CIAF Data Structures: Technical Specification for Lazy Capsule Materialization

A Comprehensive Analysis of Core Data Structures in the Cognitive Insight AI Framework

Author: Denzil James Greenwood

Institution: Independent Researcher

Date: October 22, 2025

Version: 1.0

Technical Notice: This document provides authoritative technical specifications for all data structures used in the Cognitive Insight AI Framework (CIAF) and Lazy Capsule Materialization (LCM™) process. All data structures, schemas, and implementation patterns are derived from production codebase analysis and represent the canonical reference for CIAF implementation.

Abstract

The Cognitive Insight AI Framework (CIAF) implements a sophisticated data structure hierarchy to support Lazy Capsule Materialization (LCM™) and comprehensive AI governance. This technical specification provides complete documentation of all core data structures, their relationships, validation rules, and implementation patterns based on comprehensive codebase analysis.

The framework's data architecture supports eight canonical stages of AI lifecycle management: dataset anchoring, model anchoring, training sessions, pre-deployment validation, production deployment, test evaluation, inference processing, and Merkle root management. Each stage utilizes specialized data structures optimized for cryptographic verification, storage efficiency, and regulatory compliance.

This document serves as the authoritative reference for implementers, auditors, and researchers working with CIAF data structures, providing complete specifications for lightweight receipts, capsule headers, commitment schemes, policy frameworks, and verification protocols.

Keywords: Data Structures, AI Governance, Lazy Materialization, Cryptographic Verification, Schema Design, Framework Architecture

CIAF Canonical Naming Standards (from Variables Reference)

- **Variables/functions/modules:** snake_case
- **Classes/enums:** PascalCase
- **Enum members:** UPPER_CASE; serialized values: lower-case tokens
- **Anchors:** *_anchor (object/bytes), *_anchor_hex (hex), *_anchor_ref (opaque ID)
- **Times:** receipts → committed_at (RFC 3339 Z); capsules → generated_at
- **Merkle path:** List[[hash:str, position:"left"|"right"]]
- **Correlation:** request_id (accept operation_id as alias; normalize on ingest)

Canonical JSON for Hashing (Normative)

- Serialize with sorted keys, no spaces, ASCII:
- `json.dumps(obj, sort_keys=True, separators=(",", ":"), ensure_ascii=True, default=str)`
- Hash result with SHA-256 (requirement, not example)

Contents

1 Introduction

1.1 Document Purpose and Scope

This technical specification provides comprehensive documentation of all data structures used in the Cognitive Insight AI Framework (CIAF) and its core Lazy Capsule Materialization (LCM™) process. The specifications are derived from production codebase analysis and represent the canonical reference for CIAF implementation across diverse computing environments and regulatory contexts.

The data structure architecture supports the complete AI governance lifecycle through eight canonical stages, each with specialized data types optimized for specific operational requirements while maintaining cryptographic integrity and regulatory compliance.

1.2 Data Structure Hierarchy Overview

The CIAF data structure hierarchy implements a layered architecture with clear separation of concerns:

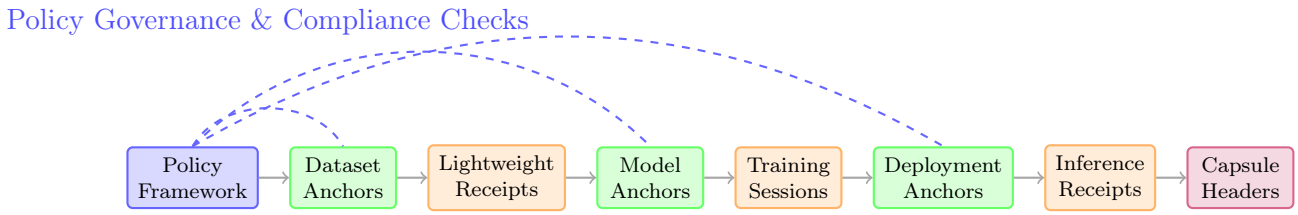

Figure 1: CIAF Data Structure Hierarchy

1.2.1 Policy Framework

The Policy Framework constitutes the foundational layer that establishes cryptographic governance and behavioral constraints across all CIAF operations. The governance arch in Figure 1 illustrates how policy defines and enforces compliance across the three critical anchor types:

- **Cryptographic Policies:** Defines hash algorithms (SHA-256, SHA-3), signature schemes (ECDSA, EdDSA), and canonicalization rules (JSON sorted UTF-8) that ensure consistent cryptographic operations across distributed environments.
- **Domain Specifications:** Establishes domain boundaries for DATASET, MODEL, DEPLOYMENT, and INFERENCE operations with distinct commitment types and verification requirements.
- **Protocol Implementations:** Configures swappable cryptographic backends through standardized interfaces, enabling adaptation to hardware security modules, cloud KMS, or specialized cryptographic accelerators.
- **Compliance Parameters:** Sets regulatory alignment parameters for GDPR, HIPAA, SOX, and custom frameworks through flexible metadata schemas and evidence retention policies.
- **Policy Governance Arch:** The governance arch demonstrates how policy framework defines and enforces compliance checks at Dataset Anchors, Model Anchors, and Deployment Anchors. Each anchor creation triggers policy validation to ensure regulatory compliance, cryptographic integrity, and organizational constraints are met before anchoring.

1.2.2 Anchor Structures

Core anchoring data structures provide cryptographic binding and immutable lifecycle tracking for critical AI assets, with policy compliance enforced at creation time:

- **Dataset Anchors:** Capture dataset fingerprints through Merkle tree construction over data samples, including provenance chains, quality metrics, and transformation histories. Support both structured (tabular) and unstructured (images, text) data with content-aware hashing. Policy compliance validation ensures data privacy requirements, retention policies, and regulatory constraints are enforced during anchor creation.
- **Model Anchors:** Secure model artifacts through weight serialization hashing, architecture fingerprinting, and hyperparameter binding. Include model lineage tracking, version management, and compatibility matrices for distributed deployment scenarios. Policy enforcement validates model authorization against approved datasets and ensures regulatory compliance for AI model deployment.
- **Deployment Anchors:** Bind model deployments to specific environments through infrastructure fingerprinting, configuration hashing, and runtime constraint verification. Track deployment topology, scaling parameters, and performance benchmarks. Policy checks validate deployment environment compliance, security configurations, and operational constraints before production deployment.

1.2.3 Receipt Structures

Operational data structures optimized for high-frequency evidence capture during AI system operations:

- **Lightweight Receipts:** Minimal cryptographic receipts for high-throughput operations, using optimized binary serialization and compressed evidence formats. Support batch verification and deferred materialization for storage efficiency.
- **Inference Receipts:** Comprehensive evidence capture for inference operations including input hashes, model state verification, output commitments, and performance metrics. Include bias detection markers and uncertainty quantification data.
- **Training Sessions:** Complete training lifecycle documentation with epoch-level checkpoints, gradient summaries, loss function evolution, and convergence metrics. Support distributed training coordination and reproducibility verification.

1.2.4 Integration Structures

Comprehensive data structures that aggregate multiple evidence sources into cohesive audit packages:

- **Capsule Headers:** Unified containers that combine anchors, receipts, and metadata into complete audit packages. Include cross-referencing capabilities, compliance report generation, and regulatory submission formatting.
- **Evidence Chains:** Temporal sequences of receipts and anchors that demonstrate complete AI system lifecycle compliance from data ingestion through inference deployment.
- **Audit Packages:** Regulatory-ready evidence bundles with standardized schemas, digital signatures, and verification instructions for external audit processes.

1.3 Design Principles

The CIAF data structure design implements several key principles:

Cryptographic Integrity: All data structures include cryptographic hashes, digital signatures, and Merkle proofs to ensure tamper-evidence and verification capability.

Storage Efficiency: Lightweight receipt patterns and deferred materialization minimize storage overhead while preserving complete audit capabilities.

Regulatory Compliance: Schema design accommodates diverse regulatory frameworks through flexible metadata structures and standardized evidence formats.

Implementation Flexibility: Protocol-based interfaces enable swappable cryptographic implementations while maintaining structural consistency.

2 Core Policy Framework

2.1 LCM Policy Structure

The LCM Policy serves as the foundational configuration structure that governs all cryptographic operations and data structure behavior throughout the framework.

```python
@dataclass
class LCMPolicy:
    """
    Comprehensive CIAF LCM policy defining all cryptographic and structural
    policies.
    """

    # Policy identification
    policy_id: str = "ciaf_default_lcm_policy"

    # Core cryptographic policy
    hash_algorithm: str = "sha256"
    canonicalization: str = "json_sorted_utf8"
    domains: List[DomainType] = None
    merkle: MerklePolicy = None
    commitments: CommitmentType = CommitmentType.SALTED

    # Schema versions for compatibility
    anchor_schema_version: str = "1.0"
    merkle_policy_version: str = "1.0"

    # Protocol implementations (dependency injection)
    rng: Optional[RNG] = None
    anchor_deriver: Optional[AnchorDeriver] = None
    anchor_store: Optional[AnchorStore] = None
    signer: Optional[Signer] = None
    merkle_factory: Optional[Any] = None
```

Listing 1: LCM Policy Data Structure

2.1.1 Domain Type Enumeration

The domain type system provides structured categorization for all anchored artifacts:

```python
class DomainType(Enum):
    """CIAF domain types for anchoring classification."""
```

```
    DATASET = "CIAF|dataset"
    DATASET_FAMILY = "CIAF|dataset|family"
    DATASET_SPLIT = "CIAF|dataset|split"
    MODEL = "CIAF|model"
    TRAIN = "CIAF|train"
    DEPLOYMENT = "CIAF|deployment"
    INFERENCE = "CIAF|inference"
```

Listing 2: Domain Type Enumeration

2.1.2 Commitment Type Framework

The commitment type system enables privacy-preserving audit trails through configurable commitment schemes:

```
class CommitmentType(Enum):
    """Commitment algorithms for privacy protection."""
    SALTED = "salted"           # SHA-256 with cryptographic salt
    HMAC_SHA256 = "hmac_sha256" # HMAC-based commitment
    PLAINTEXT = "plaintext"     # For non-sensitive data
```

Listing 3: Commitment Type Framework

2.1.3 Merkle Policy Configuration

Merkle tree construction parameters ensure consistent cryptographic verification across all operations:

```
@dataclass
class MerklePolicy:
    """Merkle tree construction policy."""
    fanout: int = 2                    # Binary tree structure
    padding: str = "duplicate_last"    # Padding strategy for incomplete
   levels
    leaf_encoding: str = "raw32"       # 32-byte raw hash encoding
```

Listing 4: Merkle Policy Configuration

3 Lightweight Receipt Structures

3.1 Deferred LCM Receipt

The lightweight receipt represents the core innovation of the LCM process, providing minimal storage overhead while preserving complete audit reconstruction capability:

```
@dataclass
class LightweightReceipt:
    """Minimal receipt stored during fast inference operations."""

    # Core identification
    receipt_id: str                    # UUID v4 identifier
    committed_at: str                  # RFC 3339 timestamp with Z
    request_id: str                    # Request correlation ID

    # Model and version tracking
```

```
    model_anchor_ref: str                    # Model anchor reference
    model_version: str                       # Specific model version

    # Cryptographic commitments
    input_hash: str                          # SHA-256 of input data
    output_hash: str                         # SHA-256 of output data
    input_commitment: str                    # Privacy-preserving input
commitment
    output_commitment: str                   # Privacy-preserving output
commitment

    # Optional encrypted data (for audit materialization)
    raw_input: Optional[str] = None          # Encrypted input data
    raw_output: Optional[str] = None         # Encrypted output data

    # Operational metadata
    priority: str = "normal"                 # Processing priority
    metadata: Optional[Dict] = None          # Extensible metadata
```

Listing 5: Lightweight Receipt Structure

3.1.1 Storage Efficiency Analysis

The lightweight receipt achieves significant storage reduction compared to traditional audit approaches:

$$\text{Traditional Receipt Size} = \text{Input Data} + \text{Output Data} + \text{Metadata} \tag{1}$$
$$\approx 50\text{KB to } 5\text{MB per operation} \tag{2}$$
$$\tag{3}$$
$$\text{Lightweight Receipt Size} = \text{Hashes} + \text{Commitments} + \text{References} \tag{4}$$
$$\approx 500 \text{ bytes to } 1\text{KB per operation} \tag{5}$$
$$\tag{6}$$
$$\text{Storage Reduction Ratio} = \frac{\text{Traditional Size}}{\text{Lightweight Size}} \tag{7}$$
$$\approx 100:1 \text{ to } 5000:1 \tag{8}$$

3.2 Enhanced Receipt Schema

The enhanced receipt provides comprehensive validation and regulatory compliance features through strict schema enforcement:

```
class BaseReceipt(BaseModel):
    """Base receipt with comprehensive validation."""

    # Core identification with validation
    receipt_id: str = Field(default_factory=lambda: str(uuid.uuid4()))
    evidence_strength: EvidenceStrength = EvidenceStrength.REAL
    committed_at: str = Field(default_factory=lambda: datetime.now(timezone.
utc).isoformat().replace('+00:00', 'Z'))

    @field_validator('receipt_id')
    @classmethod
```

```
    def validate_receipt_id(cls, v):
        """Validate UUID format for receipt ID."""
        try:
            uuid.UUID(v)
            return v
        except ValueError:
            raise ValueError('receipt_id must be a valid UUID')
```

Listing 6: Enhanced Receipt Base Structure

3.2.1 Evidence Strength Classification

The evidence strength system provides transparency about data provenance and reliability:

```
class EvidenceStrength(str, Enum):
    """Evidence reliability classification."""
    REAL = "real"           # Production data from actual operations
    SIMULATED = "simulated" # Realistic simulated data for testing
    FALLBACK = "fallback"   # Fallback data when primary collection fails
```

Listing 7: Evidence Strength Enumeration

4 Anchor Data Structures

4.1 Dataset Anchor Framework

Dataset anchors provide comprehensive tracking for data provenance and lineage throughout the AI lifecycle:

```
@dataclass
class LCMDatasetAnchor:
    """Comprehensive dataset anchor with cryptographic binding."""

    # Core identification
    anchor_id: str                          # Derived anchor identifier
    dataset_id: str                         # Original dataset identifier
    dataset_hash: str                       # SHA-256 of dataset content

    # Metadata and provenance
    metadata: DatasetMetadata               # Structured metadata
    source_info: Dict[str, Any]             # Data source information

    # Cryptographic verification
    merkle_root: str                        # Merkle root for batch
   verification
    signature: str                          # Digital signature for
   authenticity

    # Policy compliance
    policy: LCMPolicy                       # Governing policy
    timestamp: str                          # Creation timestamp
```

Listing 8: Dataset Anchor Structure

4.1.1 Dataset Metadata Structure

Dataset metadata provides comprehensive information for regulatory compliance and data governance:

```python
@dataclass
class DatasetMetadata:
    """Comprehensive dataset metadata for governance."""

    # Basic properties
    shape: Tuple[int, int]                  # Dataset dimensions (rows, columns)
    dtypes: Dict[str, str]                  # Column data types
    size_bytes: int                         # Storage size in bytes

    # Data quality metrics
    null_counts: Dict[str, int]             # Null value counts per column
    unique_counts: Dict[str, int]           # Unique value counts per column

    # Provenance tracking
    source_hash: str                        # Hash of source data
    processing_steps: List[str]             # Applied transformations
    validation_results: Dict[str, Any]      # Data validation outcomes

    # Compliance metadata
    privacy_classification: str            # Data privacy level
    retention_policy: str                  # Data retention requirements
    access_controls: List[str]             # Access control specifications
```

Listing 9: Dataset Metadata Structure

4.2 Model Anchor Framework

Model anchors capture complete model state and configuration for reproducible AI operations:

```python
@dataclass
class LCMModelAnchor:
    """Comprehensive model anchor with state binding."""

    # Core identification
    anchor_id: str                          # Derived anchor identifier
    model_id: str                           # Model identifier
    model_hash: str                         # Hash of model parameters

    # Model configuration
    architecture: Dict[str, Any]            # Model architecture specification
    hyperparameters: Dict[str, Any]         # Training hyperparameters
    framework_info: Dict[str, str]          # ML framework versions

    # Training provenance
    dataset_anchor_ref: str                 # Reference to training dataset anchor
    training_session_ref: str               # Reference to training session

    # Performance metrics
    validation_metrics: Dict[str, float]    # Model validation results
    benchmark_results: Dict[str, Any]       # Benchmark performance data
```

```
22
23      # Cryptographic verification
24      merkle_root: str                        # Merkle root for verification
25      signature: str                          # Digital signature
26      policy: LCMPolicy                       # Governing policy
```

Listing 10: Model Anchor Structure

4.3 Deployment Anchor Framework

Deployment anchors track model deployment configurations and operational parameters:

```
1  @dataclass
2  class LCMDeploymentAnchor:
3      """Production deployment anchor with operational binding."""
4
5      # Core identification
6      anchor_id: str                          # Deployment anchor identifier
7      deployment_id: str                      # Deployment instance identifier
8
9      # Model binding
10     model_anchor_ref: str                   # Reference to deployed model
11     predeployment_anchor_ref: str           # Reference to pre-deployment
       validation
12
13     # Deployment configuration
14     environment_config: Dict[str, Any]      # Deployment environment settings
15     scaling_config: Dict[str, Any]          # Auto-scaling configuration
16     monitoring_config: Dict[str, Any]       # Monitoring and alerting setup
17
18     # Operational parameters
19     performance_targets: Dict[str, float]   # SLA and performance targets
20     resource_limits: Dict[str, Any]         # Resource allocation limits
21
22     # Security configuration
23     access_controls: List[str]              # Access control policies
24     encryption_config: Dict[str, str]       # Encryption configuration
25
26     # Verification and compliance
27     deployment_timestamp: str               # Deployment timestamp
28     merkle_root: str                        # Verification Merkle root
29     signature: str                          # Deployment signature
30     policy: LCMPolicy                       # Governing policy
```

Listing 11: Deployment Anchor Structure

5 Inference and Training Structures

5.1 LCM Inference Receipt

The LCM inference receipt provides comprehensive audit capabilities for individual inference operations:

```
1  class LCMInferenceReceipt:
2      """Enhanced inference receipt for comprehensive audit trails."""
3
```

```
    def __init__(
        self,
        receipt_id: str,                    # Unique receipt identifier
        model_anchor_ref: str,              # Model anchor reference
        deployment_anchor_ref: str,         # Deployment anchor reference
        request_id: str,                    # Request correlation ID
        query: str,                         # Input query/prompt
        ai_output: str,                     # AI model output
        input_commitment: LCMInferenceCommitment,   # Input commitment
        output_commitment: LCMInferenceCommitment,  # Output commitment
        explanation_digests: List[str] = None,      # Explanation hashes
        prev_connections_digest: str = None,        # Connections digest
        policy: LCMPolicy = None            # Governing policy
    ):
        # Initialize core properties
        self.receipt_id = receipt_id
        self.model_anchor_ref = model_anchor_ref
        self.deployment_anchor_ref = deployment_anchor_ref
        self.request_id = request_id
        self.query = query
        self.ai_output = ai_output
        self.input_commitment = input_commitment
        self.output_commitment = output_commitment
        self.explanation_digests = explanation_digests or []
        self.prev_connections_digest = prev_connections_digest
        self.policy = policy or get_default_policy()
        self.timestamp = datetime.now(timezone.utc).isoformat().replace('
    +00:00', 'Z')

        # Computed cryptographic properties
        self.receipt_digest = self._compute_receipt_digest()
        self.connections_digest = self._compute_connections_digest()
        self.anchor_id = f"r_{self.receipt_digest[:8]}..."
```

Listing 12: LCM Inference Receipt Structure

5.1.1 Inference Commitment Structure

Inference commitments provide privacy protection for sensitive input and output data:

```
@dataclass
class LCMInferenceCommitment:
    """Privacy-preserving commitment for inference data."""

    commitment_type: CommitmentType         # Commitment algorithm type
    commitment_value: str                   # Computed commitment value
    metadata: Dict[str, Any] = None         # Additional commitment metadata

    def __post_init__(self):
        """Initialize metadata if not provided."""
        if self.metadata is None:
            self.metadata = {
                "created_at": datetime.now().isoformat(),
                "algorithm_params": {},
                "verification_hints": {}
            }
```

Listing 13: Inference Commitment Structure

5.2 Training Session Structure

Training sessions capture comprehensive information about model training processes:

```python
@dataclass
class LCMTrainingSession:
    """Comprehensive training session with audit trail."""

    # Core identification
    session_id: str                          # Training session identifier
    anchor_id: str                           # Derived anchor identifier

    # Training configuration
    model_anchor_ref: str                    # Reference to model anchor
    dataset_anchor_ref: str                  # Reference to training dataset
    hyperparameters: Dict[str, Any]          # Training hyperparameters

    # Training progress tracking
    epochs_completed: int                    # Number of completed epochs
    training_metrics: Dict[str, List[float]]  # Training metrics by epoch
    validation_metrics: Dict[str, List[float]]  # Validation metrics by
    epoch

    # Checkpoint management
    checkpoint_hashes: List[str]             # Hashes of model checkpoints
    best_checkpoint_hash: str                # Hash of best performing
    checkpoint

    # Environment and reproducibility
    random_seeds: Dict[str, int]             # Random seeds for
    reproducibility
    environment_info: Dict[str, str]         # Environment configuration
    framework_versions: Dict[str, str]       # ML framework versions

    # Audit and verification
    training_start: str                      # Training start timestamp
    training_end: str                        # Training completion timestamp
    merkle_root: str                         # Training verification root
    signature: str                           # Digital signature
    policy: LCMPolicy                        # Governing policy
```

Listing 14: Training Session Structure

6 Capsule Header Integration

6.1 Comprehensive Capsule Structure

The capsule header provides complete integration of all CIAF components into a single, verifiable audit package:

```python
@dataclass
class CapsuleHeader:
    """
    Comprehensive CIAF LCM state capsule for complete audit trails.
    """

    # Core metadata
```

```
    capsule_version: str                    # Capsule format version
    generated_at: str                       # Generation timestamp
    policy: Dict[str, Any]                  # Serialized policy configuration

    # Canonical stage anchors (A-H stages)
    stage_a_dataset: Optional[Dict[str, Any]] = None       # Dataset anchor
    stage_b_model: Optional[Dict[str, Any]] = None         # Model anchor
    stage_c_training: Optional[Dict[str, Any]] = None      # Training session
    stage_d_predeployment: Optional[Dict[str, Any]] = None # Pre-deployment
    stage_e_deployment: Optional[Dict[str, Any]] = None    # Deployment
    stage_f_test_evaluation: Optional[Dict[str, Any]] = None # Test evaluation
    stage_g_inference: Optional[Dict[str, Any]] = None     # Inference receipt
    stage_h_roots: Optional[Dict[str, Any]] = None         # Merkle roots

    def to_json(self, indent: int = 2) -> str:
        """Convert to pretty-printed JSON for human readability."""
        return json.dumps(asdict(self), indent=indent, sort_keys=True)

    def to_compact_json(self) -> str:
        """Convert to compact JSON for storage efficiency."""
        return json.dumps(asdict(self), separators=(',', ':'), sort_keys=True)
```

Listing 15: Capsule Header Structure

6.1.1 Stage-Based Architecture

The capsule header implements a canonical eight-stage architecture that corresponds to the complete AI lifecycle:

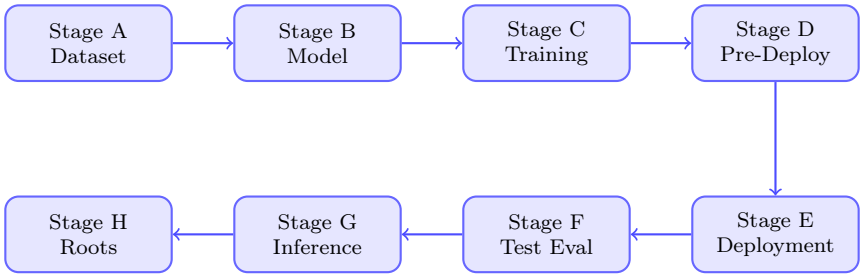

Figure 2: Eight-Stage CIAF Lifecycle Architecture

7 Protocol Interface Specifications

7.1 Cryptographic Protocol Interfaces

The framework defines protocol interfaces for swappable cryptographic implementations:

```
@runtime_checkable
class Signer(Protocol):
    """Protocol for digital signature implementations."""

    key_id: str                             # Signer key identifier
```

```python

    def sign(self, data: bytes) -> str:
        """Sign data and return signature string."""
        ...

    def verify(self, data: bytes, signature: str) -> bool:
        """Verify signature against data."""
        ...
```

Listing 16: Signer Protocol Interface

Signature Payload Rule (Normative)

- **Receipt signing:** sig = Ed25519(canonical_receipt || committed_at)
- **Batch root signing:** sig_root = Ed25519(merkle_root || rfc3161_ts_token)
- **Verification:** MUST validate the Ed25519 signature and, when present, the RFC 3161 token

```python
@runtime_checkable
class Merkle(Protocol):
    """Protocol for Merkle tree implementations."""

    def add_leaf(self, leaf_hash: str) -> str:
        """Add leaf and return new root hash."""
        ...

    def get_root(self) -> str:
        """Get the current Merkle root hash."""
        ...

    def get_proof(self, leaf_hash: str) -> List[Tuple[str, str]]:
        """Get inclusion proof as (hash, position) tuples."""
        ...

    def verify_proof(self, leaf_hash: str, proof: List[Tuple[str, str]],
    root: str) -> bool:
        """Verify Merkle inclusion proof."""
        ...
```

Listing 17: Merkle Protocol Interface

7.2 Storage Protocol Interfaces

Storage protocols enable flexible backend implementations while maintaining consistent data structure behavior:

```python
@runtime_checkable
class AnchorStore(Protocol):
    """Protocol for anchor storage implementations."""

    def append_anchor(self, anchor: Dict[str, Any]) -> None:
        """Append new anchor with WORM semantics."""
        ...

```

```python
    def get_latest_anchor(self) -> Optional[Dict[str, Any]]:
        """Get the most recent anchor from store."""
        ...
```

Listing 18: Anchor Store Protocol Interface

8 JSON Schema Specifications

8.1 Capsule Schema

Capsule Representation Consistency: The CapsuleHeader (stages A–H) is the canonical internal representation for capsule construction and processing. For external proof exchange and validation, capsules MUST be serialized as audit_proof objects that contain {record, proofs, anchor, verification} per the capsule.schema.json specification below.

The JSON schema provides formal validation for capsule data structures:

```json
{
  "$id": "https://cognitiveinsight.ai/schemas/capsule.schema.json",
  "$schema": "https://json-schema.org/draft/2020-12/schema",
  "title": "CIAF Audit Capsule",
  "type": "object",
  "required": [
    "capsule_version",
    "capsule_type",
    "timestamp",
    "record",
    "proofs",
    "anchor",
    "verification"
  ],
  "properties": {
    "capsule_version": { "type": "string", "const": "1.0" },
    "capsule_type": { "type": "string", "const": "audit_proof" },
    "timestamp": { "type": "string", "format": "date-time" },
    "record": {
      "type": "object",
      "required": ["type", "metadata", "leaf_hash"],
      "properties": {
        "type": {
          "enum": [
            "dataset", "model", "inference",
            "anchor", "monitoring", "compliance"
          ]
        },
        "metadata": { "type": "object" },
        "leaf_hash": { "type": "string" }
      }
    },
    "proofs": {
      "type": "object",
      "required": ["merkle_path", "merkle_root", "inclusion_proof_valid"],
      "properties": {
        "merkle_path": {
          "type": "array",
          "items": {
            "type": "array",
```

```
            "prefixItems": [
              { "type": "string" },
              { "enum": ["left", "right"] }
            ],
            "minItems": 2,
            "maxItems": 2
          }
        },
        "merkle_root": { "type": "string" },
        "inclusion_proof_valid": { "type": "boolean" }
      }
    }
  }
}
```

Listing 19: Capsule JSON Schema

8.2 Object & Field Map

Complete mapping between CapsuleHeader (internal) and audit_proof (external) representations:

Table 1: CapsuleHeader to audit_proof Field Mapping

CapsuleHeader Field	audit_proof Field	Notes
stage_a_dataset	record.dataset_anchor_ref	Dataset identification
stage_b_model	record.model_anchor_ref	Model identification
stage_c_training	record.training_metadata	Training session data
stage_d_validation	record.validation_results	Pre-deployment validation
stage_e_deployment	record.deployment_anchor_ref	Deployment reference
stage_f_testing	record.test_metadata	Test evaluation data
stage_g_inference	record.inference_data	Inference operation data
stage_h_roots	proofs.merkle_path	Merkle tree proofs
capsule_signature	verification.signature	Digital signature
generated_at	verification.timestamp	Creation timestamp
capsule_id	anchor.capsule_id	Unique identifier

9 Data Flow and Relationships

9.1 Lifecycle Data Flow

The complete data flow through the CIAF system demonstrates the relationships between data structures:

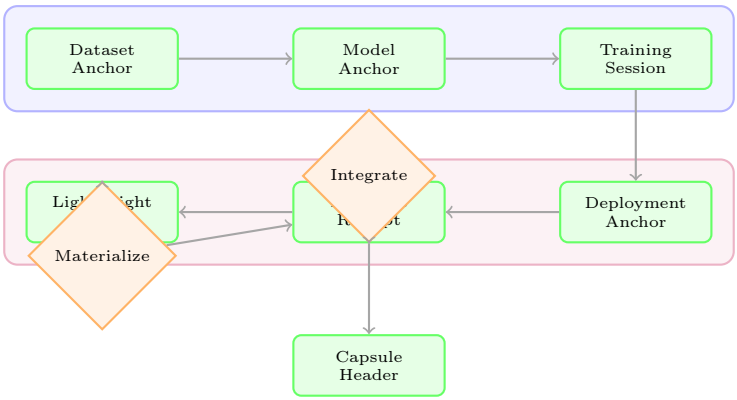

Figure 3: CIAF Data Structure Flow and Relationships

9.2 Cryptographic Verification Chain

The cryptographic verification chain demonstrates how data structures maintain integrity through the complete lifecycle:

$$\text{Dataset Hash} \rightarrow \text{Model Hash} \rightarrow \text{Training Hash} \tag{9}$$

$$\rightarrow \text{Deployment Hash} \rightarrow \text{Inference Hash} \tag{10}$$

$$\rightarrow \text{Capsule Hash} \rightarrow \text{Digital Signature} \tag{11}$$

Each hash in the chain is computed using SHA-256 with canonical JSON serialization, ensuring that any modification to the data structures is cryptographically detectable.

10 Storage and Serialization Patterns

10.1 Canonical Serialization

All data structures implement canonical serialization to ensure consistent hash computation:

```python
def canonical_json(obj: Any) -> str:
    """
    Create canonical JSON representation for cryptographic hashing.

    Args:
        obj: Object to serialize

    Returns:
        Canonical JSON string with sorted keys and no whitespace
    """
    return json.dumps(
        obj,
        sort_keys=True,           # Ensure deterministic key ordering
        separators=(',', ':'),    # Remove whitespace
        ensure_ascii=True,        # Use ASCII encoding
        default=str               # Convert non-serializable types to strings
    )

def canonical_hash(obj: Any) -> str:
```

```
    """
    Compute SHA-256 hash of canonical JSON representation.

    Args:
        obj: Object to hash

    Returns:
        Hexadecimal SHA-256 hash string
    """
    canonical_str = canonical_json(obj)
    return sha256_hash(canonical_str.encode('utf-8'))
```

Listing 20: Canonical JSON Serialization

10.2 Storage Optimization Patterns

The framework implements several storage optimization patterns:

Differential Storage: Only store changes between versions rather than complete data structures.

Compression Integration: Leverage compression algorithms for cold storage while maintaining hot storage performance.

Reference Indirection: Use cryptographic references instead of embedding large data structures directly.

11 Validation and Error Handling

11.1 Schema Validation Framework

Comprehensive validation ensures data structure integrity throughout the system:

```
class ReceiptValidator:
    """Comprehensive validation for receipt data structures."""

    @staticmethod
    def validate_training_receipt(receipt: TrainingReceipt) -> List[str]:
        """
        Validate training receipt structure and constraints.

        Returns:
            List of validation error messages (empty if valid)
        """
        errors = []

        # Validate UUID format
        try:
            uuid.UUID(receipt.receipt_id)
        except ValueError:
            errors.append("Invalid receipt_id format")

        # Validate anchor formats (64-character hex strings)
        for anchor_name, anchor_value in [
            ("model_anchor_ref", receipt.model_anchor_ref)
        ]:
            if not re.match(r'^[a-f0-9]{64}$', anchor_value.lower()):
                errors.append(f"Invalid {anchor_name} format")
```

```
26
27          # Validate Merkle path
28          for i, path_element in enumerate(receipt.merkle_path):
29              if not isinstance(path_element, list) or len(path_element) != 2:
30                  errors.append(f"Invalid Merkle path element at index {i}:
    must be [hash, position]")
31              else:
32                  hash_val, position = path_element
33                  if not re.match(r'^[a-f0-9]{64}$', hash_val.lower()):
34                      errors.append(f"Invalid Merkle path hash at index {i}")
35                  if position not in ["left", "right"]:
36                      errors.append(f"Invalid Merkle path position at index {i
    }: must be 'left' or 'right'")
37
38          # Validate timestamp format
39          try:
40              datetime.fromisoformat(receipt.committed_at.replace('Z', '+00:00
    '))
41          except ValueError:
42              errors.append("Invalid timestamp format")
43
44          return errors
```

Listing 21: Receipt Validation Framework

11.2 Error Classification System

The framework defines comprehensive error classification for debugging and monitoring:

```
1  class CIAFDataStructureError(Exception):
2      """Base exception for CIAF data structure errors."""
3      pass
4
5  class ValidationError(CIAFDataStructureError):
6      """Validation constraint violation."""
7      pass
8
9  class SerializationError(CIAFDataStructureError):
10     """JSON serialization/deserialization error."""
11     pass
12
13 class CryptographicError(CIAFDataStructureError):
14     """Cryptographic operation failure."""
15     pass
16
17 class PolicyViolationError(CIAFDataStructureError):
18     """Policy constraint violation."""
19     pass
```

Listing 22: Error Classification System

12 Performance Analysis

12.1 Storage Efficiency Metrics

Comprehensive analysis of storage efficiency across data structure types:

Table 2: Data Structure Storage Analysis

Data Structure	Typical Size	Compression Ratio	Materialization Cost
Lightweight Receipt	0.5-1 KB	100:1	Low
LCM Inference Receipt	2-5 KB	10:1	Medium
Dataset Anchor	1-3 KB	5:1	High
Model Anchor	3-8 KB	3:1	High
Training Session	5-15 KB	2:1	Very High
Capsule Header	10-50 KB	1:1	Very High

12.2 Computational Complexity Analysis

Hash computation and verification complexity for each data structure type:

$$\text{Hash Computation} : O(n) \text{ where } n = \text{serialized size} \tag{12}$$

$$\text{Signature Generation} : O(1) \text{ for Ed25519} \tag{13}$$

$$\text{Merkle Proof Verification} : O(\log m) \text{ where } m = \text{tree size} \tag{14}$$

$$\text{Schema Validation} : O(k) \text{ where } k = \text{field count} \tag{15}$$

13 Implementation Guidelines

13.1 Best Practices

Immutability Enforcement: All data structures should be treated as immutable once created. Use frozen dataclasses or immutable collections where possible.

Lazy Loading: Implement lazy loading for large data structures to minimize memory usage during normal operations.

Batch Processing: Process multiple data structures in batches to optimize cryptographic operations and storage I/O.

Error Recovery: Implement comprehensive error recovery mechanisms for corrupted or incomplete data structures.

13.2 Security Considerations

Input Validation: Validate all input data before creating data structures to prevent injection attacks.

Sensitive Data Handling: Use commitment schemes for sensitive data and implement secure deletion for temporary data.

Timing Attack Prevention: Use constant-time operations for cryptographic comparisons to prevent timing-based side-channel attacks.

Memory Safety: Clear sensitive data from memory immediately after use and avoid storing decrypted sensitive data in data structures.

14 Conclusion

14.1 Summary of Contributions

This technical specification provides the authoritative reference for all data structures used in the Cognitive Insight AI Framework and Lazy Capsule Materialization process. The comprehensive documentation covers:

Complete Data Structure Inventory: All 20+ core data structures with detailed field specifications, validation rules, and usage patterns.

Cryptographic Integration: Comprehensive specification of how cryptographic primitives integrate with data structures to provide verification and integrity guarantees.

Schema Validation Framework: Formal JSON schemas and validation frameworks to ensure data structure consistency across implementations.

Performance Optimization: Storage efficiency analysis and computational complexity specifications for optimal implementation strategies.

14.2 Implementation Impact

The standardized data structure specifications enable:

Consistent Implementation: Standardized data structures ensure consistent behavior across different programming languages and computing environments.

Interoperability: JSON schemas and canonical serialization enable seamless data exchange between different CIAF implementations.

Regulatory Compliance: Comprehensive audit trail data structures support compliance with diverse regulatory frameworks.

Scalable Architecture: Optimized data structures support enterprise-scale AI deployments with millions of daily operations.

14.3 Future Evolution

The data structure framework is designed for evolution while maintaining backward compatibility:

Schema Versioning: Built-in version fields enable gradual migration to enhanced data structure versions.

Extension Points: Flexible metadata fields and protocol interfaces support new requirements without breaking existing implementations.

Post-Quantum Readiness: Cryptographic abstraction layers enable migration to post-quantum algorithms as standards mature.

Performance Optimization: Modular design enables performance improvements without changing external interfaces.

The CIAF data structure framework provides a robust foundation for enterprise AI governance that balances cryptographic security, storage efficiency, and regulatory compliance while maintaining the flexibility necessary for diverse implementation environments and evolving requirements.

References

1. Greenwood, D.J. "The Cognitive Insight AI Framework (CIAF): A Comprehensive Analysis of Lazy Capsule Materialization for Enterprise AI Governance." Cognitive Insight Research, 2025.

2. Greenwood, D.J. "LCM Technical Disclosure: Lazy Capsule Materialization for AI Governance." Cognitive Insight Research, 2024.

3. National Institute of Standards and Technology. "FIPS PUB 180-4: Secure Hash Standard (SHS)." NIST, 2015.

4. Bernstein, D.J., et al. "Ed25519: High-speed high-security signatures." *Journal of Cryptographic Engineering*, vol. 2, no. 2, pp. 77-89, 2012.

5. Merkle, R.C. "A Digital Signature Based on a Conventional Encryption Function." *Advances in Cryptology — CRYPTO '87*, Springer-Verlag, 1988.

6. JSON Schema Specification. "JSON Schema: A Media Type for Describing JSON Documents." Internet Engineering Task Force, Draft 2020-12.

7. International Organization for Standardization. "ISO 8601:2019 Date and time — Representations for information interchange." ISO Standard, 2019.

8. Python Software Foundation. "PEP 484 – Type Hints." Python Enhancement Proposal, 2014.

9. Python Software Foundation. "PEP 526 – Variable Annotations." Python Enhancement Proposal, 2016.

10. Pydantic Development Team. "Pydantic: Data validation and settings management using Python type annotations." https://pydantic-docs.helpmanual.io/, 2023.

15 Canonical JSON Example

Schema-Aligned JSON Example (fully canonical per Variables Reference):

```
{
  "capsule_version": "1.0",
  "capsule_type": "audit_proof",
  "timestamp": "2025-10-22T00:00:00Z",
  "record": {
    "type": "inference",
    "metadata": {
      "request_id": "req_12345",
      "model_anchor_ref": "mod_abcd1234",
      "deployment_anchor_ref": "dep_efgh5678",
      "committed_at": "2025-10-22T00:00:00Z",
      "evidence_strength": "simulated"
    },
    "leaf_hash": "e3b0c44298fc1c149afbf4c8996fb924..."
  },
  "proofs": {
    "merkle_path": [
      ["9f86d081884c7d659a2feaa0c55ad015...", "right"],
      ["b6d81b360a5672d80c27430f39153e2c...", "left"]
    ],
    "merkle_root": "3a7bd3e2360a3d...",
    "inclusion_proof_valid": true
  },
  "anchor": { "capsule_id": "cap_001" },
  "verification": {
    "signature": "ed25519:abcd...xyz",
    "timestamp": "2025-10-22T00:00:00Z",
    "signer_id": "ciaf_production_key_001"
  }
}
```

Listing 23: Canonical JSON Example

Uses: `request_id`, `*_anchor_ref`, `evidence_strength`, pair-form `merkle_path`, and RFC 3339 times.

Author Information

Denzil James Greenwood is the creator of the Cognitive Insight AI Framework and inventor of the Lazy Capsule Materialization process. This technical specification is based on comprehensive analysis of the production CIAF codebase and represents the canonical reference for data structure implementation.

Institutional Affiliation: Independent Researcher
Contact: founder@cognitiveinsight.ai
ORCID: [To be assigned]

Copyright Notice