONTRACT_VERIFICATION,
        PERMISSION_MODEL_AUDIT,
        RESERVE_ADEQUACY,
        FULL_STACK
      }
      date: uint256
      report_uri: string               // IPFS hash
      signature: bytes                 // auditor's signature over certificate ha
    }
  ]

  // --- Operator Metadata ---
  operator_metadata: {
    operator_name: string
    operator_track_record_uri: string  // optional link to operator history
    model_type: string                 // e.g., "gpt-4o-2026-03", "claude-opus-4.
    model_version_attested: bool       // true if TEE attests model version
  }

  // --- Certificate Signature ---
  certificate_hash: bytes32
  issuer_signature: bytes              // operator signs
  attestor_signatures: [bytes]         // auditors co-sign
}
```

5.2 On-Chain Registry

A minimal smart contract registry that stores certificate hashes and provides lookup:

```
interface ICCPRegistry {
```

```
    // Publish a certificate (operator or authorized issuer)
    function publish(bytes32 certificateHash, bytes calldata certificate)
      external;

    // Revoke a certificate (operator only)
    function revoke(bytes32 certificateHash) external;

    // Check if a certificate is valid (not expired, not revoked)
    function isValid(bytes32 certificateHash)
      external view returns (bool);

    // Get certificate data
    function getCertificate(bytes32 certificateHash)
      external view returns (bytes);

    // Lookup by agent address
    function getActiveCertificate(address agentId)
      external view returns (bytes32 certificateHash);

    // Events
    event CertificatePublished(
      bytes32 indexed certificateHash,
      address indexed agentId,
      address indexed operatorId,
      uint256 expiresAt
    );
    event CertificateRevoked(bytes32 indexed certificateHash);
}
```

5.3 Verification Flow

A counterparty (human or agent) verifies before transacting:

1. **Lookup:** Query registry by agent address → get active certificate hash
2. **Retrieve:** Fetch full certificate from on-chain storage or IPFS
3. **Validate signatures:** Verify operator and attestor signatures
4. **Check status:** Not expired, not revoked
5. **Evaluate constraints:** Parse constraints against counterparty's risk policy
    - Are the loss bounds within my tolerance for this transaction?
    - Are the critical constraints agent-independent?
    - Is the reserve exogenous and adequately sized?

6. **Decision:** Accept, reject, or request additional constraints

Steps 1–6 can be fully automated — an agent evaluating whether to transact with another agent by parsing the counterparty's certificate programmatically.

---

6. Functional Requirements

6.1 Certificate Lifecycle

ID	Requirement	Priority
F1	Operator can create and publish a certificate for an agent	P0
F2	Certificate is stored on-chain (registry) with full data on IPFS	P0
F3	Certificate has a defined expiry and can be renewed	P0
F4	Operator can revoke a certificate at any time	P0
F5	Only one active certificate per agent address at a time	P0
F6	Certificate status transitions: ACTIVE → REVOKED or ACTIVE → EXPIRED	P0
F7	Certificate versioning (schema version field) for forward compatibility	P1

6.2 Constraint Specification

ID	Requirement	Priority
F8	Each constraint specifies enforcement type (smart contract, TEE, HSM, MPC, off-chain)	P0
F9	Each constraint includes `agent_independent: bool` flag	P0
F10	For on-chain enforcement, constraint links to the specific contract address	P0
F11	For formally verified constraints, certificate links to verification proof (IPFS)	P0
F12	Certificate computes `containment_bound` from agent-independent constraints only	P0

F13	Constraints support at minimum: max single-action loss, max periodic loss, permission scope, reversibility window, human oversight threshold	P0
F14	Constraint types are extensible (enum can be extended in future versions)	P1

6.3 Reserve Specification

ID	Requirement	Priority
F15	Reserve specifies amount, denomination, and custody contract	P0
F16	Reserve includes `exogenous: bool` flag	P0
F17	Reserve ratio is expressed relative to max periodic loss	P0
F18	Reserve custody contract is verifiable on-chain (anyone can check balance)	P0
F19	Reserve cannot be unilaterally withdrawn by the agent	P0

6.4 Attestation

ID	Requirement	Priority
F20	Auditor signs the certificate hash with their on-chain key	P0
F21	Attestation specifies scope (what was audited)	P0
F22	Attestation links to full audit report on IPFS	P0
F23	Certificate can have multiple attestations (different auditors, different scopes)	P0
F24	Auditor registry (optional): on-chain list of recognized auditors	P1

6.5 Verification & Query

ID	Requirement	Priority
F25	Any address can query the registry to check if an agent has a valid certificate	P0
F26	Query returns full certificate data or a pointer to IPFS	P0

F27	Verification can be performed entirely on-chain (for smart contract integrations)	P0
F28	SDK/library for off-chain verification (TypeScript, Python)	P1
F29	x402/MPP integration: certificate can be included in payment challenge-response flow	P2

---

7. Non-Functional Requirements

ID	Requirement	Priority
NF1	Registry contract is immutable after deployment (no admin keys, no upgrade proxy)	P0
NF2	Registry contract is formally verified	P0
NF3	Gas cost for publish < 500k gas; lookup < 50k gas	P1
NF4	Schema supports multi-chain deployment (same contract on multiple EVM chains)	P1
NF5	Certificate data is human-readable (JSON) as well as ABI-encoded	P1
NF6	Protocol is permissionless — anyone can publish, anyone can verify	P0
NF7	No governance token, no protocol fee, no rent extraction	P0

---

8. Cross-Domain Portability

One of the key open problems identified in the essay: a certificate on Ethereum says nothing to Solana or Visa's TAP. The protocol should address this progressively:

Phase 1: Single-chain EVM

Deploy on one EVM chain (e.g., Base, Avalanche C-Chain). Certificate references contracts on the same chain. Verification is fully on-chain.

Phase 2: Multi-chain EVM

Deploy identical registry on multiple EVM chains. Cross-chain certificate references use

chain_id + contract_address pairs. Off-chain verification SDK aggregates across chains.

Phase 3: Cross-ecosystem

Define a chain-agnostic certificate format (JSON + signature) that can be verified off-chain by any system. Bridge to non-EVM chains (Solana, Cosmos) via attestation relays. Integration with Visa TAP, Skyfire KYA, and other centralized identity systems as optional metadata.

---

9. Integration Points

9.1 Agent Wallet Protocols

- **MoonPay OWS:** Certificate published alongside wallet creation. OWS policy engine references certificate constraints.
- **ERC-7710 Delegations:** Delegation scope matches certificate permission scope. Certificate proves the delegation is backed by reserves.
- **ERC-4337 Smart Accounts:** Certificate constraints enforced as UserOperation validation logic.

9.2 Payment Protocols

- **x402:** Server includes minimum certificate requirements in 402 challenge. Client agent includes certificate hash in payment credential. Server verifies before accepting payment.
- **MPP:** Session pre-authorization references certificate. Session spending limits cannot exceed certificate's max periodic loss.
- **Skyfire KYA:** KYA JWT extended with `ccp_certificate_hash` field. Merchants can verify both identity (KYA) and containment (CCP) in one flow.

9.3 Agent Registries

- **ERC-8004:** Containment certificate referenced in Identity Registry metadata. Reputation Registry scores can weight containment quality. Validation Registry can verify certificate validity as a validation hook.
- **Protocol access control:** DeFi protocols can gate access by requiring a valid certificate with minimum containment thresholds (e.g., "agent must have agent-independent max periodic loss ≤ 10,000 USDC and reserve ratio ≥ 3x").

10. Risk & Limitations

10.1 What the certificate does NOT guarantee

- **Agent correctness:** The certificate says nothing about whether the agent will make good decisions. It bounds the *damage* from bad decisions.
- **Model behavior:** The certificate does not attest to the agent's alignment, capabilities, or output quality.
- **Legal liability:** The certificate does not assign or transfer legal liability. It provides evidence of containment quality that may be relevant in liability assessment.
- **Absolute safety:** All containment has residual failure probability. The certificate quantifies and structures that residual; it does not eliminate it.

10.2 Attack vectors on the protocol itself

- **Stale certificates:** Agent's containment architecture may change after certificate issuance. Mitigation: expiry dates, revocation, re-attestation requirements.
- **Fraudulent attestation:** Auditor signs a certificate for a containment architecture they did not actually verify. Mitigation: auditor reputation, auditor staking (P2), multiple independent attestors.
- **Reserve manipulation:** Operator funds reserve at certification time, then withdraws. Mitigation: reserve is held in a smart contract that enforces lock-up through certificate expiry; real-time on-chain balance verification.
- **Constraint circumvention:** Agent finds a path around constraints (e.g., composition gap between two contracts). Mitigation: formal verification of constraint contracts; full-stack audits; containment bound assumes agent-influenceable layers are compromised.

10.3 Systemic risk

If many agents share the same containment architecture (same smart contracts, same auditors, same model), a single vulnerability could invalidate many certificates simultaneously. The protocol should encourage diversity but cannot enforce it. Monitoring systemic concentration (how many certificates depend on the same contracts) is a P2 feature.

---

11. Phased Delivery

Phase 0: Specification (Current)

- Finalize certificate schema
- Publish essay + PRD for community feedback
- Identify initial auditor partners

Phase 1: MVP — Single-Chain Registry

- Deploy registry contract on one EVM chain
- CLI tool for operators to create, sign, and publish certificates
- TypeScript/Python SDK for verification
- One reference integration (e.g., with an existing agent wallet or payment protocol)
- 2–3 pilot certificates with real agents and real auditors

Phase 2: Ecosystem Integration

- Multi-chain deployment
- Integration with x402 / MPP payment flows
- Integration with ERC-8004 / ERC-7710
- Auditor registry (optional, on-chain)
- Dashboard for browsing and comparing certificates

Phase 3: Advanced Features

- Cross-ecosystem portability (non-EVM, off-chain systems)
- Dynamic certificate updates (constraint changes trigger re-attestation)
- Systemic concentration monitoring
- Auditor staking / slashing for fraudulent attestation
- Risk function marketplace (standardized risk evaluation templates)

---

12. Success Metrics

Metric	Phase 1 Target	Phase 2 Target

Certificates published	5–10 pilot	100+
Unique agent operators	3–5	30+
Independent auditors	1–2	5+
Protocol integrations	1 reference	3+ (wallet, payment, registry)
Verification queries/day	Proof of concept	1,000+
Community contributors	5–10	20+

---

13. Open Questions

1. **Should the registry be upgradeable or immutable?** Current spec says immutable. This maximizes trust but limits evolution. Alternative: versioned registries (v1, v2) with migration tooling.
2. **How should auditor credibility be established?** Phase 1 relies on auditor reputation. Phase 3 could introduce auditor staking. Is there a lightweight middle ground?
3. **Should the protocol charge fees?** Current spec says no. This maximizes adoption but requires alternative funding (grants, ecosystem support). Revisit if sustainability becomes an issue.
4. **How to handle certificate updates?** If an operator adds a new constraint or increases reserves, should that require a new certificate or an amendment to the existing one?
5. **What is the minimum viable certificate?** A pilot agent doing $100/day in transactions does not need the same rigor as one doing $1M/day. Should there be tiered certificate classes?
6. **How does this interact with the EU Product Liability Directive?** A valid containment certificate could serve as evidence of "reasonable care" by the deployer. Legal review needed.
7. **Can the risk function itself be standardized?** Or should the protocol only provide inputs and leave the function to each counterparty?

---

14. Relationship to the Essay

This PRD operationalizes the framework described in "Trust Infrastructure for Probabilistic

Agents." The essay provides the conceptual foundation; this document specifies the product.

Essay Concept	PRD Implementation
Agent-independent vs. agent-influenceable containment	`agent_independent: bool` field on every constraint
Containment bound	`containment_bound` derived metric in certificate
Exogenous reserves	`reserve.exogenous: bool` + on-chain custody verification
Risk function, not risk score	Certificate provides inputs; counterparty applies function
Operator reputation as marginal signal	`operator_metadata` + `operator_track_record_uri` — optional, not load-bearing
Machine-readable, on-chain, verifiable	Registry contract + IPFS storage + verification SDK

This PRD is a working draft intended for collaborative development alongside the essay. It is a seed for mechanism design, not a finished specification.