IA Descentralizada y Verificable + DAO de Memoria Finita

Sistema Integrado de Inteligencia Artificial Común y Gobernanza Resiliente

Documento Técnico Completo v2.0

RESUMEN EJECUTIVO

Este documento presenta dos sistemas complementarios que abordan los problemas fundamentales de autoridad técnica y política en infraestructuras digitales:

1. **IA Verificable por Blockchain**: Una inteligencia artificial que no puede mentir sobre su proceso ni cambiar sin dejar rastro.
2. **DAO de Memoria Finita (FMD-DAO)**: Un sistema de gobernanza que equilibra experticia técnica y legitimidad democrática, evitando tanto la tecnocracia como la parálisis por consenso.

Ambos sistemas comparten un axioma termodinámico fundamental:

> **La memoria infinita es muerte. La amnesia total es caos.**

ÍNDICE

PARTE I: INTELIGENCIA ARTIFICIAL VERIFICABLE

1.1 Principio Fundamental

La IA no vive en la blockchain. Es verificada por ella.

- La inferencia ocurre off-chain (donde el cómputo es viable)
- Los procesos y resultados quedan anclados en blockchain
- Cualquiera puede verificar: qué versión del modelo, bajo qué reglas, con qué datos, si fue alterado

La blockchain no ejecuta. **Certifica.**

1.2 Qué Garantiza el Sistema

Garantías:

- **Integridad**: Los resultados no pueden alterarse silenciosamente
- **Transparencia**: Los cambios de reglas son observables

- **Trazabilidad**: Cada actor deja rastro criptográfico
- **Disputabilidad**: Cualquier inferencia puede ser desafiada y re-verificada

No garantiza:

- Corrección semántica del contenido
- Verdad absoluta
- Alineación ética perfecta
- Ausencia total de sesgos

1.3 Arquitectura Técnica

```
None
Usuario/Cliente
      ↓
Capa de Coordinación (off-chain)
      ↓
Nodo de Inferencia (off-chain, independiente)
      ↓
Capa de Verificación
      ↓
BLOCKCHAIN (registro inmutable)
      ↓
Respuesta Final + Prueba Verificable + Attestation
```

Compromisos On-Chain:

```
None
commitment = H(
  model_hash ||
  ruleset_hash ||
  prompt_hash ||
  output_hash ||
  execution_nonce ||
  timestamp ||
  operator_signature
)
```

Solo hashes, attestations y referencias se almacenan on-chain, nunca contenido completo.

1.4 Estrategias de Verificación

Fase 0 (6-8 semanas):

- Re-ejecución completa por auditores
- Modelo pequeño (≤100M parámetros)
- Inferencia determinista
- Challenge window: 24-72h

Fase 1 (12-16 semanas):

- Redundancia probabilística (k-of-n nodos)
- Pruebas ZK-ML parciales (opcional)
- Detección de divergencias
- Challenge window: 7 días

Fase 2 (6-12 meses):

- Mercado abierto de operadores
- Sistema de reputación no transferible
- Incentivos por honestidad
- Slashing por fraude probado

1.5 Gobernanza de la IA

Reglas inmutables:

- Públicas y versionadas
- Activadas con timelocks (no instantáneas)
- Cambios requieren evidencia pública

Memoria del modelo:

- $\tau_{IA} \approx$ 33-50 meses (memoria operacional larga)
- No aprende en producción
- Actualizaciones explícitas y versionadas

Ningún cambio ocurre sin dejar huella pública.

PARTE II: DAO DE MEMORIA FINITA (FMD-DAO)

2.1 Propósito Fundamental

La FMD-DAO se rige por una idea simple pero profunda:

> **Todo sistema que recuerde demasiado o demasiado poco colapsa.**

El sistema mantiene la oscilación estable dentro del **Valle de Resiliencia**:

```
None
R = τ × Ω

Donde:
- τ = memoria efectiva (ventana temporal de aprendizaje)
- Ω = frecuencia de revisión (adaptabilidad)
- Valle óptimo: 1 < R < 3
```

2.2 Principios Rectores

1. **Memoria Finita**: Toda información, reputación o autoridad decae con el tiempo
2. **Adaptación Periódica**: Normas se revisan según ciclos fijos, no impulsos emocionales
3. **Transparencia Métrica**: El índice R se calcula y publica constantemente
4. **Ruido Estabilizador**: Variabilidad controlada evita resonancias destructivas
5. **Gobernanza Bicameral**: Expertos y Comunes mantienen equilibrio cognitivo

2.3 Estructura Bicameral

Cámara 1: Expertos (C1)

Composición: 3-15 miembros seleccionados mediante Proof-of-Knowledge verificada on-chain

Función:

- Revisar y validar propuestas técnicamente
- Operar como filtro cognitivo (no como aristocracia)

Parámetros:

- τ_exp = 90 días (memoria de experticia)
- Ω_exp = 1/45 días (revisión cada 45 días)
- R_exp ≈ 2.25 (dentro del valle)

Incentivos:

- Remuneración proporcional a calidad de revisión
- Financiados por presupuesto votado por C2
- Reputación temporal con decay automático

Cámara 2: Comunes (C2)

Composición: Base amplia y abierta (mínimo 1 token o verificación civil)

Función:

- Iniciar y votar propuestas
- Definir presupuesto global
- Vetar decisiones de C1 con supermayoría (⅔)

Parámetros:

- τ_com = 60 días (ventana de decisión)
- Ω_com = 1/mes (frecuencia mensual)
- R_com ≈ 2.0 (sincronizado con C1)

Incentivos:

- Sin recompensas monetarias directas
- Única recompensa: voz y voto significativos
- Legitimidad democrática

2.4 Interacción Entre Cámaras

Flujo normal:

1. C2 propone y vota (quórum adaptativo)
2. Si aprueba → pasa a C1 para revisión técnica
3. C1 valida viabilidad e impacto
4. Si C1 aprueba → ejecución
5. Si C1 rechaza → devuelve con informe público

Ritual de Memoria Finita (cada 90 días):

- Archivar acuerdos caducos
- Redistribuir reputación y recursos
- Rebalancear τ y Ω según métricas del dashboard

Mecanismo de Re-equilibrio: Si |R_C1 - R_C2| > 1.5 → ajuste automático:

- Cámara más lenta acelera su revisión
- Cámara más rápida amplía su memoria
- Retorno dinámico al valle compartido

2.5 Relación de Memorias IA-DAO

```
None
Ratio: τ_IA / τ_DAO ≈ 200:1

Razón funcional:
- IA: sustrato lento (memoria operacional)
- DAO: dinámica rápida (memoria política)
```

Ratio adaptativo según polarización:

```
None
Si polarización > 70% → aumentar ratio (más elasticidad)
Si polarización < 30% → reducir ratio (más acoplamiento)
```

El ratio actúa como **amortiguador político**: cuando la comunidad se polariza, aumentar la distancia entre memorias fuerza al sistema a perder energía en la zona central, evitando oscilaciones destructivas.

PARTE III: SISTEMA INMUNOLÓGICO (RESPUESTA INFLAMATORIA)

3.1 Concepto

En crisis, el sistema puede activar una **respuesta inflamatoria** temporal: parámetros se ajustan para respuesta rápida, pero con decay forzoso y auditoría obligatoria.

No es dictadura. Es un **reflejo espinal con accountability brutal**.

3.2 Detección de Amenazas

Métricas monitoreadas:

```
None
- Colapso de participación (>70% caída en 48h)
- Fragmentación de consenso (Gini >0.8)
- Drenaje de tesorería (>30% salida en 7 días)
- Spike de reputación (1 actor >40% en 72h)
- Desviación de oráculos (>3σ)
- Falla técnica (exploit, halt)
```

Threat Score: Suma ponderada de métricas

Niveles de Severidad:

- 0-3: Normal
- 4-6: Alerta amarilla (observación aumentada)
- 7-9: Alerta naranja (restricciones leves)
- 10+: Alerta roja (inflamación total)

3.3 Velocidad de Amenaza (Factor Crítico)

No solo importa la magnitud, sino la derivada temporal:

```
None
Velocidad = dS/dt (puntos threat/hora)

Clasificación:
- LOGARÍTMICA: dS/dt < 2 (crecimiento lento)
- LINEAL: 2 ≤ dS/dt < 5 (moderado)
- EXPONENCIAL: dS/dt ≥ 5 (crisis explosiva)
```

Principio: La velocidad de amenaza determina el nivel de democracia permisible.

3.4 Respuesta Graduada

Crisis Exponencial (dS/dt > 5)

Ajustes paramétricos:

- Quórum: 40% (vs 10% normal)
- Timelock: 14 días (vs 48h normal)
- τ_DAO: 15 días (vs 60 días)
- τ_IA: 18 meses (vs 33 meses)
- Ratio τ_IA/τ_DAO: 432 (vs 200)

Pesos de cancelación de inflamación:

- C1 (Expertos): 60%
- Oráculo Verificado: 30%
- C2 (Comunes): 10% (testimonial, no tiebreak)

Razón: En crisis rápida, experticia técnica debe prevalecer sobre legitimidad democrática.

Crisis Logarítmica (dS/dt < 2)

Ajustes paramétricos:

- Parámetros casi normales
- Ligero aumento de vigilancia

Pesos de cancelación:

- C1: 33%
- Oráculo: 33%
- C2: 34% (democracia plena)

Razón: Hay tiempo para deliberación colectiva.

3.5 Tiebreaker en Zona Gris (50-67% score)

Si Crisis Exponencial:

```
C1 dividido (45-55%) → Oráculo decide
C1 >55% + Oráculo aprueba → Cancelar
C1 y Oráculo en desacuerdo → Mantener inflamación (conservador)

C2 NO participa en tiebreaker técnico
```

Si Crisis Logarítmica:

```
Votación ordinal entre C1, C2, Oráculo
Mayoría simple (2 de 3) decide

C2 participa plenamente
```

3.6 Garantías Anti-Dictadura

Decay Forzoso:

- Duración máxima: 30 días
- Decay lineal automático
- No perpetuable

Período Refractario:

- Mínimo 90 días entre activaciones
- Excepción: catástrofe (score ≥13)

Auditoría Post-Mortem (obligatoria):

- Registro on-chain de todas las decisiones
- Votación retroactiva (30 días post-resolución)
- Pregunta: ¿Fue la activación justificada?
- Si no → penalización severa a quienes decidieron

Penalizaciones por falsa alarma:

- Slash de reputación 30-40%
- Cooldown 180-270 días

- Ajuste de umbrales (más difícil próxima vez)

3.7 Propuestas Durante Inflamación

C2 SIEMPRE puede proponer (derecho inalienable)

Pero la ejecución depende de velocidad:

Crisis Exponencial:

- Revisión C1: OBLIGATORIA
- Revisión Oráculo: OBLIGATORIA
- Timelock mínimo: 12 horas
- C1 no puede ser bypasseado

Crisis Logarítmica:

- Proceso casi normal
- C2 puede saltar C1 con supermayoría (75%)

3.8 Cuarentena Post-Inflamación

Duración proporcional a severidad:

- Severity ≥12: 30 días
- Severity 8-11: 14 días
- Severity 4-7: 7 días
- Severity <4: 3 días

Durante cuarentena:

C2 puede proponer libremente, pero:

- Umbrales de aprobación más altos (50% → 60%)
- Timelocks extendidos (×2)
- C1 tiene veto temporal reforzado
- Propuestas críticas bloqueadas hasta fin de cuarentena

Decay lineal de restricciones: Las restricciones no son binarias. Decaen gradualmente del 100% al 0% durante el período de cuarentena.

Propuestas por nivel de riesgo:

Riesgo	C2 Threshold	C1 Review	Timelock	Estado
CRÍTICO	70%	Obligatorio	21d	Bloqueado
ALTO	65%	Obligatorio	14d	Permitido

MEDIO	55%	Opcional	7d	Permitido
BAJO	55%	Opcional	7d	Permitido

3.9 Compensación para C2

C2 pierde voto de cancelación en crisis rápida, pero gana:

1. **Derecho de auditoría inmediata:**
   - Puede exigir informe técnico en tiempo real
   - C1 debe explicar decisiones públicamente
   - Penalización si C1 no responde en 2h
2. **Veto post-facto reforzado:**
   - Con 75% (vs 67% normal) puede revocar decisión
   - Penalización doble a C1 (40% reputación vs 30%)
   - Cooldown más largo (270 días vs 180)
3. **Control presupuestario:**
   - C2 controla recursos asignados
   - C1 decide QUÉ hacer, no CUÁNTO gastar sin límite

PARTE IV: GESTIÓN DE SESGOS

4.1 Filosofía del Sesgo

Premisa: Es imposible eliminar sesgos, como en humanos o sistemas éticos.

Estrategia: Convertir el sesgo en sustrato evolutivo.

Principio: Sesgo evolutivo (útil) y sesgo de mala fe (extractivo) se tratan con la misma vara. No intentamos juzgar intenciones (imposible on-chain), solo consecuencias.

4.2 Ventana de Tolerancia

```
None
Sesgo ∈ [S_min, S_max]

Si S < S_min → sistema demasiado rígido
Si S > S_max → sistema capturado

Objetivo: mantener S en zona de fertilidad cognitiva
```

4.3 Depuración Pseudoexponencial

```
None
S(t) = S₀ × e^(-αt) + S_equilibrio

Características:
- Mejora visible en primeros 6-12 meses (exponencial)
- Asíntota a S_equilibrio ≠ 0 (nunca cero sesgo)
- α ajustable según tolerancia de inversores
```

Mensaje para inversores: El sistema mejora predeciblemente, pero nunca promete perfección (eso sería fraudulento).

4.4 Migración de Oráculos

Bootstrap del sistema:

```
None
Fase 1: Oráculos externos (Chainlink, API3, Band)
         Sistema aprende de fuentes establecidas

Fase 2: Oráculos híbridos (externos + consenso interno)
         Pesos se rebalancean según precisión

Fase 3: Oráculos internos mayoritarios
         Sistema auto-calibra, externos como check

Fase 4: Sistema como su propio oráculo
         Externos solo redundancia (5-10%)
```

Métrica de transición:

```
None
Migrar si:
  internal_accuracy > external_deviation ×
confidence_threshold
```

4.5 Diversidad Inicial (Siembra de Sesgos)

Para que la depuración tenga material sobre el que actuar, el sistema incluye diversidad intencional en fase génesis:

- Nodo A: optimista sobre nuevas propuestas
- Nodo B: pesimista/conservador
- Nodo C: enfocado en métricas cuantitativas
- Nodo D: enfocado en narrativas cualitativas

La tensión mutua crea el gradiente que permite depuración evolutiva.

4.6 Ruido Regenerativo

Para evitar convergencia excesiva (fragilidad ante shocks):

```
None
Cada k ciclos:
  - Seleccionar 10% de nodos
  - Introducir parámetro aleatorio acotado
  - Si mejora R → persiste
  - Si empeora → decae naturalmente
```

Es mutación genética: la mayoría es neutral o deletérea, pero ocasionalmente aparece ventaja adaptativa.

PARTE V: RESOLUCIÓN DE DISPUTAS (DISPUTE RESOLUTION)

5.1 Objetivo y Alcance

Objetivo: Que "verificable" signifique que si alguien miente sobre el proceso (modelo/reglas/ejecución), se puede probar on-chain y penalizar.

Qué se disputa:

- No la "verdad del texto" (imposible)
- Sino la **consistencia del proceso**:
    - ¿Se usó el modelo declarado?
    - ¿Se aplicaron las reglas correctas?
    - ¿El output corresponde al input + modelo + reglas?

5.2 Arquitectura de Attestation + Stake

Componentes:

1. **Attestation firmada:** Cada respuesta viene con attestation criptográfica del operador del nodo:

```
None
attestation = sign(
  operator_private_key,
  commitment_hash
)
```

2. **Stake bloqueado:** El nodo bloquea un bond (collateral) por:
- Período fijo (ej: 7 días), o
- Por request individual (más granular)
3. **Registro on-chain:**

solidity

```
None
struct Attestation {
    bytes32 commitmentHash;
    address operator;
    uint256 timestamp;
    uint256 stakeAmount;
    bytes signature;
    AttestationState state; // ACTIVE, CHALLENGED, SLASHED,
VALIDATED
}
```

5.3 Challenge Window

Toda attestation entra en estado "challengeable":

Fase	Challenge Window	Razón
Testnet	24-72 horas	Testing rápido
Mainnet	7 días	Dar tiempo a auditores
Producción madura	3 días	Balance velocidad/seguridad

Durante la ventana:

- Cualquiera puede desafiar
- Operador no puede retirar stake
- Output es "provisional" (usable pero no final)

Post-ventana:

- Si no hay disputa → attestation validada
- Operador puede retirar stake
- Output es "final"

5.4 Dispute Game

Fase 0: Re-ejecución determinista

El challenger presenta:

solidity

```
None
struct Challenge {
    bytes32 attestationId;
    bytes32 claimedCorrectOutput;
    uint256 challengeBond; // ej: 0.1 ETH
    string reason;
}
```

Juego de disputa:

1. Challenge se registra on-chain
2. Se seleccionan n "verifiers" registrados (ej: n=7)
3. Cada verifier re-ejecuta la inferencia:
    - Mismo model_hash
    - Mismo prompt_hash
    - Mismo seed
    - Mismo runtime_hash
4. Si k-of-n (ej: 5-of-7) coinciden con challenger → fraude probado
5. Si k-of-n coinciden con operador → challenger pierde

Fase 1: k-of-n recompute

solidity

```
None
function resolveDispute(bytes32 challengeId) public returns
(DisputeResult) {
    Challenge memory c = challenges[challengeId];
    Attestation memory a = attestations[c.attestationId];

    // Seleccionar n verifiers aleatoriamente
    address[] memory verifiers = selectVerifiers(challengeId,
7);
```

```
        // Solicitar re-ejecución
        bytes32[] memory outputs = new bytes32[](7);
        for (uint i = 0; i < verifiers.length; i++) {
            outputs[i] = IVerifier(verifiers[i]).recompute(
                a.commitmentHash
            );
        }

        // Contar coincidencias
        uint matchesOperator = 0;
        uint matchesChallenger = 0;

        for (uint i = 0; i < outputs.length; i++) {
            if (outputs[i] == a.outputHash) matchesOperator++;
            if (outputs[i] == c.claimedCorrectOutput)
matchesChallenger++;
        }

        // Decisión por mayoría (k=5 de n=7)
        if (matchesChallenger >= 5) {
            return DisputeResult.CHALLENGER_WINS;
        } else if (matchesOperator >= 5) {
            return DisputeResult.OPERATOR_WINS;
        } else {
            return DisputeResult.INCONCLUSIVE; // arbitraje C1
        }
    }
```

Fase 2: ZK-ML proofs (futuro)

Para modelos más grandes, verificación de subgrafos con pruebas de conocimiento cero:

- Verificar capas críticas del modelo
- Pruebas estadísticas de correctitud
- Sin re-ejecutar el modelo completo

5.5 Slashing Rules

Gradación por severidad:

solidity

```
None
enum FraudSeverity {
    MINOR,      // divergencia <5%
    MAJOR,      // divergencia >20% o hash incorrecto
    CRITICAL    // modelo cambiado, reglas violadas
}

function calculateSlashing(FraudSeverity severity) public pure
returns (
    uint slashPercent,
    uint cooldownDays,
    bool permanentBan
) {
    if (severity == FraudSeverity.MINOR) {
        return (10, 0, false);      // slash 10%, sin cooldown
    } else if (severity == FraudSeverity.MAJOR) {
        return (50, 180, false);    // slash 50%, 180d cooldown
    } else { // CRITICAL
        return (100, 0, true);      // slash 100%, ban
permanente
    }
}

function executeSlashing(
    address operator,
    uint slashPercent,
    uint cooldownDays,
    bool permanentBan
) internal {
    uint slashAmount = operatorStakes[operator] * slashPercent
/ 100;

    // Slash stake
    operatorStakes[operator] -= slashAmount;

    // Distribuir: 60% al challenger, 40% a treasury
    uint rewardChallenger = slashAmount * 60 / 100;
    uint toTreasury = slashAmount - rewardChallenger;
```

```
    // Aplicar cooldown o ban
    if (permanentBan) {
        bannedOperators[operator] = true;
    } else if (cooldownDays > 0) {
        operatorCooldown[operator] = block.timestamp +
(cooldownDays * 1 days);
    }

    emit OperatorSlashed(operator, slashAmount, cooldownDays,
permanentBan);
}
```

Si el challenger falla:

solidity

```
None
function slashChallenger(address challenger, uint bondAmount)
internal {
    // Quemar bond del challenger (anti-spam)
    // 50% quemado, 50% al operador como compensación

    uint burned = bondAmount / 2;
    uint compensation = bondAmount - burned;

    // Burn
    payable(address(0)).transfer(burned);

    // Compensar operador
    payable(operator).transfer(compensation);

    emit ChallengerSlashed(challenger, bondAmount);
}
```

5.6 Economía de la Disputa

El diseño económico debe cumplir:

```
Para detectores honestos:
  reward_esperada > costo_challenge

Para operadores honestos:
  costo_stake < ingresos_esperados

Para atacantes:
  costo_ataque > ganancia_ataque
```

Parámetros sugeridos:

Rol	Bond	Reward si gana	Pérdida si pierde
Operador	10 ETH	Fees por inference	10% - 100% stake
Challenger (Minor)	0.1 ETH	0.6 ETH	0.1 ETH
Challenger (Major)	1 ETH	5 ETH	1 ETH
Challenger (Critical)	5 ETH	10 ETH	5 ETH

Cálculo de incentivos:

```
P(fraude) = probabilidad de que attestation sea fraudulenta
P(detectar) = probabilidad de que challenger detecte fraude

Expected Value para challenger:
  EV = P(fraude) × P(detectar) × reward - (1 - P(fraude) ×
P(detectar)) × bond

Para que sea rentable desafiar:
  EV > 0

Esto implica:
  reward > bond / (P(fraude) × P(detectar))
```

5.7 Verifiers Rotativos

Problema: Si siempre se usan los mismos verifiers para k-of-n, pueden coludir.

Solución: Selección pseudoaleatoria por disputa.

solidity

```
None
function selectVerifiers(
    bytes32 disputeId,
    uint n
) internal view returns (address[] memory) {
    // Pool de verifiers elegibles
    address[] memory pool = getEligibleVerifiers();
    require(pool.length >= n, "Insufficient verifiers");

    // Seed pseudoaleatorio: disputeId + block.prevrandao
    bytes32 seed = keccak256(abi.encodePacked(
        disputeId,
        block.prevrandao,
        block.timestamp
    ));

    // Selección Fisher-Yates con seed
    address[] memory selected = new address[](n);
    uint poolSize = pool.length;

    for (uint i = 0; i < n; i++) {
        // Índice pseudoaleatorio
        uint index = uint(keccak256(abi.encodePacked(seed,
i))) % poolSize;

        selected[i] = pool[index];

        // Remover seleccionado del pool (sin reemplazo)
        pool[index] = pool[poolSize - 1];
        poolSize--;
    }

    return selected;
}

function getEligibleVerifiers() internal view returns
(address[] memory) {
    // Verifiers deben cumplir:
    // 1. Reputación > umbral
```

```
        // 2. No involucrados en la disputa
        // 3. No en cooldown
        // 4. Stake bloqueado suficiente

        address[] memory eligible = new
    address[](registeredVerifiers.length);
        uint count = 0;

        for (uint i = 0; i < registeredVerifiers.length; i++) {
            address v = registeredVerifiers[i];

            if (verifierReputation[v] >= MIN_REPUTATION &&
                !isInCooldown(v) &&
                verifierStakes[v] >= MIN_VERIFIER_STAKE) {
                eligible[count] = v;
                count++;
            }
        }

        // Trim array
        address[] memory result = new address[](count);
        for (uint i = 0; i < count; i++) {
            result[i] = eligible[i];
        }

        return result;
    }
```

5.8 El Problema del Determinismo

Desafío: Fase 0 exige determinismo perfecto, pero:

- Hardware diferente → variaciones de punto flotante
- Versiones de runtime → comportamientos sutiles
- Cuantización → errores de redondeo

Solución pragmática: Tolerancia ε

solidity

```
None
uint constant EPSILON = 1e15; // 0.001 en unidades de 1e18

function compareOutputs(
    bytes32 output1,
    bytes32 output2
) internal pure returns (OutputComparison) {
    // Convertir hashes a valores numéricos (si aplicable)
    // O comparar directamente si son hashes de texto

    uint diff = abs(uint(output1) - uint(output2));

    if (diff == 0) {
        return OutputComparison.IDENTICAL;
    } else if (diff < EPSILON) {
        return OutputComparison.NEGLIGIBLE_DIFF; // considerado
idéntico
    } else if (diff < 2 * EPSILON) {
        return OutputComparison.GRAY_ZONE; // requiere
arbitraje C1
    } else {
        return OutputComparison.DIVERGENT; // fraude probado
    }
}
```

Manejo de zona gris:

solidity

```
None
function handleGrayZoneDispute(bytes32 challengeId) internal {
    // Si divergencia está en [ε, 2ε]:

    // 1. Notificar a C1 para arbitraje humano
    requestC1Arbitration(challengeId);

    // 2. Operador NO pierde stake inmediatamente
    // 3. Challenger NO pierde bond
    // 4. Ambos esperan decisión C1 (max 7 días)
```

```
    // 5. C1 decide:
    //    - Fraude → slash operador
    //    - Ruido aceptable → devolver bonds
    //    - Ambiguo → split stake 50/50
}
```

Reputación gradual:

Si un operador cae repetidamente en zona gris (aunque no sea fraude), su reputación decae:

solidity

```
None
function updateReputationGrayZone(address operator) internal {
    grayZoneCount[operator]++;

    if (grayZoneCount[operator] > 10) {
        // Decay reputacional por inconsistencia
        reputation[operator] = reputation[operator] * 95 /
100;
    }

    // Reset cada 90 días
    if (block.timestamp > lastReset[operator] + 90 days) {
        grayZoneCount[operator] = 0;
    }
}
```

5.9 Implementación Técnica

Contratos principales:

solidity

```
None
// AttestationRegistry.sol
contract AttestationRegistry {
    mapping(bytes32 => Attestation) public attestations;
    mapping(address => uint256) public operatorStakes;
```

```
    function submitAttestation(
        bytes32 commitmentHash,
        bytes memory signature
    ) external payable returns (bytes32 attestationId);

    function challengeAttestation(
        bytes32 attestationId,
        bytes32 claimedCorrectOutput
    ) external payable returns (bytes32 challengeId);
}

// DisputeResolver.sol
contract DisputeResolver {
    function resolveDispute(bytes32 challengeId) external
returns (DisputeResult);
    function selectVerifiers(bytes32 disputeId, uint n)
internal returns (address[]);
    function executeSlashing(address operator, FraudSeverity
severity) internal;
}

// VerifierRegistry.sol
contract VerifierRegistry {
    mapping(address => bool) public registeredVerifiers;
    mapping(address => uint256) public verifierReputation;

    function registerAsVerifier() external payable;
    function submitVerification(bytes32 challengeId, bytes32
output) external;
}
```

Ejemplo de flujo completo:
```
1. Operador ejecuta inferencia
   → genera commitment_hash
   → firma attestation
   → bloquea 10 ETH stake
```

```
   → registra on-chain

2. Challenger detecta sospecha
   → paga 1 ETH bond
   → submite challenge con output correcto

3. Sistema selecciona 7 verifiers aleatoriamente
   → cada uno re-ejecuta
   → submiten sus outputs

4. 6 de 7 coinciden con challenger
   → fraude probado (MAJOR)

5. Slashing automático:
   → operador pierde 5 ETH (50% stake)
   → challenger recibe 3 ETH (60% de slash)
   → treasury recibe 2 ETH (40% de slash)
   → operador en cooldown 180 días

6. Auditoría post-mortem:
   → caso se archiva como precedente
   → umbrales se ajustan si necesario
```

PARTE VI: RESISTENCIA SYBIL

6.1 Criticidad del Problema

Resistencia Sybil es donde la mayoría de DAOs mueren.

Sin defensa adecuada:

- Un actor con capital crea 1000 identidades
- Captura C2 (comunes)
- Captura red de operadores
- Sistema colapsa

No existe solución perfecta. Se requiere diseño por capas con trade-offs explícitos.

6.2 Modelo S3: Stake + Decay + Caps (Fase 0)

Para C2 (Comunes):

solidity

```
None
contract C2GovernanceS3 {
    uint constant MIN_STAKE = 1 ether; // 1 token governance
    uint constant MAX_VOTING_POWER = 1000 ether; // cap por
dirección
    uint constant REPUTATION_DECAY_RATE = 2; // % por semana

    mapping(address => uint256) public votingPower;
    mapping(address => uint256) public lastActivity;

    function getEffectiveVotingPower(address voter) public
view returns (uint) {
        require(votingPower[voter] >= MIN_STAKE, "Insufficient
stake");

        // Cap aplicado
        uint capped = min(votingPower[voter],
MAX_VOTING_POWER);

        // Decay por inactividad
        uint weeksSinceActivity = (block.timestamp -
lastActivity[voter]) / 1 weeks;
        uint decayFactor = (100 - REPUTATION_DECAY_RATE *
weeksSinceActivity);

        if (decayFactor < 50) decayFactor = 50; // mínimo 50%

        return capped * decayFactor / 100;
    }

    function vote(uint proposalId, bool support) external {
        uint power = getEffectiveVotingPower(msg.sender);
        require(power > 0, "No voting power");

        // Registrar voto
        proposals[proposalId].votes[support] += power;
```

```
        // Actualizar actividad
        lastActivity[msg.sender] = block.timestamp;
    }
}
```

Ventajas:

- Implementable en 2 semanas
- Sin fricción de onboarding
- Testing rápido

Desventajas:

- Plutocracia mitigada pero presente
- 1 persona con 10M tokens puede crear 10k direcciones con 1k tokens cada una

Para Operadores de Inferencia:

solidity

```
None
contract OperatorRegistryS3 {
    uint constant MIN_OPERATOR_BOND = 10 ether; // 10 ETH
mainnet

    struct Operator {
        uint256 stake;
        uint256 reputation; // 0-100
        uint256 uptime; // % últimos 30d
        uint256 avgLatency; // ms
        bool active;
    }

    mapping(address => Operator) public operators;

    function registerOperator() external payable {
        require(msg.value >= MIN_OPERATOR_BOND, "Insufficient
bond");
        require(!operators[msg.sender].active, "Already
registered");
```

```
        operators[msg.sender] = Operator({
            stake: msg.value,
            reputation: 50, // inicial neutral
            uptime: 100,
            avgLatency: 0,
            active: true
        });
    }

    function getAssignmentRate(address operator) public view
returns (uint) {
        Operator memory op = operators[operator];
        require(op.active, "Not active");

        // Fórmula: 40% uptime + 30% honestidad + 20% latencia +
10% stake
        uint uptimeScore = min(op.uptime, 98) * 40 / 98;
        uint honestyScore = (op.reputation * op.reputation) *
30 / 10000;
        uint latencyScore = (1000 - min(op.avgLatency, 1000))
* 20 / 1000;
        uint stakeScore = log2(op.stake / MIN_OPERATOR_BOND) *
10;

        return uptimeScore + honestyScore + latencyScore +
stakeScore;
    }
}
```

Rendimientos decrecientes en stake:
```
log2(stake / min_stake) × 10

Ejemplos:
- 10 ETH → log2(1) × 10 = 0 (mínimo)
- 20 ETH → log2(2) × 10 = 10
- 40 ETH → log2(4) × 10 = 20
- 100 ETH → log2(10) × 10 = 33
```

Razón: Quien pone 100× el stake no debe ganar 100× los jobs. Previene plutocracia de nodos.

6.3 Modelo S2: Quadratic + Identidad (Fase 1)

6-12 meses después de lanzamiento:

solidity

```
None
contract C2GovernanceS2 {
    // Poder de voto = sqrt(tokens) × identity_multiplier

    enum IdentityTier {
        NONE,           // 1.0x
        GITHUB,         // 1.2x (GitHub verificado)
        PROOF_HUMANITY, // 1.5x (Proof of Humanity)
        MULTI_PROOF     // 2.0x (múltiples pruebas)
    }

    mapping(address => IdentityTier) public identityTier;
    mapping(address => uint256) public tokenBalance;

    function getVotingPower(address voter) public view returns
(uint) {
        uint sqrtTokens = sqrt(tokenBalance[voter]);
        uint multiplier =
getIdentityMultiplier(identityTier[voter]);

        return sqrtTokens * multiplier / 100;
    }

    function getIdentityMultiplier(IdentityTier tier) internal
pure returns (uint) {
        if (tier == IdentityTier.NONE) return 100;
        if (tier == IdentityTier.GITHUB) return 120;
        if (tier == IdentityTier.PROOF_HUMANITY) return 150;
        return 200; // MULTI_PROOF, cap
    }

    function verifyIdentity(IdentityTier tier, bytes memory
proof) external {
```

```
        // Verificar proof según tier
        // GitHub: OAuth + firma
        // PoH: verificar registro en Proof of Humanity
        // Multi: requerir 2+ pruebas

        require(validateProof(tier, proof), "Invalid proof");

        // Actualizar tier (solo puede subir, no bajar)
        if (tier > identityTier[msg.sender]) {
            identityTier[msg.sender] = tier;
            emit IdentityUpgraded(msg.sender, tier);
        }
    }
}
```

Ventajas:
- Onboarding gradual (no exige PoP desde día 1)
- Incentivo económico a verificarse
- Reduce plutocracia sin excluir

Ejemplo:
```
Usuario A: 10,000 tokens, sin identidad
  → sqrt(10000) × 1.0 = 100 poder de voto

Usuario B: 1,000 tokens, Proof of Humanity
  → sqrt(1000) × 1.5 = 47.4 poder de voto

Usuario C: 100,000 tokens, sin identidad
  → sqrt(100000) × 1.0 = 316 poder de voto

Usuario D: 4,000 tokens, Multi-proof
  → sqrt(4000) × 2.0 = 126 poder de voto
```

Usuario A tiene 100× más tokens que B, pero solo 2.1× más poder de voto.

6.4 Modelo S1: One-Human-One-Vote (Fase 2)

12-24 meses: madurez del sistema

solidity

```
None
contract C2GovernanceS1 {
    // Integración con Proof-of-Personhood

    IWorldID public worldID;
    IBrightID public brightID;
    IProofOfHumanity public proofOfHumanity;

    mapping(address => bool) public verified;
    mapping(uint256 => bool) public usedNullifiers; // evitar
doble voto

    function verifyHuman(
        bytes memory proof,
        uint256 nullifierHash
    ) external {
        require(!usedNullifiers[nullifierHash], "Already
verified");

        // Verificar con World ID o PoH
        bool valid = worldID.verifyProof(
            proof,
            nullifierHash,
            msg.sender
        );

        require(valid, "Invalid humanity proof");

        verified[msg.sender] = true;
        usedNullifiers[nullifierHash] = true;

        emit HumanVerified(msg.sender);
    }

    function vote(uint proposalId, bool support) external {
        require(verified[msg.sender], "Not verified human");

        // 1 human = 1 vote (igualitario)
        proposals[proposalId].votes[support] += 1;
```

```
        emit VoteCast(msg.sender, proposalId, support);
    }
}
```

Importante: C1 (Expertos) sigue siendo stake-weighted.

Racionalidad:
- C2 debe ser igualitario → legitimidad democrática
- C1 debe ponderar expertise → eficacia técnica

6.5 Sybil en Operadores de Inferencia

Mecanismos de defensa:

1. **Bond alto:**
```
Mainnet: 10 ETH mínimo
Crear 100 nodos → 1000 ETH bloqueados
```

2. **Slashing por fraude:**
```
Fraude probado → 50-100% stake perdido
Incentivo: honestidad > ganancia por fraude
```

3. **Reputación no transferible:**

solidity

None

```
// Reputación NO puede ser vendida o transferida
function transferReputation(address from, address to, uint
amount) external {
    revert("Reputation is non-transferable");
}

// Reputación solo se gana por comportamiento
```

```
function earnReputation(address operator) internal {
    // +1 por cada 100 inferencias honestas
    // -10 por cada disputa perdida
}
```

4. **Tasa de asignación proporcional:**
```
assignment_rate = f(uptime, honesty, latency, stake)

Con rendimientos decrecientes en stake:
- 10 ETH → rate base
- 100 ETH → rate 1.5×, no 10×
- 1000 ETH → rate 2×, no 100×
```

5. **Diversificación forzada:**

solidity

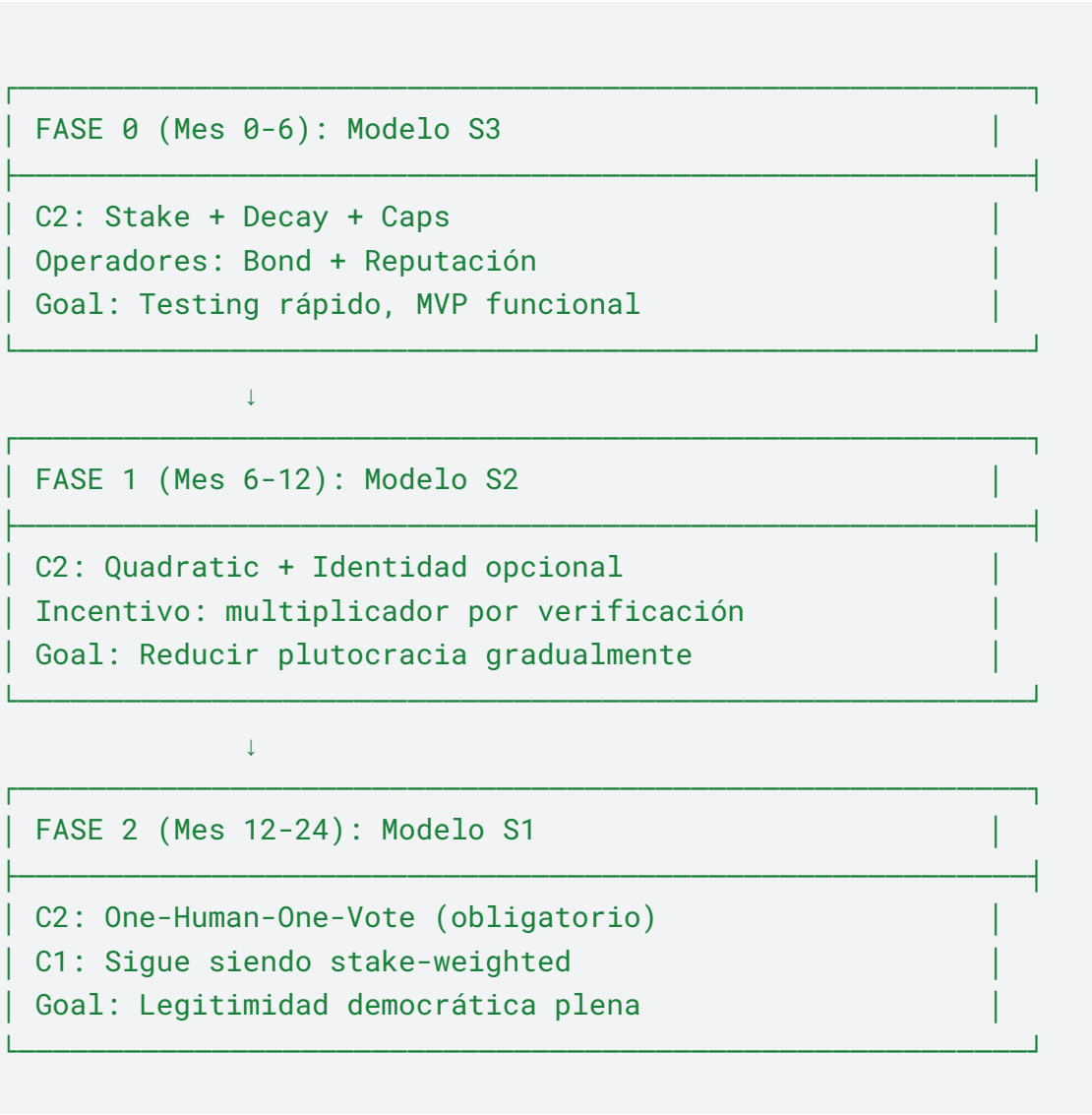
None
```
// No asignar >10% de jobs a un solo operador
function assignJob() internal returns (address operator) {
    address[] memory eligible = getEligibleOperators();

    for (uint i = 0; i < eligible.length; i++) {
        address op = eligible[i];

        // Verificar que no tiene >10% de jobs activos
        if (operatorActiveJobs[op] < totalActiveJobs / 10) {
            return op;
        }
    }

    revert("No eligible operators");
}
```

6.6 Roadmap de Transición
```
```



Criterios de transición:

De S3 a S2:

- ✓ >500 usuarios activos
- ✓ Sistema estable por 90 días
- ✓ Integración con providers de identidad lista

De S2 a S1:

- ✓ >2000 usuarios activos
- ✓ >60% usuarios verificaron identidad voluntariamente
- ✓ Votación C2 aprueba transición (67%)

6.7 Implementación Técnica

Contratos modulares:

solidity

```
None
// ISybilResistance.sol (interface)
interface ISybilResistance {
    function getVotingPower(address voter) external view
returns (uint);
    function isEligible(address voter) external view returns
(bool);
}

// SybilResistanceS3.sol
contract SybilResistanceS3 is ISybilResistance {
    // Implementación Stake + Decay + Caps
}

// SybilResistanceS2.sol
contract SybilResistanceS2 is ISybilResistance {
    // Implementación Quadratic + Identity
}

// SybilResistanceS1.sol
contract SybilResistanceS1 is ISybilResistance {
    // Implementación One-Human-One-Vote
}

// GovernanceCore.sol
contract GovernanceCore {
    ISybilResistance public sybilModule;

    // Permite cambiar módulo sin redeploy completo
    function upgradeSybilModule(address newModule) external
onlyGovernance {
        require(validateModule(newModule), "Invalid module");
        sybilModule = ISybilResistance(newModule);
        emit SybilModuleUpgraded(newModule);
    }
}
```

Testing comparativo:

solidity

None

```
// TestSybilAttack.sol
contract TestSybilAttack {
    function simulateAttack(
        ISybilResistance module,
        uint attackerCapital,
        uint numSybils
    ) public returns (
        uint capturePercentage,
        uint costPerVote
    ) {
        // Simular creación de sybils
        uint capitalPerSybil = attackerCapital / numSybils;
        uint totalVotingPower = 0;

        for (uint i = 0; i < numSybils; i++) {
            address sybil = address(uint160(i + 1000));

            // Dar capital al sybil
            stakingToken.transfer(sybil, capitalPerSybil);

            // Medir poder de voto obtenido
            uint power = module.getVotingPower(sybil);
            totalVotingPower += power;
        }

        // Calcular % de captura vs total system
        uint systemTotalPower = getTotalSystemVotingPower();
        capturePercentage = totalVotingPower * 100 /
systemTotalPower;

        // Costo por unidad de voto
        costPerVote = attackerCapital / totalVotingPower;

        return (capturePercentage, costPerVote);
    }
}
```

Resultados esperados:

```
| Modelo | Capital Atacante | Sybils | Captura | Costo/Voto |
|--------|------------------|--------|---------|------------|
| S3 | 1M tokens | 1000 | 35% | 28.5 tokens |
| S2 | 1M tokens | 1000 | 18% | 55.5 tokens |
| S1 | 1M tokens | 1000 | <1% | N/A (requiere humanos) |

---

## PARTE VII: MEDICIÓN EXACTA DE MÉTRICAS

### 7.1 Regla de Oro

**Toda métrica que activa poder excepcional (inflamación)
debe:**
1. Tener definición matemática exacta
2. Fuente de datos explícita
3. Resistencia razonable a manipulación
4. Registro on-chain del input usado (hash o snapshot)

**Principio:** Si no puedes medirlo objetivamente, no puede
triggear autoridad.

### 7.2 Participation Collapse

**Definición:**
```
participation_rate = unique_voters / eligible_voters

Collapse si:
  current_rate < median(historical_rates) × (1 - threshold)

Donde threshold = 0.7 (caída >70%)
```

**Implementación:**

solidity
```

None

```
contract ParticipationMonitor {
    struct EpochData {
        uint timestamp;
        uint uniqueVoters;
        uint eligibleVoters;
        uint participationRate; // en basis points (10000 =
100%)
    }

    uint constant HISTORY_WINDOW = 4; // últimas 4 semanas
    uint constant COLLAPSE_THRESHOLD = 7000; // 70%

    EpochData[HISTORY_WINDOW] public history;
    uint public currentEpoch;

    function detectParticipationCollapse() public view returns
(bool) {
        require(currentEpoch >= HISTORY_WINDOW, "Insufficient
history");

        // Calcular mediana histórica
        uint[] memory rates = new uint[](HISTORY_WINDOW);
        for (uint i = 0; i < HISTORY_WINDOW; i++) {
            rates[i] = history[i].participationRate;
        }
        uint medianRate = calculateMedian(rates);

        // Participación actual
        uint currentRate = getCurrentParticipationRate();

        // ¿Caída >70% vs mediana?
        bool collapsed = currentRate < medianRate * (10000 -
COLLAPSE_THRESHOLD) / 10000;

        // Protección: no triggear si eligibleVoters creció
súbitamente
        // (airdrop masivo no debe causar falso positivo)
        uint medianEligible =
calculateMedian(getHistoricalEligible());
```

```
        uint currentEligible = getEligibleVoters();

        if (currentEligible > medianEligible * 2) {
            // Ajustar por dilución
            uint dilutionFactor = (currentEligible * 10000) /
medianEligible;
            collapsed = currentRate < (medianRate * 10000) /
dilutionFactor;
        }

        return collapsed;
    }

    function calculateMedian(uint[] memory values) internal
pure returns (uint) {
        // Ordenar array
        uint[] memory sorted = sort(values);
        uint len = sorted.length;

        if (len % 2 == 0) {
            return (sorted[len/2 - 1] + sorted[len/2]) / 2;
        } else {
            return sorted[len/2];
        }
    }

    function recordEpoch() external {
        require(block.timestamp >= getLastEpochTime() + 1
weeks, "Too soon");

        uint voters = countUniqueVoters(7 days);
        uint eligible = getEligibleVoters();
        uint rate = (voters * 10000) / eligible;

        // Rotar historial (FIFO)
        for (uint i = 0; i < HISTORY_WINDOW - 1; i++) {
            history[i] = history[i + 1];
        }
```

```
        history[HISTORY_WINDOW - 1] = EpochData({
            timestamp: block.timestamp,
            uniqueVoters: voters,
            eligibleVoters: eligible,
            participationRate: rate
        });

        currentEpoch++;

        emit EpochRecorded(currentEpoch, rate);
    }

    function countUniqueVoters(uint timeWindow) internal view
returns (uint) {
        // Contar direcciones únicas que votaron
        // NO contar direcciones creadas hace <7 días
(anti-sybil)

        uint count = 0;
        uint cutoffTime = block.timestamp - timeWindow;
        uint minAccountAge = 7 days;

        for (uint i = 0; i < voters.length; i++) {
            address voter = voters[i];
            uint lastVoteTime = lastVotes[voter];
            uint accountCreation = accountCreationTime[voter];

            if (lastVoteTime >= cutoffTime &&
                accountCreation <= block.timestamp -
minAccountAge) {
                count++;
            }
        }

        return count;
    }
}
```

```
**Protecciones anti-manipulación:**
1. Usa mediana (no promedio) → resistente a outliers
2. No cuenta votos de cuentas nuevas (<7 días)
3. Ajusta por dilución si hay airdrop masivo
4. Ignora epochs con eventos extraordinarios marcados

### 7.3 Consensus Fragmentation

**Preferencia: Entropía de Shannon sobre Gini**

**Razón:**
- Gini mide desigualdad (bueno para riqueza)
- Entropía mide dispersión/incertidumbre (mejor para consenso)

**Definición:**
```
H = -Σ p_i × log₂(p_i)

Donde p_i = fracción de votos en opción i

H_normalized = H / log₂(n_opciones)

Fragmentación:
  H_norm < 40% → consenso fuerte
  40% < H_norm < 70% → consenso débil
  H_norm > 70% → fragmentación (trigger amarillo)
  H_norm > 85% → fragmentación crítica (trigger rojo)
```

**Implementación:**

solidity

```
None
contract ConsensusMonitor {
    function calculateConsensusEntropy(uint proposalId) public
view returns (uint) {
        Proposal memory prop = proposals[proposalId];

        // Agrupar votos en buckets
```
```

```
        uint[] memory buckets = new uint[](3);
        buckets[0] = prop.votesYes;
        buckets[1] = prop.votesNo;
        buckets[2] = prop.votesAbstain;

        uint total = buckets[0] + buckets[1] + buckets[2];
        require(total > 0, "No votes");

        // Calcular H = -Σ p_i × log₂(p_i)
        uint entropy = 0;

        for (uint i = 0; i < buckets.length; i++) {
            if (buckets[i] > 0) {
                // p_i en basis points (10000 = 100%)
                uint p_i = (buckets[i] * 10000) / total;

                // log₂(p_i) usando cambio de base: log₂(x) =
ln(x) / ln(2)
                // Aproximación: usar lookup table o biblioteca
                uint log2_pi = log2(p_i);

                // -p_i × log₂(p_i)
                entropy += (p_i * log2_pi) / 10000;
            }
        }

        // Normalizar: H_norm = H / log₂(n)
        uint H_max = log2(buckets.length) * 10000;
        uint H_normalized = (entropy * 100) / H_max;

        return H_normalized; // retorna % (0-100)
    }

    function detectFragmentation() public view returns (bool,
uint8 severity) {
        uint[] memory entropies = new
uint[](recentProposals.length);

        // Calcular entropía de últimas N propuestas
```

```
        for (uint i = 0; i < recentProposals.length; i++) {
            entropies[i] =
calculateConsensusEntropy(recentProposals[i]);
        }

        // Mediana de entropías
        uint medianEntropy = calculateMedian(entropies);

        if (medianEntropy > 85) {
            return (true, 10); // crítico
        } else if (medianEntropy > 70) {
            return (true, 6); // moderado
        }

        return (false, 0);
    }

    // Lookup table para log₂ (optimización de gas)
    function log2(uint x) internal pure returns (uint) {
        require(x > 0, "log2(0) undefined");

        if (x >= 5000) return 0; // log₂(0.5) ≈ -1 → 0 en
nuestra escala
        if (x >= 2500) return 10000; // log₂(0.25) ≈ -2
        if (x >= 1250) return 15000; // log₂(0.125) ≈ -3

        // Para valores más precisos, usar biblioteca
logarítmica
        // o aproximación polinómica
    }
}
```

Ejemplo práctico:
```
Propuesta A: 90% YES, 8% NO, 2% ABSTAIN
  p_yes = 0.90, p_no = 0.08, p_abs = 0.02
  H = -(0.90×log₂(0.90) + 0.08×log₂(0.08) + 0.02×log₂(0.02))
  H ≈ 0.57 bits
```

```
  H_max = log₂(3) ≈ 1.58 bits
  H_norm = 0.57 / 1.58 = 36% → consenso fuerte ✓

Propuesta B: 35% YES, 33% NO, 32% ABSTAIN
  H ≈ 1.58 bits
  H_norm = 1.58 / 1.58 = 100% → fragmentación crítica ✗
```

7.4 Treasury Drain

Definición:
```
drain_ratio = anomalous_outflow_7d / balance_7d_ago

anomalous_outflow = total_outflow - legitimate_streaming

Thresholds:
  >50% → crítico (severity 10)
  >30% → severo (severity 7)
  >15% → moderado (severity 4)
```

Implementación:

solidity

None
```
contract TreasuryMonitor {
    struct StreamingPayment {
        address recipient;
        uint256 amountPerDay;
        uint256 startTime;
        uint256 endTime;
        bool active;
    }

    mapping(uint => StreamingPayment) public
streamingPayments;
    uint public streamingPaymentCount;
```

```
    function detectTreasuryDrain() public view returns (bool,
uint8 severity) {
        // Balance hace 7 días
        uint balance_7d_ago = getHistoricalBalance(7 days);

        // Outflow últimos 7 días
        uint total_outflow = calculateOutflow(7 days);

        // Streaming legítimo pre-aprobado
        uint legitimate_streaming =
calculateLegitimateStreaming(7 days);

        // Outflow anómalo
        uint anomalous = total_outflow > legitimate_streaming
            ? total_outflow - legitimate_streaming
            : 0;

        // Ratio de drenaje (en basis points)
        uint drain_ratio = (anomalous * 10000) /
balance_7d_ago;

        if (drain_ratio > 5000) return (true, 10); // >50%
crítico
        if (drain_ratio > 3000) return (true, 7);  // >30%
severo
        if (drain_ratio > 1500) return (true, 4);  // >15%
moderado

        return (false, 0);
    }

    function calculateOutflow(uint timeWindow) internal view
returns (uint) {
        uint total = 0;
        uint cutoffTime = block.timestamp - timeWindow;

        for (uint i = 0; i < treasuryTransactions.length; i++)
{
            Transaction memory tx = treasuryTransactions[i];
```

```
            if (tx.timestamp >= cutoffTime && tx.direction ==
Direction.OUT) {
                total += tx.amount;
            }
        }

        return total;
    }

    function calculateLegitimateStreaming(uint timeWindow)
internal view returns (uint) {
        uint total = 0;
        uint days = timeWindow / 1 days;

        for (uint i = 0; i < streamingPaymentCount; i++) {
            StreamingPayment memory sp = streamingPayments[i];

            if (sp.active &&
                block.timestamp >= sp.startTime &&
                block.timestamp <= sp.endTime) {
                // Calcular cuánto se ha pagado en este window
                uint daysActive = min(days, (block.timestamp -
sp.startTime) / 1 days);
                total += sp.amountPerDay * daysActive;
            }
        }

        return total;
    }

    function getHistoricalBalance(uint timeAgo) internal view
returns (uint) {
        // Leer de snapshots históricos
        uint targetTimestamp = block.timestamp - timeAgo;

        // Buscar snapshot más cercano
        for (uint i = balanceSnapshots.length; i > 0; i--) {
```

```
            if (balanceSnapshots[i-1].timestamp <=
targetTimestamp) {
                return balanceSnapshots[i-1].balance;
            }
        }

        revert("No historical data");
    }
}
```

Protección adicional (rate limits):

solidity

```
None
contract TreasuryRateLimits {
    uint constant MAX_SINGLE_TX = 10; // 10% treasury por tx
    uint constant MAX_DAILY = 20; // 20% treasury por día
    uint constant TIMELOCK_LARGE = 48 hours; // para >10%

    mapping(uint => uint) public dailySpent; // day => amount

    function transfer(address to, uint amount) external
onlyGovernance {
        uint treasuryBalance = address(this).balance;

        // Límite por transacción
        require(amount <= treasuryBalance * MAX_SINGLE_TX /
100,
                "Exceeds per-tx limit");

        // Límite diario
        uint today = block.timestamp / 1 days;
        uint todaySpent = dailySpent[today];
        require(todaySpent + amount <= treasuryBalance *
MAX_DAILY / 100,
                "Exceeds daily limit");

        // Timelock para grandes transferencias
```

```
        if (amount > treasuryBalance * MAX_SINGLE_TX / 100) {
            require(hasTimelockPassed(to, amount), "Timelock
active");
        }

        // Ejecutar
        payable(to).transfer(amount);
        dailySpent[today] += amount;

        emit TreasuryTransfer(to, amount);
    }
}
```

7.5 Reputation Spike

Dos métricas complementarias:

A) Concentración (Herfindahl-Hirschman Index)

Definición:
```
HHI = Σ (share_i)²

Donde share_i = reputation_i / total_reputation

HHI ∈ [0, 10000]
  HHI < 1500 → competitivo
  1500 < HHI < 2500 → moderadamente concentrado
  HHI > 2500 → concentración alta
  HHI > 4000 → concentración crítica
```

Implementación:

solidity

```
None
contract ReputationMonitor {
```

```
    function calculateReputationHHI() public view returns
(uint) {
        uint totalRep = getTotalReputation();
        require(totalRep > 0, "No reputation");

        uint HHI = 0;
        address[] memory topActors = getTopNActors(20); // top
20

        for (uint i = 0; i < topActors.length; i++) {
            address actor = topActors[i];

            // Share en basis points (10000 = 100%)
            uint share = (reputation[actor] * 10000) /
totalRep;

            // Sumar cuadrado
            HHI += (share * share) / 10000;
        }

        return HHI;
    }

    function detectConcentration() public view returns (bool,
uint8 severity) {
        uint HHI = calculateReputationHHI();

        if (HHI > 4000) return (true, 8);  // crítico
        if (HHI > 2500) return (true, 5);  // moderado

        return (false, 0);
    }
}
```

B) Delta Reputacional (spike detection)

Definición:
```
```

```
Para cada actor en top N:
  delta = rep_now - rep_72h_ago
  delta_pct = delta / total_reputation

Spike si:
  delta_pct > 40% en 72h
```

Implementación:

solidity

None

```
contract ReputationSpikeDetector {
    struct ReputationSnapshot {
        uint256 timestamp;
        mapping(address => uint256) reputation;
    }

    ReputationSnapshot[24] public snapshots; // 1 snapshot cada
3h × 24 = 72h
    uint public currentSnapshotIndex;

    function recordSnapshot() external {
        require(block.timestamp >= getLastSnapshotTime() + 3
hours, "Too soon");

        // Rotar snapshots (FIFO)
        currentSnapshotIndex = (currentSnapshotIndex + 1) %
24;

        ReputationSnapshot storage snap =
snapshots[currentSnapshotIndex];
        snap.timestamp = block.timestamp;

        // Guardar reputación de top actors
        address[] memory topActors = getTopNActors(50);
        for (uint i = 0; i < topActors.length; i++) {
            snap.reputation[topActors[i]] =
reputation[topActors[i]];
```

```
        }

        emit SnapshotRecorded(currentSnapshotIndex,
block.timestamp);
    }

    function detectReputationSpike() public view returns
(bool, address spikedActor) {
        uint totalRep = getTotalReputation();
        address[] memory topActors = getTopNActors(20);

        // Snapshot de hace ~72h
        uint oldSnapshotIndex = currentSnapshotIndex; // el más
viejo
        ReputationSnapshot storage oldSnap =
snapshots[oldSnapshotIndex];

        for (uint i = 0; i < topActors.length; i++) {
            address actor = topActors[i];

            uint rep_now = reputation[actor];
            uint rep_72h_ago = oldSnap.reputation[actor];

            // Delta absoluto
            uint delta = rep_now > rep_72h_ago ? rep_now -
rep_72h_ago : 0;

            // Delta como % del total
            uint delta_pct = (delta * 100) / totalRep;

            // Spike si ganó >40% de reputación total en 72h
            if (delta_pct > 40) {
                return (true, actor);
            }
        }

        return (false, address(0));
    }
```

```
    function getGrayZoneCount(address actor) public view
returns (uint) {
        // Cuenta cuántas veces cayó en "zona gris" (divergencia
pequeña)
        return grayZoneCount[actor];
    }

    function penalizeGrayZone(address operator) internal {
        grayZoneCount[operator]++;

        // Si supera umbral, decay reputacional
        if (grayZoneCount[operator] > 10) {
            reputation[operator] = reputation[operator] * 95 /
100;
            emit ReputationPenalty(operator, "Excessive gray
zone");
        }

        // Reset cada 90 días
        if (block.timestamp > lastGrayZoneReset[operator] + 90
days) {
            grayZoneCount[operator] = 0;
            lastGrayZoneReset[operator] = block.timestamp;
        }
    }
}
```

7.6 Oracle Deviation

Definición:
```
Valores de múltiples oráculos: v₁, v₂, ..., v

Mediana: M = median(v₁, ..., v )

MAD (Median Absolute Deviation):
  MAD = median(|v₁ - M|, |v₂ - M|, ..., |v  - M|)
```

```
Z-score robusto para cada oráculo:
  z_i = |v_i - M| / (MAD + ε)

Outlier si z_i > 3 (moderado) o z_i > 5 (crítico)
```

Implementación:

solidity

```
None
contract OracleMonitor {
    address[] public oracles;

    function detectOracleDeviation() public view returns (
        bool hasOutlier,
        uint8 severity,
        address[] memory outliers
    ) {
        require(oracles.length >= 3, "Need 3+ oracles");

        // Obtener valores de todos los oráculos
        uint[] memory values = new uint[](oracles.length);
        for (uint i = 0; i < oracles.length; i++) {
            values[i] = IOracle(oracles[i]).getValue();
        }

        // Calcular mediana
        uint median = calculateMedian(values);

        // Calcular MAD (Median Absolute Deviation)
        uint[] memory deviations = new uint[](values.length);
        for (uint i = 0; i < values.length; i++) {
            deviations[i] = abs(int(values[i]) - int(median));
        }
        uint MAD = calculateMedian(deviations);

        // Detectar outliers
        address[] memory detectedOutliers = new
address[](oracles.length);
```

```
        uint outlierCount = 0;
        uint maxZScore = 0;

        for (uint i = 0; i < values.length; i++) {
            // Z-score robusto
            uint z_score = deviations[i] * 1e18 / (MAD + 1);
// +1 previene /0

            if (z_score > 3 * 1e18) {
                detectedOutliers[outlierCount] = oracles[i];
                outlierCount++;

                if (z_score > maxZScore) maxZScore = z_score;

                emit OracleOutlier(oracles[i], values[i],
median, z_score);
            }
        }

        // Determinar severidad
        uint8 sev = 0;
        if (maxZScore > 5 * 1e18) {
            sev = 8; // crítico
        } else if (maxZScore > 3 * 1e18) {
            sev = 4; // moderado
        }

        // Trim array
        address[] memory result = new address[](outlierCount);
        for (uint i = 0; i < outlierCount; i++) {
            result[i] = detectedOutliers[i];
        }

        return (outlierCount > 0, sev, result);
    }

    function abs(int x) internal pure returns (uint) {
        return x >= 0 ? uint(x) : uint(-x);
    }
```

```
}
```

Manejo de cisne negro (todos los oráculos divergen):

solidity

```
None
function handleBlackSwan() internal {
    // Si TODOS los oráculos tienen z_score alto
    // → evento extraordinario, no fallo de oráculos

    uint highZScoreCount = 0;
    for (uint i = 0; i < oracles.length; i++) {
        if (calculateZScore(i) > 3 * 1e18) {
            highZScoreCount++;
        }
    }

    if (highZScoreCount == oracles.length) {
        // Cisne negro: todos divergen
        // Activar "oráculo de última instancia" (C1 manual)

        emit BlackSwanEvent("All oracles diverge - manual
intervention required");
        requestC1ManualOracle();

        return;
    }

    // Si solo algunos divergen → outliers normales
    // proceder con slashing/alertas
}
```

7.7 Diseño Anti-Manipulación

Principios generales:

1. **Usar medianas, no promedios:**

solidity

```
None
// Resistente a outliers
uint median = calculateMedian(values);

// vs promedio (vulnerable)
uint average = sum(values) / values.length; // ✗
```

2. **Ventanas móviles:**

solidity

```
None
// No medir un solo punto
uint current = getCurrentValue();

// Medir sobre ventana
uint[] memory window = getValuesInWindow(7 days);
uint median = calculateMedian(window);
```

3. **Rate limits:**

solidity

```
None
// Limitar acciones que afectan métricas
mapping(address => uint) public lastAction;

function vote() external {
    require(block.timestamp >= lastAction[msg.sender] + 1
hours,
            "Rate limited");

    lastAction[msg.sender] = block.timestamp;
    // ...
}
```

4. **Snapshots on-chain:**

solidity

```
None
struct MetricsSnapshot {
    bytes32 dataHash; // hash de los datos crudos
    uint256 timestamp;
    uint256 participationRate;
    uint256 consensusEntropy;
    uint256 treasuryBalance;
    uint256 reputationHHI;
}

MetricsSnapshot[] public snapshots;

function recordSnapshot() external {
    // Guardar snapshot cada epoch
    // Permite auditoría post-mortem
}
```

5. **Ignora cuentas nuevas:**

solidity

```
None
uint constant MIN_ACCOUNT_AGE = 7 days;

function isEligibleForMetrics(address account) internal view
returns (bool) {
    return accountCreationTime[account] <= block.timestamp -
MIN_ACCOUNT_AGE;
}
```

7.8 Implementación Técnica

Contrato integrador:

solidity

```
None
contract ThreatScoreCalculator {
    ParticipationMonitor public participationMonitor;
    ConsensusMonitor public consensusMonitor;
    TreasuryMonitor public treasuryMonitor;
```

```
    ReputationMonitor public reputationMonitor;
    OracleMonitor public oracleMonitor;

    struct ThreatScore {
        uint8 total;
        uint8 participation;
        uint8 consensus;
        uint8 treasury;
        uint8 reputation;
        uint8 oracle;
        uint8 technical;
    }

    function calculateThreatScore() public view returns
(ThreatScore memory) {
        ThreatScore memory score;

        // Participation (max 3 puntos)
        (bool pCollapse, ) =
participationMonitor.detectParticipationCollapse();
        if (pCollapse) score.participation = 3;

        // Consensus (max 2 puntos)
        (bool cFrag, uint8 cSev) =
consensusMonitor.detectFragmentation();
        if (cFrag) score.consensus = cSev >= 10 ? 2 : 1;

        // Treasury (max 3 puntos)
        (bool tDrain, uint8 tSev) =
treasuryMonitor.detectTreasuryDrain();
        if (tDrain) score.treasury = tSev >= 10 ? 3 : tSev >=
7 ? 2 : 1;

        // Reputation (max 2 puntos)
        (bool rSpike, ) =
reputationMonitor.detectReputationSpike();
        uint HHI = reputationMonitor.calculateReputationHHI();
        if (rSpike) score.reputation = 2;
        else if (HHI > 2500) score.reputation = 1;
```

```
        // Oracle (max 2 puntos)
        (bool oDev, uint8 oSev, ) =
oracleMonitor.detectOracleDeviation();
        if (oDev) score.oracle = oSev >= 8 ? 2 : 1;

        // Technical (max 4 puntos) - implementado en otro
contrato
        if (hasTechnicalFailure()) score.technical = 4;

        // Total
        score.total = score.participation + score.consensus +
score.treasury +
                      score.reputation + score.oracle +
score.technical;

        return score;
    }

    function getThreatLevel() public view returns
(ThreatLevel) {
        uint8 score = calculateThreatScore().total;

        if (score >= 10) return ThreatLevel.RED;
        if (score >= 7) return ThreatLevel.ORANGE;
        if (score >= 4) return ThreatLevel.YELLOW;
        return ThreatLevel.GREEN;
    }
}
```

Dashboard (off-chain, alimentado por eventos):

javascript

```
JavaScript
// Frontend dashboard
const metrics = {
  participation: {
    current: 67,      // %
```

```
  median: 82,       // %
  status: 'warning', // green/yellow/red
  trend: '↓ -15%'
},
consensus: {
  entropy: 45,      // %
  status: 'healthy',
  trend: '→ stable'
},
treasury: {
  balance: '1.2M',
  outflow7d: '80K',
  status: 'healthy',
  trend: '↑ +5%'
},
reputation: {
  HHI: 1823,
  topActor: 'Actor_A (18%)',
  status: 'healthy',
  trend: '→ stable'
},
oracles: {
  median: '$3,245',
  outliers: 0,
  status: 'healthy',
  trend: '↑ +2%'
},
threatScore: {
  total: 3,
  level: 'GREEN',
  components: {
    participation: 0,
    consensus: 0,
    treasury: 0,
    reputation: 0,
    oracle: 0,
    technical: 0
  }
}
```

```
};
```

PARTE VIII: POLÍTICA DE UPDATES

8.1 Infraestructura vs Producto

Decisión filosófica: Este sistema se posiciona como **infraestructura**, no producto.

Implicaciones:

- Updates deben ser lentos, auditables, reversibles
- Canary deployments obligatorios
- Backward compatibility prioritaria
- Transparencia total

Contraste con "producto":

Aspecto	Producto	Infraestructura
Velocidad de update	Rápida	Lenta
Rollback	Difícil	Obligatorio
Breaking changes	Tolerados	Inaceptables
Auditoría	Opcional	Obligatoria
Versioning	Semántico	Estricto

8.2 U1: Ruleset Updates (Política y Seguridad)

Cambios en las reglas de gobernanza, límites, umbrales.

Características:

- Timelock: 7-30 días (según impacto)
- Requiere C2 + revisión C1 obligatoria
- Reversible con "emergency rollback" (pero sujeto a post-mortem)
- Versionado semántico

Implementación:

solidity

None

```
contract RulesetGovernance {
    struct RulesetProposal {
        bytes32 newRulesetHash;
        string description;
        string semanticVersion; // "2.1.0"
        uint256 proposalTime;
        uint256 timelockEnd;
        bool approved;
        bool executed;
        ImpactLevel impact;
    }

    enum ImpactLevel {
        LOW,      // 7 días timelock
        MEDIUM,   // 14 días
        HIGH,     // 30 días
        CRITICAL  // 60 días + supermayoría
    }

    bytes32 public currentRulesetHash;
    bytes32 public previousRulesetHash; // para rollback
    string public currentVersion;

    mapping(uint => RulesetProposal) public proposals;
    uint public proposalCount;

    function proposeRulesetUpdate(
        bytes32 _rulesetHash,
        string memory _description,
        string memory _version,
        ImpactLevel _impact
    ) public onlyC2 {
        require(getCurrentThreatSpeed() !=
ThreatSpeed.EXPONENTIAL,
                "No ruleset changes during crisis");

        uint timelockDuration = getTimelockForImpact(_impact);

        RulesetProposal memory prop = RulesetProposal({
```

```
            newRulesetHash: _rulesetHash,
            description: _description,
            semanticVersion: _version,
            proposalTime: block.timestamp,
            timelockEnd: block.timestamp + timelockDuration,
            approved: false,
            executed: false,
            impact: _impact
        });

        proposals[proposalCount] = prop;
        proposalCount++;

        emit RulesetProposed(proposalCount - 1, _rulesetHash,
_version);
    }

    function getTimelockForImpact(ImpactLevel impact) internal
pure returns (uint) {
        if (impact == ImpactLevel.LOW) return 7 days;
        if (impact == ImpactLevel.MEDIUM) return 14 days;
        if (impact == ImpactLevel.HIGH) return 30 days;
        return 60 days; // CRITICAL
    }

    function approveRuleset(uint proposalId) public {
        RulesetProposal storage prop = proposals[proposalId];

        // Votación C2
        uint c2Support = getC2Approval(proposalId);
        uint c2Threshold = prop.impact == ImpactLevel.CRITICAL
? 75 : 60;
        require(c2Support >= c2Threshold, "Insufficient C2
support");

        // Revisión C1 (obligatoria)
        require(getC1Approval(proposalId) >= 50, "C1 review
required");
```

```
        prop.approved = true;

        emit RulesetApproved(proposalId);
    }

    function executeRuleset(uint proposalId) public {
        RulesetProposal storage prop = proposals[proposalId];
        require(prop.approved, "Not approved");
        require(block.timestamp >= prop.timelockEnd, "Timelock
active");
        require(!prop.executed, "Already executed");

        // Guardar anterior para rollback
        previousRulesetHash = currentRulesetHash;

        // Activar nuevo
        currentRulesetHash = prop.newRulesetHash;
        currentVersion = prop.semanticVersion;

        prop.executed = true;

        emit RulesetUpdated(prop.newRulesetHash,
prop.semanticVersion);
    }

    // Emergency rollback (solo durante inflamación)
    function emergencyRollback(string memory reason) public {
        require(
            currentSeverity >= 7, // inflamación activa
            "Only during inflammation"
        );
        require(
            msg.sender == C1_multisig || msg.sender ==
oracleAddress,
            "Not authorized"
        );
        require(previousRulesetHash != bytes32(0), "No
previous ruleset");
```

```
        // Rollback
        bytes32 temp = currentRulesetHash;
        currentRulesetHash = previousRulesetHash;
        previousRulesetHash = temp;

        emit EmergencyRollback(currentRulesetHash, reason);

        // OBLIGATORIO: post-mortem audit
        scheduleEmergencyAudit();
    }
}
```

8.3 U2: Model Updates (Pesos de la IA)

Cambios en el modelo de IA: pesos, arquitectura, cuantización, runtime.

Características:

- La propuesta **debe incluir**:
    - `model_hash` (pesos)
    - `arch_hash` (arquitectura / grafo)
    - `runtime_hash` (versiones exactas: framework, kernels, libc, GPU driver si aplica)
    - `quant_hash` (cuantización / precisión)
    - `training_recipe_hash` (procedimiento)
    - `eval_suite_hash` (suite de evaluación acordada)
- Evaluación por **verifiers** independientes (k-of-n).
- Rollout canary obligatorio:
    - 5% → 25% → 50% → 100%
- Rollback automático si métricas críticas caen.

Canary + Guardrails

Guardrails mínimos:

- No degradar `honesty_score` (disputes perdidas) por encima de X
- No incrementar `gray_zone_rate` por encima de Y
- No degradar latencia P95 por encima de Z
- No romper compatibilidad de reglas (ruleset)

Regla canónica:
Un modelo nuevo **solo "existe"** cuando pasa por canary + ventana de challenge.

Implementación (esqueleto)

```
struct ModelRelease {
  bytes32 modelHash;
  bytes32 archHash;
  bytes32 runtimeHash;
  bytes32 quantHash;
  bytes32 recipeHash;
  bytes32 evalSuiteHash;
  string  semanticVersion; // "3.0.0"
  uint256 proposedAt;
  uint256 timelockEnd;
  uint8   canaryStage; // 0,5,25,50,100
  bool    approvedC2;
  bool    reviewedC1;
  bool    executed;
}

function proposeModelRelease(ModelRelease memory r) external onlyC2
{ ... }
function reviewC1(uint id, bool ok, string memory reportHash)
external onlyC1 { ... }
function approveC2(uint id) external onlyC2 { ... } // supermayoría
si cambio mayor
function executeCanary(uint id, uint8 stage) external
onlyCoordinator { ... }
function autoRollback(uint id, string memory reason) external
onlyGuardian { ... }
```

8.4 U3: Dataset Updates (Datos y Memoria)

Aquí hay un punto clave que ya insinuaste: **la IA no aprende en producción**.

Principios:

- Sin “aprendizaje silencioso”.
- Dataset updates solo en **ciclos de release**.
- Cada release publica:
    - `dataset_manifest_hash`
    - `provenance_hash` (fuentes)
    - licencias
    - checks de contaminación

Separación obligatoria:

- **Dataset de entrenamiento** (offline, versionado)
- **Memoria operacional** (si existe, debe ser finita, auditable y opt-in)
- **Logs** (hashes / métricas, jamás contenido sensible)

Memoria operacional (si la usas)

Si quieres una “memoria” sin traicionar la tesis:

- memoria **local** al usuario (cifrada, controlada por el usuario)
- decay explícito (TTL)
- exportable / borrable (derecho de salida)

8.5 Version Compatibility

Regla: request = (model_version, ruleset_version)

Modos:

- `latest_stable`
- `pinned(vX.Y)`
- `compat(vX)` (cualquier patch de X)

Soporte temporal sugerido:

- `latest_stable`: siempre
- `pinned`: 6–12 meses
- EOL (end-of-life) anunciado con timelock.

Prohibición:

- Breaking changes sin:
    - timelock extendido
    - migrador automático
    - rollback posible

8.6 Implementación Técnica

Contratos / módulos mínimos en updates:

- `RulesetGovernance`
- `ModelReleaseRegistry`
- `DatasetManifestRegistry`
- `VersionRouter` (coordinador decide a qué versión enruta)

El coordinador **no decide por autoridad**, decide por reglas:

- stage canary
- compatibilidad
- score de salud

PARTE IX: IMPLEMENTACIÓN TÉCNICA GENERAL

9.1 Stack Tecnológico

On-chain

- EVM (por liquidez y tooling) o L2 (por costos)
- Contratos: governance, attestations, disputes, metrics snapshots, release registry

Off-chain

- Coordinador (stateless en lo posible, auditado por logs)
- Nodos de inferencia (operadores)
- Verifiers (recompute / auditoría)
- Dashboard de métricas (event-driven)

Cripto

- Firmas ECDSA
- Merkle trees para batch de eventos
- Commit-reveal para prompts sensibles (opcional)

9.2 Contratos Principales

Lista mínima:

1. `AttestationRegistry`
2. `DisputeResolver`
3. `VerifierRegistry`
4. `GovernanceCore` (C1/C2)
5. `ThreatScoreCalculator` + `MetricsSnapshot`
6. `RulesetGovernance`
7. `ModelReleaseRegistry`
8. `SybilResistanceModule` (upgradable)
9. `Treasury` + `RateLimits`

9.3 Costos Operacionales

Tres costos dominantes:

1. **Recompute** (verifiers)
2. **Infra de inferencia** (GPU)
3. **On-chain** (gas)

Reglas de eficiencia:

- On-chain solo hashes + eventos
- Snapshots por epoch (semanal/mensual)
- Disputes con bonds para filtrar spam

9.4 Roadmap de Desarrollo

Fase 0 (6–8 semanas)

- Attestations + stake
- Challenge window
- Recompute determinista
- MVP C1/C2 + snapshots básicos

Fase 1 (3–4 meses)

- k-of-n recompute
- threat score completo
- canary updates
- rate limits tesorería

Fase 2 (6–12 meses)

- mercado de operadores
- reputación robusta no transferible
- identidad opcional (S2)
- mejoras de verificación (parcial ZK opcional)

PARTE X: CASOS DE USO

1. **IA común para decisiones críticas**

- propuestas de presupuesto
- priorización de roadmap
- auditoría de reglas

2. **Infra de agentes verificables**

- agentes que ejecutan tareas y dejan huella
- cada acción tiene attestation + dispute path

3. **Oráculos híbridos**

- el sistema aprende a ponderar fuentes externas vs internas
- migración controlada

4. **Gobernanza resiliente en comunidades polarizadas**

- el sistema inmunológico evita captura y evita parálisis

PARTE XI: RIESGOS Y MITIGACIONES

R1: Colusión de verifiers

- rotación pseudoaleatoria
- stake de verifier
- penalización por outputs falsos
- diversidad geográfica/operacional recomendada

R2: Captura de C2 por capital (Sybil/plutocracia)

- S3 → S2 → S1
- caps + decay + quadratic
- identidad gradual

R3: Coordinador off-chain se vuelve punto único

- coordinadores múltiples
- logs comparables

- reglas de enrutamiento on-chain (router)
- fallback a modo degradado

R4: Determinismo imperfecto

- runtime hash estricto
- tolerancias ε + zona gris
- reputación penaliza inestabilidad aunque no sea fraude

R5: Overfitting de métricas (gaming)

- medianas
- ventanas móviles
- rate limits
- auditoría post-mortem

PARTE XII: MÉTRICAS DE ÉXITO

Métricas técnicas

- % attestations sin disputa (pero ojo: demasiado alto puede ser falta de vigilancia)
- tiempo promedio de resolución de disputa
- tasa de falsos positivos de challengers
- latencia P95 de inferencia

Métricas de gobernanza

- participación saludable (no solo alta)
- entropía/fragmentación dentro de rango

- ratio de propuestas ejecutadas vs propuestas aprobadas
- tiempo a decisión (no demasiado corto ni demasiado largo)

Métricas inmunológicas

- cantidad de inflamaciones por año
- % inflamaciones justificadas (post-mortem)
- severidad promedio
- recuperación (tiempo a normalidad)

PARTE XIII: LIMITACIONES Y NO-PROMESAS

Este documento se protege de la “venta de milagros”.

No promete:

- verdad semántica
- ausencia de sesgo
- ética universal
- inmunidad total a captura (solo resistencia)
- ZK-ML completo en fase temprana

Sí promete:

- cambios visibles y auditables
- disputa posible con consecuencias
- memoria finita como ley operativa
- resiliencia: el sistema puede degradarse sin colapsar

PARTE XIV: EPÍLOGO FILOSÓFICO

La tesis central no es tecnológica. Es termodinámica.

Un sistema con memoria infinita se petrifica: confunde historia con ley.
Un sistema sin memoria repite su trauma: confunde impulso con verdad.

La FMD-DAO impone una disciplina:

- recordar lo suficiente para aprender,
- olvidar lo suficiente para seguir vivo.

La IA verificable impone otra:

- poder sin huella es tiranía,
- cambio sin rastro es corrupción.

La meta no es perfección.
La meta es **corrección en movimiento**.

ANEXOS

A. Glosario mínimo

- Attestation, Commitment, Challenge Window, Dispute Game, Slashing, Canary, τ, Ω, R, Inflamación, Cuarentena

B. Parámetros sugeridos (tabla)

- timelocks por impacto
- bonds (operador/challenger/verifier)
- thresholds de threat score y dS/dt
- caps / decay rates

C. Pseudocódigo de router de versiones

- cómo enrutar `latest_stable` vs `pinned`

D. Plantilla de Post-Mortem

- qué pasó, por qué pasó, evidencia on-chain, quién decidió, qué se ajusta

ANEXO A — Glosario mínimo

Attestation
Declaración criptográfica firmada por un operador que afirma: “esta salida corresponde a este input bajo este modelo y estas reglas”. Incluye `commitment_hash`, firma, timestamp y stake asociado.

Commitment
Huella hash que “amarra” los componentes críticos de una ejecución sin publicar el contenido:
`H(model_hash || ruleset_hash || prompt_hash || output_hash || nonce || timestamp || operator_sig)`
Sirve como ancla inmutable para verificación y disputa.

Challenge Window
Ventana temporal en la que una attestation puede ser desafiada. Durante ese período el stake queda bloqueado y el resultado es “provisional” (usable pero no final). Tras la ventana, si no hay challenge, se considera final.

Dispute Game
Protocolo formal para resolver un desafío: selección de verifiers, re-ejecución/recompute (o verificación parcial), comparación de outputs con reglas (incluida zona gris), y decisión final con consecuencias económicas.

Slashing
Penalización automática (total o parcial) del stake o reputación al probar fraude, incumplimiento de reglas o comportamiento malicioso. Suele redistribuirse entre challenger y tesorería.

Canary (deployment)
Despliegue escalonado de un update (modelo/reglas/dataset) a un porcentaje pequeño de tráfico para observar métricas y detectar degradaciones antes del rollout total (ej.: 5%→25%→50%→100%).

τ (tau)
Memoria efectiva: ventana temporal sobre la cual el sistema “recuerda” (reputación, autoridad, datos, precedentes). Mayor τ = más estabilidad, menos adaptabilidad.

Ω (omega)
Frecuencia de revisión/adaptación: cada cuánto el sistema reevalúa reglas, reputación, parámetros y decisiones. Mayor Ω = más adaptabilidad, más riesgo de volatilidad.

R (Índice de Resiliencia)
Producto de memoria por frecuencia: `R = τ × Ω`. Controla el “régimen” del sistema:

- R muy bajo → caos (cambia demasiado rápido)

- R muy alto → rigidez (se congela)
- R en valle óptimo (p.ej. 1–3) → resiliencia

Inflamación
Modo excepcional temporal activado por amenaza alta y/o rápida (score y derivada). Ajusta quórum, timelocks, pesos y procedimientos para priorizar contención y seguridad con accountability fuerte.

Cuarentena
Período posterior a inflamación donde se suaviza el retorno a normalidad. Las restricciones decaen gradualmente; se endurecen umbrales y timelocks para evitar "recaídas" y exploit post-crisis.

ANEXO B — Parámetros sugeridos (tablas)

B1) Timelocks por impacto (U1 Ruleset / U2 Model / U3 Dataset)

Impacto	Ejemplos típicos	Timelock sugerido	Umbral C2	Revisión C1	Canary
LOW	UI/telemetría, límites no críticos	7 días	60%	obligatoria	opcional
MEDIUM	ajuste thresholds, cambios menores en proceso	14 días	60%	obligatoria	5→25→50→100
HIGH	cambios de incentivos, slashing, módulos clave	30 días	67%	obligatoria + reporte	5→25→50→100
CRITICAL	breaking change, seguridad/constitución del sistema	60 días	75%	obligatoria + auditoría externa	5→25→50→100 + freeze

Regla práctica: "Infraestructura no se actualiza rápido". Y menos si toca poder (slashing, sybil, treasury).

B2) Bonds / Staking (operador / challenger / verifier)

Rol	Depósito mínimo	Bloqueo	Reward si gana	Pérdida si pierde	Notas

Operador (Inference)	10 ETH	≥ Challenge Window	fees + reputación	slash 10–100% + cooldown	stake escalable con rendimientos decrecientes
Challenger (Minor)	0.1 ETH	hasta resolución	60% del slash minor	pierde bond	anti-spam
Challenger (Major)	1 ETH	hasta resolución	60% del slash major	pierde bond	para casos de alto impacto
Challenger (Critical)	5 ETH	hasta resolución	60% del slash critical	pierde bond	incentiva cazar fraudes graves
Verifier	2–5 ETH	por epoch	fee por verificación correcta	slash si miente/omite	stake de verifier evita colusión barata

Distribución estándar del slash: 60% challenger / 40% treasury (ajustable por gobernanza).

B3) Thresholds de threat score y dS/dt (velocidad)

Score total (ejemplo operativo)

Threat Score	Estado	Acción
0–3	GREEN	normal
4–6	YELLOW	vigilancia + restricciones leves
7–9	ORANGE	modo reforzado (timelocks↑, quórum↑)
≥10	RED	inflamación total

Velocidad (derivada)

dS/dt (puntos por hora)	Clasificación	Principio
<2	LOGARÍTMICA	hay tiempo para deliberación
2–5	LINEAL	ajustar umbrales moderadamente
≥5	EXPONENCIAL	priorizar contención técnica

Recomendación: usar ambas señales: `activar = f(score, dS/dt)` para evitar activaciones por ruido.

B4) Caps / Decay rates (anti-captura)

C2 (comunes) — Fase S3 (stake + decay + caps)

Parámetro	Sugerido	Razón
Min stake elegible	1 unidad	evita spam sin fricción fuerte
Cap por dirección (voting power)	equivalente a 1000 unidades	reduce plutocracia directa
Min edad de cuenta para métricas	7 días	reduce sybil reciente
Decay por inactividad	2%/semana (floor 50%)	evita “voto zombi” y poder fosilizado

Reputación (operadores/verifiers)

Parámetro	Sugerido	Razón
Reputación no transferible	rígido	evita mercados de reputación
Decay reputacional	1–3%/semana	memoria finita operativa
Reset de “zona gris”	cada 90 días	evita castigo infinito por ruido
Límite de jobs por operador	10% jobs activos	diversificación forzada

---

ANEXO C — Pseudocódigo del router de versiones (latest_stable vs pinned)

Objetivo: enrutar requests de forma **determinista y auditable**, sin “magia” del coordinador.

Modelo mental

- `latest_stable`: versión activa aprobada + completó canary + no está bajo rollback
- `pinned(vX.Y.Z)`: versión exacta solicitada (si está dentro de ventana de soporte)
- `compat(vX)`: acepta cualquier patch compatible (X.*) que sea estable

Pseudocódigo (estilo Solidity)

```
enum ReqMode { LATEST_STABLE, PINNED, COMPAT_MAJOR }
```

```
struct Version {
  string semver;           // "3.1.2"
  bytes32 modelHash;
  bytes32 rulesetHash;
  bytes32 datasetHash;
  bool isStable;           // true tras canary completo + challenge
window
  bool isDeprecated;       // anunciado EOL
  uint256 supportUntil;    // timestamp EOL
}

mapping(bytes32 => Version) public versionsById; // id = H(semver)
bytes32 public latestStableId;                   // puntero único
mapping(bytes32 => bytes32[]) public compatIndex; // major -> list
stable ids (opcional)

function routeRequest(
  ReqMode mode,
  string memory pinnedSemver,
  uint256 userFlags // opcional: "no canary", "prefer old", etc
) public view returns (Version memory v) {

  if (mode == ReqMode.LATEST_STABLE) {
    Version memory ls = versionsById[latestStableId];
    require(ls.isStable, "No stable version");
    return ls;
  }

  if (mode == ReqMode.PINNED) {
    bytes32 id = keccak256(bytes(pinnedSemver));
    Version memory pv = versionsById[id];
    require(pv.modelHash != bytes32(0), "Pinned not found");
    require(block.timestamp <= pv.supportUntil, "Pinned EOL");
    // permitir pinned aunque no sea latest, pero debe ser estable
    require(pv.isStable, "Pinned not stable");
    return pv;
  }

  // COMPAT_MAJOR (ej: "3.x")
  bytes32 major = extractMajorKey(pinnedSemver); // "3"
```

```
  bytes32[] memory candidates = compatIndex[major];

  // elegir el más nuevo que sea estable y soportado
  bytes32 best = bytes32(0);
  for (uint i = 0; i < candidates.length; i++) {
    Version memory c = versionsById[candidates[i]];
    if (c.isStable && block.timestamp <= c.supportUntil) {
      if (best == bytes32(0) || semverGreater(c.semver,
versionsById[best].semver)) {
        best = candidates[i];
      }
    }
  }
  require(best != bytes32(0), "No compat stable");
  return versionsById[best];
}
```

Reglas extra recomendadas:

- Si hay **inflamación**, el router puede forzar `latest_stable` (o `safe_mode`) para evitar pinned vulnerable.
- El router debe emitir eventos en cada cambio de `latestStableId`.

ANEXO D — Plantilla de Post-Mortem (obligatoria)

> Documento público, firmado y anclado on-chain (hash del PDF/Markdown).
> Longitud recomendada: 1–3 páginas. Sin narrativa épica; puro dato y decisiones.

1) Resumen

- **Evento:** (nombre corto)
- **Fechas:** inicio / detección / contención / cierre
- **Severidad:** score máximo y dS/dt máximo
- **Impacto:** participación, treasury, reputación, disponibilidad, usuarios afectados

2) ¿Qué pasó? (timeline)

Tabla breve:

- T0: primer síntoma (hash/evento)
- T1: trigger de alerta
- T2: decisión (inflamación / cuarentena / rollback)
- T3: mitigación aplicada
- T4: validación de normalidad

3) ¿Por qué pasó? (causa raíz)

- Causa técnica (si aplica)
- Causa económica/incentivos (si aplica)
- Causa de gobernanza/proceso (si aplica)
- **Qué supusimos mal** (supuestos rotos)

4) Evidencia on-chain (mínimo)

- Attestations relevantes (ids / hashes)
- Challenges / Disputes (ids / resultado)
- Snapshots de métricas (epoch ids)
- Transfers de tesorería (tx hashes)
- Cambios de ruleset / releases (proposal ids)
- Logs del router (cambio de latest stable)

5) ¿Quién decidió qué?

- ¿Se activó inflamación? ¿Quién autorizó? (C1 multisig/oráculo/C2 votación)
- Votos y thresholds (C1/C2/oráculo) + timestamps

- Justificación basada en métricas (no opiniones)

6) ¿Qué se hizo? (acciones)

- Contención inmediata (rate limits, freeze, rollback, canary revert)
- Mitigación de mediano plazo (patch, reglas, parámetros)
- Recuperación (cierre de inflamación, cuarentena, retorno gradual)

7) Evaluación: ¿fue justificado?

- ¿El trigger fue correcto o falso positivo?
- ¿Hubo sobre-reacción o reacción tardía?
- ¿Qué señales fueron engañosas?

8) Cambios que se ajustan (learning)

Checklist:

- Parámetros (thresholds, timelocks, caps, bonds)
- Procedimiento (quién puede qué en crisis)
- Observabilidad (nuevas métricas, mejor snapshot)
- Incentivos (ajuste de rewards/slash)
- Seguridad técnica (hardening, testing)

9) Acciones y responsables (accountability)

- Acción / dueño / deadline / criterio de cierre
- Si hay penalizaciones: aplicar y registrar (slash/cooldown)

10) Apéndice (opcional)

- Queries reproducibles

- Script de reconstrucción del threat score para el período
- Informe externo (si hubo)