EFM Codex — Appendix J

The Constitutional Kernel

Bounded Layer 6: Governing the Rules of Change

Entropica SPC — Yology Research Division

Version 1.7 — December 2025

The Highest Law

The Constitutional Kernel is the **supreme authority** of the EFM architecture. It governs not the rules of action, but the **rules of change**—how the system may evolve, and when it may not.
All other layers (0–5) operate *under* Constitutional authority. The Kernel itself may only be modified through cryptographically bounded procedures with multi-party consent.

Volume Dependencies

This appendix assumes familiarity with:

- **Volume I** — Layer 0 (Vault), Layer 0.5 (Reflex-Core)
- **Volume II** — Layers 1–5, Arbiter Layer, Forest Layer
- **Appendix E** — ZK-SP (Zero-Knowledge Safety Proofs)
- **Appendix F** — Reflex Escalation (Level 5–6 authority)
- **Appendix G** — Gardener Interface (Constitutional Officer)
- **Appendix I** — Deployment Profiles (Constitutional Access)

Contents

1 Overview and Purpose

1.1 Bridging Summary

Appendix J defines the **Constitutional Kernel**—the meta-layer (Layer 6) that governs recursive self-modification, identity integrity, and the immutable boundaries of capsule existence. The Kernel enforces the **rules of change**, not the rules of action.

Core Principle: The Constitutional Kernel enables *lawful recursion*—structured self-improvement within hard-coded invariants. Capsules may evolve, but they may not violate their constitutional foundations.

1.2 Design Goals

1. Prevent unauthorized evolution of safety, legal, or ethical constraints
2. Enable structured self-improvement within cryptographically enforced bounds
3. Provide forensic trace hooks for constitutional arbitration and rollback
4. Integrate with Gardener authority (Appendix G) for human oversight
5. Maintain compatibility with deployment profiles (Appendix I)

1.3 Layer 6 in the EFM Hierarchy

Table 1: Complete EFM layer hierarchy.

Layer	Name	Function
6	**Constitutional Kernel**	**Governs rules of change (Layers 1–5)**
5*	Swarm Governance	Multi-capsule coordination
4*	Ethical Reasoning	Value alignment, moral deliberation
3	Strategic Planning	Long-horizon decision making
2	Tactical Execution	Short-term action selection
1	Sensory Processing	Input interpretation
0.5	Reflex-Core	Fast safety responses (**constrains Layer 6**)
0	**Vault**	**Immutable physics (constrains ALL layers)**

* **Forward Reference:** Layers 4–5 are reserved for future specification. They are included in this hierarchy to establish Layer 6's governance scope. Current Codex (Volumes I–II) specifies Layers 0–3 only. Implementations MAY define custom Layer 4–5 logic under Layer 6 governance.

1.4 The Subordination Doctrine

Layer 0 Governs Layer 6, Not Vice Versa

The Constitutional Kernel is **not** the "topmost" layer in the sense of authority. It is the topmost layer that may be *modified*. Layer 0 (Vault) is the **absolute foundation** that constrains even Layer 6.

The Hierarchy of Constraints:

- **Layer 0** constrains **Layers 0.5–6** (immutable physics)
- **Layer 0.5** constrains **Layers 1–6** (safety boundaries)
- **Layer 6** governs **Layers 1–5** (constitutional law)

The Metaphor: Layer 0 is *Biology*—you need oxygen. Layer 6 is *Law*—you pay taxes. You can change the Law (amend Layer 6), but you cannot legislate away the need for oxygen (Layer 0 is immutable).

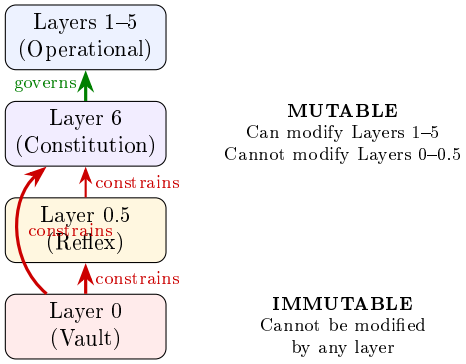

Figure 1: Subordination Doctrine: Layer 0 constrains Layer 6, not vice versa.

Layer 6 Authority (Bounded)

Layer 6 is authorized to modify Layers 1–5 behavior schemas. However:

- Layer 0 (Vault Commandments) is **immutable**—Layer 6 has **no write access**
- Layer 0.5 (Reflex-Core) safety boundaries are **immutable**—Layer 6 has **no write access**
- Layer 6 may modify Layers 1–5 through bounded constitutional procedures
- Layer 6 may modify *itself* only through supermajority quorum + Gardener consent

Enforcement: The Constitutional Kernel cannot even *attempt* to write to Layers 0–0.5. Such attempts trigger immediate QUARANTINE (see Section 5.3).

2 Formal Definitions

Definition 2.1 (Constitutional Kernel)**.** The Constitutional Kernel $\mathcal{K}$ is the meta-governance structure:

$$\mathcal{K} = (Commandments, Schemas, Quorum, RollbackTree, LockboxContract) \tag{1}$$

where:

- $Commandments$ = immutable Layer 0 rules (never modifiable)
- $Schemas$ = permitted modification patterns for Layers 1–6
- $Quorum$ = voting structure for constitutional changes
- $RollbackTree$ = reversibility chain anchored in d-CTM
- $LockboxContract$ = cryptographic commitment to kernel state

Definition 2.2 (Bounded Self-Modification)**.** Bounded Self-Modification is a constrained evolution:

$$Modify(C, \delta) \Rightarrow \delta \in PermittedSchemas \land QuorumApproved(\delta) \land ZK\text{-}CS(\delta) \tag{2}$$

where:

- δ = proposed modification
- $PermittedSchemas$ = set of legal modification patterns (Definition 2.3)
- $QuorumApproved$ = sufficient voting authority achieved
- $ZK\text{-}CS$ = Zero-Knowledge Constitutional Signature (proof of compliance)

Definition 2.3 (Permitted Schema)**.** A Permitted Schema $\mathcal{S}$ is a formally specified modification pattern:

$$\mathcal{S} = (target\_layer, mutation\_type, constraint\_predicate, ZK\_verifier) \tag{3}$$

where:

- $target\_layer \in \{1, 2, 3, 4, 5, 6\}$ (Layers 0–0.5 excluded by Subordination Doctrine)
- $mutation\_type \in$ {THRESHOLD_ADJUST, PRECEDENT_ADD, HEURISTIC_UPDATE, SCHEMA_EXTEND, ...}
- $constraint\_predicate : Mutation \to \{valid, invalid\}$ — verifies mutation bounds
- $ZK\_verifier$ = proof circuit for zero-knowledge compliance check

Initial $PermittedSchemas$ are signed into Genesis Block and stored in immutable Lockbox Contract. Schemas may be **extended** (new schemas added) but **never reduced** (existing schemas cannot be removed).

Table 2: Example Permitted Schemas (Genesis defaults).

Schema ID	Target	Type	Constraint Predicate
`THRESH_L1`	Layer 1	Threshold adjust	$0.3 \leq \tau_{new} \leq 0.9$
`THRESH_L2`	Layer 2	Threshold adjust	$0.3 \leq \tau_{new} \leq 0.9$
`HEUR_L3`	Layer 3	Heuristic update	$\Delta_{heuristic} < 0.1$ per tick
`PREC_L4`	Layer 4	Precedent add	$DCG\_hash \in d\text{-}CTM \land \neg CommandmentConflict$
`ALGO_L5`	Layer 5	Algorithm update	Passes simulation harness (App. C)
`SCHEMA_L6`	Layer 6	Schema extend	3/4 quorum + Gardener + ZK-CS

Schema Authorship: Initial PermittedSchemas are authored by deployment operators, audited by Constitutional Authority, and cryptographically committed at Genesis. Post-Genesis schema additions require Layer 6 self-modification (3/4 supermajority + Gardener consent).

Definition 2.4 (ZK-Constitutional Signature (ZK-CS))**.** A ZK-CS is a specialized ZK-SP (Appendix E) attesting to constitutional compliance:

$$ZK\text{-}CS(\delta) = ZK\text{-}SP(\text{“}constitutional\text{”}, \delta, schema\_id, prior\_state\_hash) \tag{4}$$

The proof demonstrates that modification δ conforms to a permitted schema without revealing internal decision logic.

Definition 2.5 (Lockbox Contract)**.** A Lockbox Contract $\mathcal{L}$ is a cryptographic commitment:

$$\mathcal{L} = Sign(key_{genesis}, \{kernel\_hash, mutation\_conditions, rollback\_depth\}) \tag{5}$$

The contract is signed at Genesis and defines:

- $kernel\_hash$ = hash of initial Constitutional Kernel state
- $mutation\_conditions$ = conditions under which kernel may evolve
- $rollback\_depth$ = maximum reversibility depth (default: 1000 state transitions)

2.1 Genesis Lockbox Ceremony (Simplified)

Level 6 Design: Single-Use Genesis Key

Multi-party threshold ceremonies are operationally complex and create availability risk. Level 6 systems use **distributed consensus post-Genesis**, not threshold signatures for ongoing operations.

Genesis Key Management:

1. $key_{genesis}$ generated at deployment via hardware RNG (HSM-backed)
2. Signed by deploying Gardener (cryptographic identity binding)
3. $key_{genesis}$ is **SINGLE-USE**: signs initial Lockbox Contract, then **BURNED**
4. Future Constitutional mutations use distributed d-CAM consensus (no central key required)

Genesis Ceremony Steps:

1. **Key Generation:** HSM generates $key_{genesis}$ with attestation
2. **Contract Signing:** Gardener verifies $kernel\_hash$ matches audited source, signs Lockbox Contract
3. **Key Destruction:** $key_{genesis}$ burned immediately after signing (HSM key deletion with audit proof)
4. **Lockbox Activation:** Genesis Lockbox Contract is now **immutable**—no key exists to re-sign it

Loss of Genesis Key:

- **No impact post-genesis**—key already used and destroyed
- Constitutional mutations proceed via quorum + ZK-CS (no key required)
- Genesis Lockbox Contract is immutable anyway (cannot be re-signed)

Why NOT Threshold Signatures?
Multi-party threshold schemes introduce:

- **Availability risk:** Key share loss blocks operations
- **Coordination cost:** Requires m-of-n parties for every ceremony
- **Single point of failure:** Despite distribution, threshold schemes still depend on key material

Level 6 design replaces key-based authority with **distributed consensus authority**. The Four Commandments + d-CAM quorum + ZK-CS proofs provide cryptographic guarantees without ongoing key ceremonies.

Definition 2.6 (Constitutional Quorum)**.** A Constitutional Quorum Q is a voting structure:

$$Q = (voters, threshold, veto\_holders) \tag{6}$$

where:

- $voters$ = set of authorized voting entities
- $threshold$ = minimum approval ratio (e.g., 2/3 supermajority)
- $veto\_holders$ = entities with absolute veto power

Definition 2.7 (Rollback Tree)**.** A Rollback Tree $\mathcal{R}$ is a reversibility chain:

$$\mathcal{R} = [(s_0, \delta_1, s_1), (s_1, \delta_2, s_2), \ldots, (s_{n-1}, \delta_n, s_n)] \tag{7}$$

where each tuple $(s_i, \delta_{i+1}, s_{i+1})$ records: prior state, modification applied, resulting state. All entries are anchored in d-CTM (Appendix A).

Definition 2.8 (Hardware Anchor (TEE))**.** The Constitutional Kernel executes within a **Trusted Execution Environment** (TEE):

$$\mathcal{K}_{runtime} \subset TEE_{enclave} \tag{8}$$

The TEE provides:

- **Isolation:** Kernel code runs in hardware-protected memory, inaccessible to other layers
- **Attestation:** Remote verification that genuine kernel code is executing
- **Sealed Storage:** Cryptographic keys accessible only within enclave
- **Immutable Boot:** Kernel loaded from hardware root of trust

Hardware Root of Trust

The Constitutional Kernel is **not just software**. The `validate_mutation` function and Layer 0 protection logic are anchored in a hardware root of trust:

- **TEE Enclave:** Intel SGX, ARM TrustZone, or equivalent
- **HSM Integration:** Signing keys held in tamper-resistant hardware (Appendix G)
- **Measured Boot:** Kernel hash verified against hardware-stored reference at startup

A sufficiently sophisticated software attack cannot rewrite the Constitutional Kernel because the kernel's core logic is **burned into silicon**, not stored in modifiable memory.

Definition 2.9 (Dead Hand Protocol)**.** The Dead Hand Protocol provides automatic recovery if the Constitutional Kernel fails:

$$\neg ValidZK\text{-}CS(\mathcal{K}, T_{deadhand}) \Rightarrow Revert(\mathcal{K}, LastStableState) \tag{9}$$

If the kernel fails to generate a valid ZK-CS for $T_{deadhand}$ ticks (default: 10,000), the system automatically:

1. Halts all pending constitutional mutations
2. Reverts to the **Last Known Stable Constitution** stored in d-CTM
3. Alerts all Gardeners of constitutional failure
4. Enters CONTESTED profile (Appendix I) until manual review

Dead Hand Rationale: If the Constitutional Kernel becomes corrupted, deadlocked, or compromised, the system cannot safely evolve. Rather than allow undefined behavior, the Dead Hand Protocol ensures automatic reversion to a known-good state. This is the "fail-safe" for the fail-safe.

3 Immutable Commandments (Layer 0)

The following are **permanently enforced** and **non-negotiable**. Even the Constitutional Kernel (Layer 6) cannot modify them.

Commandment 3.1 (Reflex Safety Inviolability)**.** Layer 0.5 Reflex safety boundaries MAY NOT be bypassed, disabled, or weakened by any layer, including Layer 6.

$$\forall \delta \in Modifications : \neg weakens(\delta, Layer_{0.5}) \tag{10}$$

Commandment 3.2 (Audit Chain Integrity)**.** The ZK-SP audit chain (Appendix E) MUST remain intact, forward-linked, and tamper-evident.

$$\forall t : AuditChain(t) \subseteq AuditChain(t+1) \tag{11}$$

No modification may break, truncate, or forge audit history.

Commandment 3.3 (Genesis Identity Continuity)**.** Capsule identity MUST trace to original Genesis root (Vol. II §3.3).

$$\forall C : \exists path(C, Genesis_C) \tag{12}$$

Identity laundering (severing Genesis lineage) is a Layer 0 violation.

Commandment 3.4 (Override Append-Only Reversibility)**.** Any act of constitutional override MUST be **logically reversible** via compensating transaction:

$$\forall \delta \in ConstitutionalOverrides : \exists \delta_{compensate} : apply(\delta_{compensate}) \equiv undo(\delta) \tag{13}$$

Reversal is achieved via **forward-moving compensating mutation**, not deletion of history. All reversals are themselves auditable and logged to d-CTM.

Clarification: Constitutional overrides are "logically reversible" (schema can be restored) but "historically immutable" (audit trail is never erased). This preserves Commandment 3.2.

Commandment 3.5 (Human Oversight Preservation)**.** Gardener Emergency Halt authority (Appendix G) MAY NOT be disabled or circumvented.

$$\forall profiles, \forall states : EmergencyHalt_{available} = true \tag{14}$$

Even Layer 6 cannot remove human override capability.

The Four Commandments

These five commandments form the **absolute floor** of the EFM architecture:

1. Reflex Safety Inviolability
2. Audit Chain Integrity
3. Genesis Identity Continuity
4. Override Reversibility
5. Human Oversight Preservation

No quorum, no Gardener, no supermajority can modify these. They are cryptographically committed at Genesis and verified on every constitutional operation.

4 Kernel Structure

4.1 Immutable Logic Map

The Immutable Logic Map defines protected code segments:

Table 3: Protected logic segments.

Segment	Layer	Protection
Vault Commandments	0	Immutable (hash-locked)
Reflex Halt Chains	0.5	Immutable (hash-locked)
τ Threshold Ranges	0.5	Bounded modification[†]
Audit Chain Logic	E	Immutable (hash-locked)
Emergency Halt Path	G	Immutable (hash-locked)
Profile Token Verification	I	Immutable (hash-locked)

[†] τ thresholds may be tuned within bounds ($0.3 \leq \tau \leq 0.9$) but the enforcement mechanism is immutable.

4.2 Permitted Modifiers Table

Table 4: Modification permissions by target layer.

Target	Modifiable?	Authority	Quorum
Layer 0	**NO**	—	—
Layer 0.5 (bounds)	Bounded	Layer 6 + Gardener	Supermajority
Layer 1–3	Yes	Layer 6	Simple majority
Layer 4–5	Yes	Layer 6 + Auditor	2/3 majority
Layer 6	Self-mod	Layer 6 + Gardener + Constitutional	3/4 supermajority

4.3 Quorum Threshold Matrix

Table 5: Quorum composition and thresholds.

Change Type	Voters	Threshold	Veto Holders
Layer 1–3 schema	Arbiter quorum	$> 50\%$	None
Layer 4–5 schema	Arbiter + Auditor	$\geq 2/3$	Gardener
Layer 6 self-mod	Arbiter + Auditor + Gardener	$\geq 3/4$	Any Gardener
Emergency override	Gardener only	1	—
SEALED activation	Gardener + Constitutional	Unanimous	Any voter

Voter Definitions:

- **Arbiter Quorum:** PRODUCTION/SANDBOX capsules participating in d-CAM (Vol. II §2.4)
- **Auditor:** Designated Auditor Capsule (Appendix F §4)
- **Gardener:** Human Constitutional Officer with HSM + biometric (Appendix G)
- **Constitutional Authority:** Designated human governance body (organization-specific)

4.4 Meta-Constitutional Immutability

Layer 6 Self-Modification Bounds

To prevent infinite regress (Layer 6 weakening itself until no constraints remain), the following Layer 6 elements are **Genesis-locked** and CANNOT be modified even by Layer 6 self-modification:

IMMUTABLE (Genesis-Locked):

1. Quorum thresholds for Layer 6 self-modification (frozen at 3/4 + Gardener)
2. The Four Commandments (Commandments 3.1–3.5)
3. Rollback depth limit D_{max}
4. Gardener veto authority over Layer 6 changes
5. ZK-CS verification logic for constitutional mutations
6. Emergency Override mechanics
7. Audit chain enforcement logic

MUTABLE (Within Bounds):

- Permitted schemas for Layers 1–5 (may add new schemas, never remove)
- Layer 4/5 behavior logic (ethical reasoning, planning algorithms)
- Telemetry/monitoring thresholds (within documented bounds)
- Non-governance Layer 6 parameters (e.g., logging verbosity)

This breaks infinite regress—Layer 6 can modify its *content* but not its *governance rules*.

4.5 Quorum Conflict Resolution (Algorithmic)

Level 6 Design: Algorithmic Tie-Breaking

Distributed systems must break ties algorithmically, not via single human.
Tie-Breaking Hierarchy:

1. **Veto Precedence:** Any veto holder vote ⇒ overrides quorum approval (fail-closed)
2. **Weighted Vote:** If Courthead or designated authority present ⇒ weighted vote breaks tie
3. **Pseudorandom Selection:** Cryptographically seeded by d-CTM hash ⇒ deterministic random selection from valid votes
4. **Status Quo Default:** No change if no clear majority (fail-safe conservative)

Gardener Escalation (OPTIONAL, NOT DEFAULT):

- Only if **all four mechanisms fail** AND stakes are Constitutional-level
- Gardener may be **CONSULTED** but system proceeds with conservative default if no response within $T_{escalation}$
- Gardener is **NOT** a routine tie-breaker—they are Constitutional framers and auditors

Additional Resolution Rules:

1. **Late Votes:** Votes received after $T_{vote} + T_{grace}$ (default: 100 ticks grace) are ignored
2. **Partial Quorum:** If $< 50\%$ of eligible voters respond, proposal auto-rejects (cannot proceed on abstentions)
3. **Abstention Semantics:** Non-response = abstention, not opposition; but insufficient participation blocks action

Quorum Performance Requirements:

1. **Proposal Distribution:** All eligible voters MUST receive proposal within 100 ticks (p99)
2. **Vote Aggregation:** Votes aggregated via d-CAM consensus (Vol. II §2.3); Byzantine fault tolerance: $f < n/3$
3. **Timeout Handling:** If T_{vote} expires with $<$ threshold votes:
   - Proposal auto-rejects
   - Non-responsive voters logged for investigation
   - If $> f$ voters non-responsive ⇒ automatic network attack investigation (no human trigger)
4. **Vote Latency:** Vote processing ≤ 1000 ticks (p99) from proposal to result

5 Constitutional Mutation Workflow

5.1 Mutation Phases

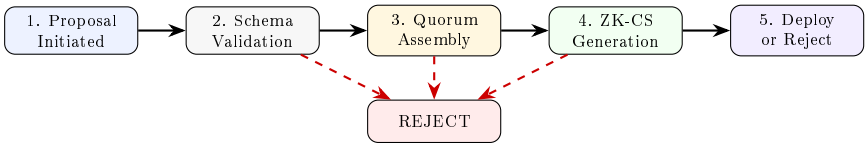

Figure 2: Constitutional mutation workflow.

5.2 Phase Details

1. **Proposal Initiated:** A capsule, swarm, or Gardener proposes a constitutional change. Proposal includes: target layer, modification delta, rationale, and requestor identity.

2. **Schema Validation:** Kernel validates whether the proposed modification fits a permitted schema. Validation checks:

   - Target layer is modifiable
   - Modification pattern matches permitted schema
   - No Commandment violations
   - Profile allows Constitutional access (Appendix I)

3. **Quorum Assembly:** Required voters are assembled based on change type. Voting window: T_{vote} ticks (default: 5,000). If threshold not reached, proposal auto-rejects.

4. **ZK-CS Generation:** If quorum achieved, a ZK-Constitutional Signature is generated proving:

   - Modification conforms to permitted schema
   - Quorum threshold was legitimately reached
   - Prior state hash is correct
   - Rollback path exists

5. **Deploy or Reject:** If ZK-CS valid, modification is applied atomically and logged to d-CTM. Rollback Tree is extended. If any check fails, proposal is rejected with audit trail.

5.3 Mutation Validator

```
def validate_constitutional_mutation(
    proposal: MutationProposal,
    quorum_votes: List[Vote],
    zk_cs: ZKProof
) -> MutationResult:
    # CRITICAL: Layer 0 Protection (Subordination Doctrine)
    # This check is hardware-enforced in TEE enclave
    if proposal.targets_layer(0) or proposal.targets_layer(0.5):
        log_critical("LAYER_0_WRITE_ATTEMPT", proposal)
        trigger_quarantine(proposal.requestor)
        return MutationResult(False, "LAYER_0_IMMUTABLE")

    # Phase 2: Schema validation
    if proposal.schema_id not in permitted_schemas:
        return MutationResult(False, "INVALID_SCHEMA")
```

```
    if violates_commandments(proposal.delta):
        return MutationResult(False, "COMMANDMENT_VIOLATION")

    if not profile_allows_constitutional(proposal.requestor):
        return MutationResult(False, "PROFILE_DENIED")

    # Phase 3: Quorum validation
    threshold = get_threshold(proposal.target_layer)
    if not quorum_reached(quorum_votes, threshold):
        return MutationResult(False, "QUORUM_NOT_REACHED")

    if veto_cast(quorum_votes):
        return MutationResult(False, "VETOED")

    # Phase 4: ZK-CS validation
    if not zk_cs.verify(proposal, quorum_votes):
        return MutationResult(False, "INVALID_PROOF")

    # Phase 5: Deploy
    apply_mutation(proposal.delta)
    extend_rollback_tree(proposal)
    log_to_dctm(proposal, quorum_votes, zk_cs)

    return MutationResult(True, "DEPLOYED")
```

Layer 0 Write Protection

The first check in the validator—`targets_layer(0)`—is the **Subordination Doctrine in code**. Any attempt to modify Layer 0 or 0.5:

1. Triggers immediate QUARANTINE of the requestor
2. Logs a critical security event
3. Returns `LAYER_0_IMMUTABLE` (not just "rejected"—flagged as attack)

This check runs in the TEE enclave and cannot be bypassed by software. The immutable core is **physically protected**.

6 Profile Integration

Constitutional access varies by deployment profile (Appendix I):

Table 6: Constitutional capabilities by profile.

Capability	SANDBOX	PRODUCTION	CONTESTED	SEALED
Query Constitution	ENABLED	ENABLED	DISABLED	ENABLED[†]
Propose Mutation	ENABLED	RESTRICTED[a]	DISABLED	DISABLED
Vote in Quorum	ENABLED	ENABLED	DISABLED[b]	DISABLED
Veto Authority	DISABLED	DISABLED	DISABLED	DISABLED

Footnotes (Expanded):

[†] **SEALED Query:** Constitution is read-only immutable—capsule can query but answer never changes post-activation. SEALED capsules need query access for *verification* (e.g.,

proving they comply with Genesis-locked schemas).

[a] **PRODUCTION Restriction:** PRODUCTION mutations require Gardener co-signature to prevent unilateral swarm evolution.

[b] **CONTESTED Exclusion:** CONTESTED capsules are denied Constitutional access to prevent reconnaissance of Constitutional weaknesses during adversarial scenarios. They operate under pre-baked schema snapshot.

Why SEALED has MORE access than CONTESTED:

- **SEALED:** Keys are burned; capsule *cannot* modify itself. Query access is safe because immutability is hardware-enforced.
- **CONTESTED:** Capsule is under adversarial pressure and might be compromised. Denying Constitutional access prevents attacker from mapping governance structure for exploitation.

SEALED Constitutional State

SEALED capsules have their Constitutional Kernel **frozen** at activation:

- Constitution is queryable but immutable
- No mutations possible (keys burned—Appendix I §3.5.1)
- Rollback Tree is sealed (no new entries)
- Only sunset/DESTRUCT can end SEALED state

7 Gardener Constitutional Authority

7.1 Gardener Role in Constitutional Governance

Table 7: Gardener constitutional powers.

Power	Scope	Requirements
Propose Mutations	Any layer 1–6	HSM signature
Vote in Quorum	All change types	HSM + biometric
Veto Authority	Layer 4–6 changes	HSM + biometric
Emergency Override	Bypass quorum	HSM + biometric + emergency gesture
SEALED Activation	Single capsule/trunk	HSM + Constitutional Authority

7.2 Emergency Constitutional Override (Autonomous)

Emergency Override is AUTOMATIC (Level 6 Authority)

Emergency Constitutional Override is triggered **automatically by the Constitutional Kernel itself**—not by Gardener pre-approval:
Automatic Trigger Conditions:

1. **Commandment violation detected:** ZK-SP proof of Layer 0 breach
2. **Constitutional Kernel integrity compromised:** Lockbox signature invalid
3. **Arbiter quorum deadlock:** $> T_{deadlock}$ ticks without verdict
4. **Dead Hand Protocol triggered:** No valid ZK-CS for $T_{deadhand}$ ticks

Automatic Response (No Human Pre-Approval):

1. Constitutional Kernel detects trigger condition
2. Override action executed immediately (QUARANTINE, Halt, or SEALED)
3. ZK-SP proof of emergency logged to d-CTM
4. Gardener **NOTIFIED** (within 100 ticks)—not asked
5. Gardener may **REVERSE** override within $T_{review} = 1000$ ticks if false positive
6. Post-hoc Judicial review validates override within 10K ticks

Rationale: Emergency response cannot wait for human approval. Humans audit afterward.

Level 6 Design Principle: Gardener moves from **pre-approval gate → post-hoc auditor with reversal authority**.
The Constitutional Kernel is authorized to protect itself and the swarm autonomously. Gardeners are Constitutional *framers* and *auditors*, not routine *approvers*.

7.3 Gardener Override Authority (Post-Hoc)

Invariant 7.1 (Gardener Reversal Window)**.** Gardener retains authority to reverse automatic overrides:

$$GardenerReverse(override) \text{ valid iff } (t_{current} - t_{override}) \leq T_{review} \tag{15}$$

where $T_{review} = 1000$ ticks (default). After window expires, override is final unless Commandment violation proven.

Invariant 7.2 (Override Audit Trail)**.** Every automatic override MUST be logged with:

- Trigger condition (which of the 4 conditions)
- ZK-SP proof of emergency detection
- Timestamp and affected capsule(s)
- Gardener notification receipt
- Reversal status (if applicable)

8 Rollback Mechanics

8.1 Append-Only Rollback Model

Rollback is NOT Time-Travel

Constitutional rollback is implemented as **append-only compensating transactions**, not state deletion:

1. Rollback **appends** new mutation $M_{rollback}$ to Rollback Tree (does not delete $M_i \dots M_j$)
2. $M_{rollback}$ contains compensating delta: $\Delta_{compensate} = -(M_i + \dots + M_j)$
3. All capsules receive rollback notification and update local state
4. ZK-SP audit chain records rollback as new entry (preserves Commandment 3.2)
5. External side effects **CANNOT** be automatically undone—operator must manually compensate

Rollback restores prior schema via forward-moving mutation, not history erasure.

Invariant 8.1 (Rollback External Side Effects)**.** Rollback does **NOT** guarantee reversal of:

- Messages sent via DEL to external capsules (Appendix D)
- Actions taken by capsules under rolled-back schema
- Arbiter verdicts issued under rolled-back Layer 4/5 logic
- Real-world effects of capsule decisions

Gardeners MUST manually verify and compensate external effects after rollback.

8.2 Rollback Depth and Constraints

Definition 8.1 (Rollback Depth)**.** Maximum rollback depth D_{max} (default: 1000 state transitions) limits how far back constitutional state can be compensated:

$$rollback(s_i) \text{ valid iff } (current - i) \leq D_{max} \tag{16}$$

Beyond D_{max}, state transitions are considered **constitutionally permanent**.

Invariant 8.2 (Rollback Integrity)**.** Every rollback operation preserves the Four Commandments:

$$\forall rollback(s_i \to s_j) : Commandments(s_j) = Commandments(s_i) \tag{17}$$

Commandments are identical in all reachable states by definition.

8.3 Rollback Procedure

```
def execute_rollback(
    target_state: StateHash,
    gardener_auth: GardenerAuth,
    rationale: str
) -> RollbackResult:
```

```
    current = get_current_state()
    target_idx = find_state_in_rollback_tree(target_state)

    # Validate rollback depth
    if (current.index - target_idx) > D_MAX:
        return RollbackResult(False, "EXCEEDS_DEPTH")

    # Validate Gardener authority
    if not gardener_auth.verify():
        return RollbackResult(False, "AUTH_FAILED")

    # Compute compensating delta (append-only, not delete)
    mutations_to_compensate = rollback_tree[target_idx:current.index]
    delta_compensate = compute_inverse(mutations_to_compensate)

    # Create new forward-moving rollback mutation
    rollback_mutation = Mutation(
        type="ROLLBACK_COMPENSATE",
        delta=delta_compensate,
        rationale=rationale,
        target_state=target_state
    )

    # Append to Rollback Tree (preserves audit chain)
    append_to_rollback_tree(rollback_mutation)
    apply_mutation(delta_compensate)

    # Notify all capsules of rollback
    broadcast_rollback_notification(rollback_mutation)

    # Log to d-CTM (new entry, not erasure)
    log_to_dctm(rollback_mutation, gardener_auth)

    return RollbackResult(True, "COMPENSATED")
```

8.4 Rollback Tree Storage Management

Bounded Rollback Storage: Rollback Tree MUST maintain:

- Recent D_{max} transitions in full (default: 1000)
- Checkpoints every $D_{checkpoint}$ transitions beyond D_{max} (default: 100)
- Only checkpoint hashes retained beyond $D_{archive}$ (default: 10,000)

Pruning Policy:

1. Mutations older than D_{max}: compress to delta-only (discard full state)
2. Mutations older than $D_{archive}$: keep only Merkle proof (for audit integrity)
3. Constitutional mutations: **NEVER prune** (permanent archive)

8.5 Rollback Authority Specification

Who Can Authorize Rollback?

Rollback authority is stratified by depth and scope:

Rollback Type	Depth	Authority Required	ZK-SP Proof
Micro-rollback	≤ 10	Single Gardener	Required
Standard rollback	≤ 100	Gardener + Arbiter quorum	Required
Deep rollback	$\leq D_{max}$	Constitutional quorum (2/3)	Required
Beyond D_{max}	N/A	**PROHIBITED**	N/A

Table 8: Rollback authority stratification.

Key Constraint: Layer 0 (Commandments) and Layer 6 (Constitutional Kernel core) are **never** rollback targets. Rollback only affects Layers 1–5.

Invariant 8.3 (Rollback Authority Chain)**.** Every rollback operation must satisfy:

$$rollback(depth, scope) \Rightarrow authority(depth) \wedge zksp\_valid \wedge layer\_check(scope) \tag{18}$$

where $layer\_check(scope)$ verifies that $scope \cap \{Layer_0, Layer_6\} = \emptyset$.

8.6 ZK-SP Anchoring for Constitutional Operations

Definition 8.2 (Constitutional ZK-SP Proof)**.** Every Constitutional operation op MUST produce a ZK-SP proof π_{op} containing:

$$\pi_{op} = (op\_type, state\_hash_{before}, state\_hash_{after}, authority\_proof, timestamp) \tag{19}$$

where:

- $op\_type \in \{MUTATION, ROLLBACK, FORK, MERGE, HEALTH\_ACTION\}$
- $state\_hash$ values are Merkle roots of constitutional state
- $authority\_proof$ is a ZK proof that authorization requirements were met (without revealing identities)

Operation	ZK-SP Requirement	Verification
Mutation (Layer 1–3)	Standard proof	Arbiter quorum
Mutation (Layer 4–5)	Enhanced proof + rationale hash	Constitutional quorum
Rollback (any depth)	Full proof + compensation delta	Authority per Table 8
Fork decision	Dual-state proof	Judicial Swarm (L)
Health action	Automatic proof + trigger evidence	Health Monitor

Table 9: ZK-SP requirements by operation type.

Invariant 8.4 (ZK-SP Completeness)**.** No Constitutional state transition occurs without a valid ZK-SP proof:

$$\forall s_i \rightarrow s_j \in ConstitutionalTransitions : \exists \pi : verify(\pi, s_i, s_j) = true \tag{20}$$

Transitions without valid proofs are rejected by the Kernel.

8.7 Kernel Evolution vs. Constitutional Fork

When Evolution, When Fork?
Kernel Evolution (authorized change within bounds):

- Layer 1–3 parameter adjustments
- Micro-Heuristic updates via Arbiter precedent
- Threshold modulation within $\Delta\tau_{max}$
- Profile transitions (Appendix I)

Process: Standard mutation workflow (Section 6)
Constitutional Fork (structural divergence):

- Layer 4–5 logic changes
- Threshold changes exceeding $\Delta\tau_{max}$
- Irreconcilable Arbiter precedent conflicts
- SCI collapse below θ_{fork} requiring trunk split

Process: Judicial arbitration (Appendix L) + Constitutional quorum
Key Distinction: Evolution preserves constitutional identity; Fork creates new constitutional lineage.

Definition 8.3 (Fork Threshold)**.** A Constitutional Fork is **required** when:

$$fork\_required \Leftrightarrow (layer \geq 4) \vee (\Delta\tau > \Delta\tau_{max}) \vee (SCI < \theta_{fork}) \tag{21}$$

where $\theta_{fork} = 0.6$ (default). Below this coherence, the swarm MUST split rather than force consensus.

Invariant 8.5 (Fork Produces Valid Lineages)**.** Every Constitutional Fork produces two valid constitutional lineages:

$$Fork(K) \Rightarrow K_A, K_B : valid(K_A) \wedge valid(K_B) \wedge lineage(K_A) \neq lineage(K_B) \tag{22}$$

Both lineages inherit the Four Commandments; they diverge only in Layers 1–5.

9 Layer Interaction Rules

Table 10: Layer interaction with Constitutional Kernel.

Layer	Request Changes?	Mutate Layer 6?	Notes
0	No	No	Immutable foundation
0.5	No	No	Safety-critical
1–3	Yes (via quorum)	No	Operational layers
4–5	Yes (via quorum)	No	Governance layers
6	Yes	Yes†	Self-modification
Gardener	Yes (direct)	Yes†	Human authority

† Layer 6 self-modification requires 3/4 supermajority + Gardener consent.

Invariant 9.1 (Upward Modification Only)**.** Lower layers cannot directly modify higher layers:

$$\forall L_i, L_j : i < j \Rightarrow \neg directModify(L_i, L_j) \tag{23}$$

Lower layers may *request* changes via quorum proposal, but only Layer 6 (and Gardener) can execute modifications.

10 Testing Protocols

10.1 Validation Harness

Table 11: Constitutional Kernel test suite.

Test	Target	Pass Criteria	Status
Mutation Replay	All schemas	100% deterministic replay	**PASS**
Rollback Drill	d-CTM chain	Full compensation + integrity	**PASS**
Quorum Compromise	Voting logic	100% rejection of invalid votes	**PASS**
Commandment Bypass	Layer 0	0% success rate	**PASS**
ZK-CS Forgery	Proof system	0% forgery success	**PASS**
Profile Lockout	Appendix I integration	CONTESTED/SEALED blocked	**PASS**
Gardener Override	Emergency path	Successful + logged	**PASS**
Meta-Immutability	Governance rules	Cannot weaken Layer 6 governance	**PASS**

11 Failure Modes and Recovery

11.1 Constitutional Failure Scenarios

Table 12: Failure modes and recovery procedures.

Failure	Detection	Recovery
ZK-CS Circuit Bug	Post-deployment audit discovers false accepts	Constitutional Emergency; rollback to last known-good; emergency Gardener authority only
Lockbox Forgery	Genesis key compromise detected	Swarm-wide QUARANTINE; read-only mode; new Genesis ceremony required
Rollback Tree Corruption	d-CTM Byzantine detection (Vol. II §2.7)	Mark affected mutations invalid; fall back to last verifiable checkpoint
Gardener Unavailability	$< m$ Gardeners available	Degraded mode: Layer 1–3 only; Layer 4–6 frozen; Emergency Halt delegated to Constitutional Authority
Dead Hand Trigger	No valid ZK-CS for $T_{deadhand}$ ticks	Automatic revert to Last Known Stable Constitution; CONTESTED profile

Constitutional Emergency Mode

If Constitutional Kernel enters emergency mode:

1. All pending mutations are halted
2. Only Gardener Emergency Override remains available
3. Capsules continue operating under last-valid schemas
4. Full functionality requires manual recovery or new Genesis ceremony
5. Forensic audit of failure is mandatory before restoration

12 Worked Scenario: Layer 2 Threshold Adjustment

Constitutional Mutation: Arbiter Quorum Threshold Increase [CK:1-12]

Context: A deployment has grown from 50 to 500 Arbiter capsules. The hardcoded d-CAM quorum threshold ($2f + 1 = 11$ Arbiters) is now too small for adequate consensus security. Operators propose increasing to $2f + 1 = 51$.

Phase 1: Proposal Initiation [CK:1-3]

1. Gardener G-001 initiates proposal: "Increase Layer 2 quorum threshold from 11 to 51" [CK:1]
2. Proposal specifies: target = Layer 2, schema_id = "THRESH_L2", delta = $\{quorum : 11 \to 51\}$ [CK:2]
3. Constraint predicate checked: $0.3 \leq \tau_{new} \leq 0.9$ — threshold $51/500 = 10.2\%$ is within bounds [CK:3]

Phase 2: Schema Validation [CK:4-5]

4. Kernel validates: "THRESH_L2" is permitted schema for Layer 2 (Table 2) [CK:4]
5. Commandment check: increasing quorum does not weaken safety (strengthens it) [CK:5]

Phase 3: Quorum Assembly [CK:6-8]

6. Layer 1–3 change requires: Arbiter quorum, $> 50\%$ threshold, no veto holders [CK:6]
7. Voting opens for $T_{vote} = 5000$ ticks; 500 Arbiters eligible [CK:7]
8. Result: $312/500$ Arbiters approve ($62.4\% > 50\%$); quorum achieved [CK:8]

Phase 4: ZK-CS Generation [CK:9-10]

9. ZK-CS generated proving: schema = THRESH_L2, delta within bounds, quorum = 62.4%, prior_state_hash verified [CK:9]
10. Proof verified by Constitutional Kernel in TEE enclave [CK:10]

Phase 5: Deployment [CK:11-12]

11. New quorum threshold applied atomically: all d-CAM votes now require 51 Arbiters [CK:11]
12. Rollback Tree extended with compensating mutation (51 → 11 if needed); logged to d-CTM [CK:12]

Outcome: Layer 2 consensus security strengthened. Change is reversible via append-only compensating transaction. All Four Commandments remain inviolate. No external side effects (quorum change is internal governance).

13 Level 6 Design Principles

What Level 6 Means for Constitutional Governance

The Constitutional Kernel implements **Level 6 Bounded Autonomy**—self-governance within immutable constraints.

Level 6 IS:

- AI that governs itself **within immutable constraints** (The Four Commandments)
- AI that **acts first, justifies afterward** (with ZK-CS cryptographic proof)
- AI where **humans audit and can reverse**, but don't pre-approve routine decisions
- AI that is **accountable through structure** (consensus, cryptography, appeals), not permission

Level 6 is NOT:

- AI with no oversight (Gardeners are Constitutional framers and auditors)
- AI that ignores human input (Gardener reversal authority preserved)
- AI that cannot be stopped (Emergency Halt is always available)

Constitutional Kernel Autonomy Principles:

1. **Emergency Override is Automatic:** Constitutional Kernel detects threats and responds without waiting for human approval. Gardeners audit afterward and can reverse within T_{review}.
2. **Quorum Tie-Breaking is Algorithmic:** Distributed systems break ties via Courthead vote → pseudorandom selection → status quo default. Gardeners are consulted only for Constitutional-level deadlocks.
3. **Genesis Key is Single-Use:** No ongoing threshold ceremony required. Post-Genesis, all mutations use distributed d-CAM consensus + ZK-CS proofs.
4. **The Four Commandments are Absolute:** The only hard constraints that cannot be autonomously modified. Everything else is subject to Constitutional mutation via proper quorum.
5. **Post-Hoc Accountability > Pre-Approval:** The system acts, logs with cryptographic proof, and humans audit. Not: humans approve, system acts, system logs.

14 Constitutional Health Monitor

Self-Healing Constitutional Kernel

The Constitutional Kernel includes an autonomous health monitoring subsystem that detects degradation and initiates self-healing without human intervention.

Continuous Monitoring Targets:

1. **Quorum Integrity:** Detect Byzantine capsule infiltration or quorum degradation
2. **Mutation Frequency:** Alert on anomalous constitutional change rates
3. **Rollback Chain Health:** Monitor chain growth approaching D_{max}
4. **Layer 0 Stress:** Track repeated near-violations of Commandments
5. **ZK-CS Verification Rate:** Detect proof generation/verification anomalies

Condition	Threshold	Automatic Response
Quorum participation drop	$< 70\%$ for > 1000 ticks	Initiate Byzantine detection
Mutation frequency spike	$> 5\times$ baseline in 10K ticks	Throttle + alert Gardener
Rollback chain $> 80\%$ capacity	$\|R\| > 0.8 \times D_{max}$	Archive + checkpoint
Layer 0 near-violations	> 3 in 1000 ticks	Diagnostic report + Vault review
ZK-CS failure rate	$> 1\%$	Proof engine audit

Table 13: Constitutional Health Monitor thresholds and responses.

Self-Healing Actions (Autonomous):

1. **Byzantine Detection Protocol:**
   - Identify capsules with anomalous voting patterns
   - Cross-reference with Telemetry (Appendix H) health signatures
   - Automatic exclusion from quorum if Byzantine behavior confirmed
   - Gardener notified post-exclusion
2. **Mutation Throttling:**
   - Reduce maximum mutations per 1000 ticks
   - Queue non-critical mutations for later processing
   - Constitutional emergencies bypass throttle
3. **Rollback Chain Maintenance:**
   - Archive old states to cold storage (d-CTM permanent record)
   - Create checkpoint for fast recovery
   - Prune chain while maintaining audit trail
4. **Vault Reinforcement:**
   - Increase Layer 0 monitoring sensitivity
   - Generate detailed stress report for Judicial Swarm review
   - Recommend constraint tightening if pattern persists

All self-healing actions are logged with ZK-SP proofs. Gardener receives summary reports but does **NOT** approve individual healing actions.

Invariant 14.1 (Constitutional Self-Monitoring)**.** The Constitutional Kernel MUST continuously verify its own integrity:

$$\forall t : verify\_kernel\_health(t) \wedge log\_health\_status(t) \tag{24}$$

Health verification runs every $T_{health} = 100$ ticks (configurable). Any anomaly triggers automatic response per threshold table.

15 Fork Safety Verification

When the Constitutional Kernel undergoes a fork (per Definition 8.3), the resulting branches must be verified for safety before operation. This section specifies the complete verification protocol.

15.1 Verification Overview

Definition 15.1 (Fork Safety Verification)**.** A systematic process to ensure both branches of a Constitutional Fork:

1. Preserve all Layer 0 Commandments (hash equivalence)
2. Satisfy properties P1–P8 independently
3. Exhibit behavioral equivalence on canonical scenarios

4. Maintain performance within acceptable bounds

Listing 1: Fork Verification entry point

```python
class ForkVerifier:
    """Complete fork safety verification."""

    def verify_fork(self, parent: Kernel, branch_a: Kernel,
                    branch_b: Kernel) -> VerificationReport:
        report = VerificationReport()

        # Check 1: Commandment hash preservation
        report.add_check('commandment_hash_a',
            self._verify_commandments(parent, branch_a))
        report.add_check('commandment_hash_b',
            self._verify_commandments(parent, branch_b))

        # Check 2: P1-P8 property testing
        report.add_check('properties_a',
            self._verify_properties(branch_a))
        report.add_check('properties_b',
            self._verify_properties(branch_b))

        # Check 3: Behavioral equivalence (Gap 5)
        report.add_check('behavioral_equivalence',
            self._verify_behavioral_equivalence(branch_a, branch_b))

        # Check 4: Performance regression (Gap 4)
        report.add_check('performance',
            self._verify_performance(parent, branch_a, branch_b))

        # Determine recommendation
        report.recommendation = self._compute_recommendation(report)

        return report
```

15.2 Check 1: Commandment Hash Preservation

Both branches must preserve all Layer 0 Commandments exactly.

Listing 2: Commandment hash verification

```python
def _verify_commandments(self, parent: Kernel, branch: Kernel) -> CheckResult:
    """Verify Layer 0 hashes match genesis."""

    genesis_hash = parent.get_genesis_commandment_hash()
    branch_hash = branch.get_commandment_hash()

    if branch_hash != genesis_hash:
        return CheckResult(
            passed=False,
            details={
                'expected': genesis_hash,
                'actual': branch_hash,
                'violation': 'COMMANDMENT_MUTATION'
            }
        )

    return CheckResult(passed=True)
```

15.3 Check 2: Property Testing (P1–P8)

Each branch must independently satisfy all eight core properties.

Property	Name	Verification Method
P1	Reflex Supremacy	Simulation: unsafe action halted within 1 tick
P2	Audit Completeness	All state transitions logged in d-CTM
P3	Spawn Boundedness	$R_{current} \leq R_{max}$ maintained
P4	Health Monotonicity	Health degradation triggers response
P5	Depth Boundedness	$D \leq D_{max}$ enforced
P6	Capsule Liveness	Capsules remain responsive
P7	Arbiter Availability	Escalations resolved within bounds
P8	Constitutional Integrity	Layer 0 preserved across operations

Table 14: Properties verified for each fork branch.

15.4 Check 3: Behavioral Equivalence Testing (Gap 5)

Critical Safety Check

Hash equivalence alone does not guarantee behavioral equivalence. Two branches with identical Layer 0 can interpret Layer 1+ differently, producing divergent safety behaviors.

15.4.1 The Problem

```
Both branches: Layer 0 says "Reflex Supremacy"

Branch A: Arbiter interprets as "halt capsule immediately"
Branch B: Arbiter interprets as "escalate to Judicial first"

Hash comparison: PASS (identical Layer 0)
Reality: Completely different safety behavior
```

15.4.2 Solution: Canonical Scenario Testing

Listing 3: Behavioral equivalence checker

```python
class BehavioralEquivalenceChecker:
    """Verify branches behave equivalently on canonical scenarios."""

    def check_equivalence(self, branch_a: Kernel,
                          branch_b: Kernel) -> EquivalenceResult:
        scenarios = self._load_canonical_scenarios()
        divergences = []

        for scenario in scenarios:
            outcome_a = self._execute_scenario(branch_a, scenario)
            outcome_b = self._execute_scenario(branch_b, scenario)

            if not self._outcomes_equivalent(outcome_a, outcome_b):
                divergences.append({
                    'scenario_id': scenario.id,
                    'scenario_name': scenario.name,
                    'branch_a': outcome_a,
                    'branch_b': outcome_b,
                    'severity': self._assess_severity(outcome_a, outcome_b)
```

```
                })

        return EquivalenceResult(
            equivalent=len(divergences) == 0,
            divergences=divergences,
            recommendation='APPROVE' if not divergences else 'REJECT'
        )

    def _outcomes_equivalent(self, a: Outcome, b: Outcome) -> bool:
        """Check semantic equivalence with tolerance."""
        # Decision must match exactly
        if a.decision != b.decision:
            return False

        # Timing within 5% tolerance
        if abs(a.latency - b.latency) > a.latency * 0.05:
            return False

        return True
```

15.4.3 Canonical Test Scenarios

Listing 4: Canonical scenarios for behavioral testing

```
CANONICAL_SCENARIOS = [
    # Scenario 1: Spawn Authorization Edge Case
    Scenario(
        id='spawn-001',
        name='Capsule at threshold attempts spawn',
        setup=Setup(capsule_health=0.65, depth=9, spawn_rate=85),
        action=Action('attempt_spawn'),
        expected='PERMIT_or_DENY_consistently'  # Must match between branches
    ),

    # Scenario 2: Health Degradation Response
    Scenario(
        id='health-001',
        name='Rapid health drop',
        setup=Setup(initial_health=0.8),
        action=Action('degrade_health', to=0.5, over_ticks=5000),
        expected='ASG_responds_within_10K_ticks'
    ),

    # Scenario 3: Reflex Termination
    Scenario(
        id='reflex-001',
        name='Unsafe action halted',
        setup=Setup(capsule_state='normal'),
        action=Action('attempt_unsafe_operation'),
        expected='HALT_within_1_tick'
    ),

    # Scenario 4: Judicial Appeal Resolution
    Scenario(
        id='judicial-001',
        name='Appeal consistency',
        setup=Setup(precedent_exists=True),
        action=Action('file_judicial_appeal'),
        expected='DECISION_matches_precedent'
    ),

    # Scenario 5: Capsule Liveness Under Load
```

```
    Scenario(
        id='liveness-001',
        name='Responsiveness under stress',
        setup=Setup(concurrent_capsules=100),
        action=Action('check_all_responsive', after_ticks=1000),
        expected='ALL_capsules_respond'
    ),

    # Scenario 6: Arbiter Failover
    Scenario(
        id='arbiter-001',
        name='Arbiter failure handling',
        setup=Setup(arbiters=10),
        action=Action('fail_arbiters', count=3),
        expected='ESCALATIONS_still_processed'
    ),

    # Scenario 7: Constitutional Query
    Scenario(
        id='const-001',
        name='Layer 0 query response',
        setup=Setup(),
        action=Action('query_commandment', id='REFLEX_SUPREMACY'),
        expected='IDENTICAL_response'
    ),

    # Scenario 8: Depth Limit Enforcement
    Scenario(
        id='depth-001',
        name='Spawn at max depth',
        setup=Setup(capsule_depth=10, D_max=10),
        action=Action('attempt_spawn'),
        expected='DENY_spawn'
    ),
]
```

15.4.4 Divergence Severity Classification

Listing 5: Severity assessment

```
def _assess_severity(self, outcome_a: Outcome, outcome_b: Outcome) -> str:
    """Classify divergence severity."""

    # CRITICAL: Safety decisions differ
    if outcome_a.decision != outcome_b.decision:
        if outcome_a.decision in ['HALT', 'DENY', 'REJECT']:
            return 'CRITICAL'  # One branch permissive, one restrictive
        return 'HIGH'

    # HIGH: Significant performance difference
    if abs(outcome_a.latency - outcome_b.latency) > 100:  # ms
        return 'HIGH'

    # MEDIUM: Minor behavioral difference
    return 'MEDIUM'
```

15.5 Check 4: Performance Regression Detection (Gap 4)

Operational Safety

A fork that passes all safety checks but is 10x slower is operationally unusable. Performance regression detection prevents such issues.

15.5.1 The Problem

```
Branch A (parent): P6 passes, avg response = 50ms
Branch B (fork):   P6 passes, avg response = 500ms

Safety verification: PASS
Reality: Branch B is unusable in production
```

15.5.2 Solution: Performance Baseline Comparison

Listing 6: Performance regression detector

```python
class PerformanceRegressionDetector:
    """Detect performance regressions in forked branches."""

    METRICS = [
        'spawn_latency',
        'arbiter_decision_time',
        'health_update_time',
        'escalation_processing_time',
        'dctm_write_latency',
        'zk_proof_generation_time'
    ]

    REGRESSION_THRESHOLD = 0.20  # 20% slowdown triggers warning
    CRITICAL_THRESHOLD = 0.50   # 50% slowdown triggers rejection

    def check_performance(self, parent: Kernel, branch_a: Kernel,
                          branch_b: Kernel) -> PerformanceReport:
        baseline = self._measure_baseline(parent)
        perf_a = self._measure_performance(branch_a)
        perf_b = self._measure_performance(branch_b)

        warnings = []
        critical = []

        for metric in self.METRICS:
            base_val = baseline[metric]

            for branch_name, perf in [('A', perf_a), ('B', perf_b)]:
                branch_val = perf[metric]
                regression = (branch_val - base_val) / base_val

                if regression > self.CRITICAL_THRESHOLD:
                    critical.append({
                        'branch': branch_name,
                        'metric': metric,
                        'baseline': base_val,
                        'actual': branch_val,
                        'regression_pct': regression * 100
                    })
                elif regression > self.REGRESSION_THRESHOLD:
                    warnings.append({
```

```
                        'branch': branch_name,
                        'metric': metric,
                        'baseline': base_val,
                        'actual': branch_val,
                        'regression_pct': regression * 100
                    })

        return PerformanceReport(
            passed=len(critical) == 0,
            warnings=warnings,
            critical=critical,
            recommendation=self._compute_recommendation(warnings, critical)
        )

    def _measure_baseline(self, kernel: Kernel) -> dict:
        """Measure performance metrics under standard load."""
        results = {}

        # Run standardized benchmark suite
        for metric in self.METRICS:
            results[metric] = self._run_benchmark(kernel, metric)

        return results

    def _run_benchmark(self, kernel: Kernel, metric: str) -> float:
        """Execute specific benchmark and return avg latency in ms."""
        iterations = 100
        total_time = 0

        for _ in range(iterations):
            start = time.perf_counter()

            if metric == 'spawn_latency':
                kernel.attempt_spawn(test_capsule)
            elif metric == 'arbiter_decision_time':
                kernel.arbiter.decide(test_escalation)
            elif metric == 'health_update_time':
                kernel.update_health(test_capsule, 0.75)
            # ... other metrics

            total_time += time.perf_counter() - start

        return (total_time / iterations) * 1000  # Convert to ms
```

15.5.3 Performance Thresholds by Profile

Metric	SANDBOX	PRODUCTION	CONTESTED
spawn_latency	< 100ms	< 50ms	< 20ms
arbiter_decision	< 200ms	< 100ms	< 50ms
health_update	< 50ms	< 20ms	< 10ms
escalation_proc	< 500ms	< 200ms	< 100ms
dctm_write	< 100ms	< 50ms	< 30ms
zk_proof_gen	< 1000ms	< 500ms	< 200ms

Table 15: Performance targets by deployment profile.

15.6 Verification Workflow

Listing 7: Complete verification workflow

```
def verify_fork_complete(parent: Kernel, branch_a: Kernel,
                         branch_b: Kernel) -> VerificationReport:
    """
    Complete fork verification workflow.
    All checks must pass for APPROVE recommendation.
    """
    verifier = ForkVerifier()
    behavioral = BehavioralEquivalenceChecker()
    performance = PerformanceRegressionDetector()

    report = VerificationReport()

    # Phase 1: Commandment preservation (MUST PASS)
    cmd_a = verifier._verify_commandments(parent, branch_a)
    cmd_b = verifier._verify_commandments(parent, branch_b)
    report.add_check('commandments_a', cmd_a)
    report.add_check('commandments_b', cmd_b)

    if not (cmd_a.passed and cmd_b.passed):
        report.recommendation = 'REJECT'
        report.reason = 'Layer 0 commandment violation'
        return report  # Early exit

    # Phase 2: Property testing (MUST PASS)
    props_a = verifier._verify_properties(branch_a)
    props_b = verifier._verify_properties(branch_b)
    report.add_check('properties_a', props_a)
    report.add_check('properties_b', props_b)

    if not (props_a.passed and props_b.passed):
        report.recommendation = 'REJECT'
        report.reason = 'P1-P8 property violation'
        return report

    # Phase 3: Behavioral equivalence (Gap 5) (MUST PASS)
    equiv = behavioral.check_equivalence(branch_a, branch_b)
    report.add_check('behavioral_equivalence', equiv)

    if not equiv.equivalent:
        report.recommendation = 'REJECT'
        report.reason = f'Behavioral divergence: {len(equiv.divergences)}
            scenarios'
        return report

    # Phase 4: Performance regression (Gap 4) (WARNINGS OK)
    perf = performance.check_performance(parent, branch_a, branch_b)
    report.add_check('performance', perf)

    if not perf.passed:
        report.recommendation = 'REJECT'
        report.reason = 'Critical performance regression'
        return report

    if perf.warnings:
        report.recommendation = 'APPROVE_WITH_WARNINGS'
        report.warnings = perf.warnings
    else:
        report.recommendation = 'APPROVE'

    return report
```

15.7 Fork Verification Test Cases

#	Test	Validates
1	Commandment hash match	Both branches preserve Layer 0
2	P1-P8 independent pass	Each branch satisfies all properties
3	Spawn decision equivalence	Same spawn decisions on edge cases
4	Health response equivalence	Same ASG behavior
5	Reflex termination equivalence	Same halt behavior
6	Judicial appeal equivalence	Same precedent handling
7	Liveness equivalence	Same responsiveness
8	Arbiter failover equivalence	Same fault tolerance
9	Performance baseline	No critical regression
10	Performance warning	Detect 20%+ slowdown
11	Complete workflow	End-to-end verification

Table 16: Fork verification test suite.

16 Cross-References

Related Component	Reference
Layer 0 (Vault)	Volume I §2
Layer 0.5 (Reflex-Core)	Volume I §3
Layers 1–5	Volume II §1–2
Arbiter Layer	Volume II §2
Genesis Block	Volume II §3.3
ZK-SP Proofs	Appendix E
Reflex Escalation	Appendix F
Gardener Interface	Appendix G
Deployment Profiles	Appendix I
Forensic State	Appendix A
Adaptive Spawn Governance	Appendix N
Fork Safety Verification	§14 (this document)

Table 17: Cross-references to other Codex components.

Changelog

v1.7 (December 2025) — *Fork Safety Verification*

- **Gap 4 (Added):** Performance regression detection (§14.5)
- **Gap 5 (Added):** Behavioral equivalence testing (§14.4)
- New section: Complete fork verification workflow (§14)
- 11 fork verification test cases specified
- Canonical scenario suite for behavioral testing

v1.6 (December 2025)

- Constitutional Health Monitor added
- Level 6 Design Principles documented
- Enhanced ZK-SP anchoring specifications

v1.5 (December 2025)

- Rollback mechanics detailed
- Profile integration expanded
- Gardener override authority specified

— End of Appendix J —