

# 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

---

<b>1</b>	<b>Introduction</b>	<b>4</b>
1.1	Document Purpose and Scope . . . . .	4
1.2	Data Structure Hierarchy Overview . . . . .	4
1.2.1	Policy Framework . . . . .	4
1.2.2	Anchor Structures . . . . .	5
1.2.3	Receipt Structures . . . . .	5
1.2.4	Integration Structures . . . . .	6
1.3	Design Principles . . . . .	6
<b>2</b>	<b>Core Policy Framework</b>	<b>6</b>
2.1	LCM Policy Structure . . . . .	6
2.1.1	Domain Type Enumeration . . . . .	7
2.1.2	Commitment Type Framework . . . . .	7
2.1.3	Merkle Policy Configuration . . . . .	7
<b>3</b>	<b>Lightweight Receipt Structures</b>	<b>8</b>
3.1	Deferred LCM Receipt . . . . .	8
3.1.1	Storage Efficiency Analysis . . . . .	8
3.2	Enhanced Receipt Schema . . . . .	9
3.2.1	Evidence Strength Classification . . . . .	9
<b>4</b>	<b>Anchor Data Structures</b>	<b>10</b>
4.1	Dataset Anchor Framework . . . . .	10
4.1.1	Dataset Metadata Structure . . . . .	10
4.2	Model Anchor Framework . . . . .	11
4.3	Deployment Anchor Framework . . . . .	12
<b>5</b>	<b>Inference and Training Structures</b>	<b>12</b>
5.1	LCM Inference Receipt . . . . .	12
5.1.1	Inference Commitment Structure . . . . .	13
5.2	Training Session Structure . . . . .	14
<b>6</b>	<b>Capsule Header Integration</b>	<b>15</b>
6.1	Comprehensive Capsule Structure . . . . .	15
6.1.1	Stage-Based Architecture . . . . .	16
<b>7</b>	<b>Protocol Interface Specifications</b>	<b>16</b>
7.1	Cryptographic Protocol Interfaces . . . . .	16
7.2	Storage Protocol Interfaces . . . . .	17
<b>8</b>	<b>JSON Schema Specifications</b>	<b>17</b>
8.1	Capsule Schema . . . . .	17
8.2	Object & Field Map . . . . .	19
<b>9</b>	<b>Data Flow and Relationships</b>	<b>19</b>

9.1	Lifecycle Data Flow . . . . .	19
9.2	Cryptographic Verification Chain . . . . .	20
<b>10</b>	<b>Storage and Serialization Patterns</b>	<b>21</b>
10.1	Canonical Serialization . . . . .	21
10.2	Storage Optimization Patterns . . . . .	21
<b>11</b>	<b>Validation and Error Handling</b>	<b>22</b>
11.1	Schema Validation Framework . . . . .	22
11.2	Error Classification System . . . . .	23
<b>12</b>	<b>Performance Analysis</b>	<b>23</b>
12.1	Storage Efficiency Metrics . . . . .	23
12.2	Computational Complexity Analysis . . . . .	23
<b>13</b>	<b>Implementation Guidelines</b>	<b>24</b>
13.1	Best Practices . . . . .	24
13.2	Security Considerations . . . . .	24
<b>14</b>	<b>Conclusion</b>	<b>24</b>
14.1	Summary of Contributions . . . . .	24
14.2	Implementation Impact . . . . .	25
14.3	Future Evolution . . . . .	25
<b>15</b>	<b>Canonical JSON Example</b>	<b>27</b>

# 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:

### Policy Governance & Compliance Checks

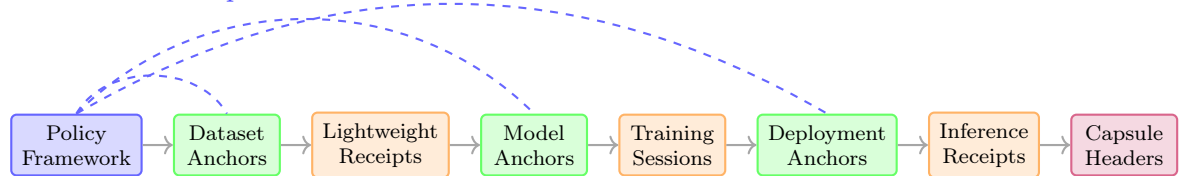


Figure 1: CIAF Data Structure Hierarchy

### 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.

```
1 @dataclass
2 class LCMPolicy:
3     """
4     Comprehensive CIAF LCM policy defining all cryptographic and
5     structural policies.
6     """
7     # Policy identification
8     policy_id: str = "ciaf_default_lcm_policy"
9
10    # Core cryptographic policy
11    hash_algorithm: str = "sha256"
12    canonicalization: str = "json_sorted_utf8"
13    domains: List[DomainType] = None
```

```

14 merkle: MerklePolicy = None
15 commitments: CommitmentType = CommitmentType.SALTED
16
17 # Schema versions for compatibility
18 anchor_schema_version: str = "1.0"
19 merkle_policy_version: str = "1.0"
20
21 # Protocol implementations (dependency injection)
22 rng: Optional[RNG] = None
23 anchor_deriver: Optional[AnchorDeriver] = None
24 anchor_store: Optional[AnchorStore] = None
25 signer: Optional[Signer] = None
26 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:

```

1 class DomainType(Enum):
2     """CIAF domain types for anchoring classification."""
3     DATASET = "CIAF|dataset"
4     DATASET_FAMILY = "CIAF|dataset|family"
5     DATASET_SPLIT = "CIAF|dataset|split"
6     MODEL = "CIAF|model"
7     TRAIN = "CIAF|train"
8     DEPLOYMENT = "CIAF|deployment"
9     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:

```

1 class CommitmentType(Enum):
2     """Commitment algorithms for privacy protection."""
3     SALTED = "salted" # SHA-256 with cryptographic salt
4     HMAC_SHA256 = "hmac_sha256" # HMAC-based commitment
5     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:

```

1 @dataclass
2 class MerklePolicy:
3     """Merkle tree construction policy."""
4     fanout: int = 2 # Binary tree structure

```



```

5 padding: str = "duplicate_last"      # Padding strategy for
  incomplete levels
6 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:

```

1 @dataclass
2 class LightweightReceipt:
3     """Minimal receipt stored during fast inference operations."""
4
5     # Core identification
6     receipt_id: str                # UUID v4 identifier
7     committed_at: str              # RFC 3339 timestamp with
8     Z                               # Request correlation ID
9     request_id: str
10
11     # Model and version tracking
12     model_anchor_ref: str           # Model anchor reference
13     model_version: str              # Specific model version
14
15     # Cryptographic commitments
16     input_hash: str                 # SHA-256 of input data
17     output_hash: str                # SHA-256 of output data
18     input_commitment: str           # Privacy-preserving
19     input commitment                 # Privacy-preserving
20     output_commitment: str          # Privacy-preserving
21     output commitment
22
23     # Optional encrypted data (for audit materialization)
24     raw_input: Optional[str] = None # Encrypted input data
25     raw_output: Optional[str] = None # Encrypted output data
26
27     # Operational metadata
28     priority: str = "normal"        # Processing priority
29     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:

$$\text{Traditional Receipt Size} = \text{Input Data} + \text{Output Data} + \text{Metadata} \quad (1)$$

$$\approx 50\text{KB to } 5\text{MB per operation} \quad (2)$$

(3)

$$\text{Lightweight Receipt Size} = \text{Hashes} + \text{Commitments} + \text{References} \quad (4)$$

$$\approx 500 \text{ bytes to } 1\text{KB per operation} \quad (5)$$

(6)

$$\text{Storage Reduction Ratio} = \frac{\text{Traditional Size}}{\text{Lightweight Size}} \quad (7)$$

$$\approx 100 : 1 \text{ to } 5000 : 1 \quad (8)$$

## 3.2 Enhanced Receipt Schema

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

```

1 class BaseReceipt(BaseModel):
2     """Base receipt with comprehensive validation."""
3
4     # Core identification with validation
5     receipt_id: str = Field(default_factory=lambda:
6 str(uuid.uuid4()))
7     evidence_strength: EvidenceStrength = EvidenceStrength.REAL
8     committed_at: str = Field(default_factory=lambda:
9 datetime.now(timezone.utc).isoformat().replace('+00:00', 'Z'))
10
11     @field_validator('receipt_id')
12     @classmethod
13     def validate_receipt_id(cls, v):
14         """Validate UUID format for receipt ID."""
15         try:
16             uuid.UUID(v)
17             return v
18         except ValueError:
19             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:

```

1 class EvidenceStrength(str, Enum):
2     """Evidence reliability classification."""
3     REAL = "real" # Production data from actual operations
4     SIMULATED = "simulated" # Realistic simulated data for testing
5     FALLBACK = "fallback" # Fallback data when primary collection
6     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:

```

1 @dataclass
2 class LCMDatasetAnchor:
3     """Comprehensive dataset anchor with cryptographic binding."""
4
5     # Core identification
6     anchor_id: str # Derived anchor
7     dataset_id: str # Original dataset
8     dataset_hash: str # SHA-256 of dataset
9     content
10
11     # Metadata and provenance
12     metadata: DatasetMetadata # Structured metadata
13     source_info: Dict[str, Any] # Data source information
14
15     # Cryptographic verification
16     merkle_root: str # Merkle root for batch
17     signature: str # Digital signature for
18     authenticity
19
20     # Policy compliance
21     policy: LCMPolicy # Governing policy
22     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:

```

1 @dataclass
2 class DatasetMetadata:
3     """Comprehensive dataset metadata for governance."""
4
5     # Basic properties
6     shape: Tuple[int, int] # Dataset dimensions
7     (rows, columns)
8     dtypes: Dict[str, str] # Column data types
9     size_bytes: int # Storage size in bytes
10
11     # Data quality metrics
12     null_counts: Dict[str, int] # Null value counts per
13     column
14     unique_counts: Dict[str, int] # Unique value counts per
15     column

```

```

13
14     # Provenance tracking
15     source_hash: str                    # Hash of source data
16     processing_steps: List[str]         # Applied transformations
17     validation_results: Dict[str, Any]  # Data validation outcomes
18
19     # Compliance metadata
20     privacy_classification: str         # Data privacy level
21     retention_policy: str               # Data retention
22     requirements
23     access_controls: List[str]          # Access control
24     specifications

```

Listing 9: Dataset Metadata Structure

## 4.2 Model Anchor Framework

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

```

1  @dataclass
2  class LCMMModelAnchor:
3      """Comprehensive model anchor with state binding."""
4
5      # Core identification
6      anchor_id: str                    # Derived anchor
7      identifier
8      model_id: str                    # Model identifier
9      model_hash: str                  # Hash of model parameters
10
11     # Model configuration
12     architecture: Dict[str, Any]      # Model architecture
13     specification
14     hyperparameters: Dict[str, Any]   # Training hyperparameters
15     framework_info: Dict[str, str]    # ML framework versions
16
17     # Training provenance
18     dataset_anchor_ref: str           # Reference to training
19     dataset anchor
20     training_session_ref: str         # Reference to training
21     session
22
23     # Performance metrics
24     validation_metrics: Dict[str, float] # Model validation results
25     benchmark_results: Dict[str, Any]   # Benchmark performance
26     data
27
28     # Cryptographic verification
29     merkle_root: str                  # Merkle root for
30     verification
31     signature: str                    # Digital signature
32     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
7     deployment_id: str # Deployment instance
8     identifier
9
10    # Model binding
11    model_anchor_ref: str # Reference to deployed
12    model
13    predeployment_anchor_ref: str # Reference to
14    pre-deployment validation
15
16    # Deployment configuration
17    environment_config: Dict[str, Any] # Deployment environment
18    settings
19    scaling_config: Dict[str, Any] # Auto-scaling
20    configuration
21    monitoring_config: Dict[str, Any] # Monitoring and alerting
22    setup
23
24    # Operational parameters
25    performance_targets: Dict[str, float] # SLA and performance
26    targets
27    resource_limits: Dict[str, Any] # Resource allocation
28    limits
29
30    # Security configuration
31    access_controls: List[str] # Access control policies
32    encryption_config: Dict[str, str] # Encryption configuration
33
34    # Verification and compliance
35    deployment_timestamp: str # Deployment timestamp
36    merkle_root: str # Verification Merkle root
37    signature: str # Deployment signature
38    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
4      def __init__(
5          self,
6          receipt_id: str,                # Unique receipt
7          model_anchor_ref: str,          # Model anchor reference
8          deployment_anchor_ref: str,     # Deployment anchor
9          request_id: str,                # Request correlation ID
10         query: str,                     # Input query/prompt
11         ai_output: str,                 # AI model output
12         input_commitment: LCMInferenceCommitment, # Input
13         output_commitment: LCMInferenceCommitment, # Output
14         explanation_digests: List[str] = None, # Explanation
15         prev_connections_digest: str = None,    # Connections
16         policy: LCMPolicy = None,              # Governing policy
17     ):
18         # Initialize core properties
19         self.receipt_id = receipt_id
20         self.model_anchor_ref = model_anchor_ref
21         self.deployment_anchor_ref = deployment_anchor_ref
22         self.request_id = request_id
23         self.query = query
24         self.ai_output = ai_output
25         self.input_commitment = input_commitment
26         self.output_commitment = output_commitment
27         self.explanation_digests = explanation_digests or []
28         self.prev_connections_digest = prev_connections_digest
29         self.policy = policy or get_default_policy()
30         self.timestamp =
31         datetime.now(timezone.utc).isoformat().replace('+00:00', 'Z')
32
33         # Computed cryptographic properties
34         self.receipt_digest = self._compute_receipt_digest()
35         self.connections_digest = self._compute_connections_digest()
36         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:

```

1  @dataclass
2  class LCMInferenceCommitment:
3      """Privacy-preserving commitment for inference data."""
4
5      commitment_type: CommitmentType # Commitment algorithm
6      commitment_value: str            # Computed commitment
7      value

```

```

7     metadata: Dict[str, Any] = None           # Additional commitment
      metadata
8
9     def __post_init__(self):
10         """Initialize metadata if not provided."""
11         if self.metadata is None:
12             self.metadata = {
13                 "created_at": datetime.now(timezone.utc).isoformat(),
14                 "algorithm_params": {},
15                 "verification_hints": {}
16             }

```

Listing 13: Inference Commitment Structure

## 5.2 Training Session Structure

Training sessions capture comprehensive information about model training processes:

```

1 @dataclass
2 class LCMTrainingSession:
3     """Comprehensive training session with audit trail."""
4
5     # Core identification
6     session_id: str           # Training session
      identifier
7     anchor_id: str           # Derived anchor
      identifier
8
9     # Training configuration
10    model_anchor_ref: str      # Reference to model
      anchor
11    dataset_anchor_ref: str    # Reference to training
      dataset
12    hyperparameters: Dict[str, Any] # Training hyperparameters
13
14    # Training progress tracking
15    epochs_completed: int      # Number of completed
      epochs
16    training_metrics: Dict[str, List[float]] # Training metrics by
      epoch
17    validation_metrics: Dict[str, List[float]] # Validation metrics
      by epoch
18
19    # Checkpoint management
20    checkpoint_hashes: List[str] # Hashes of model
      checkpoints
21    best_checkpoint_hash: str    # Hash of best performing
      checkpoint
22
23    # Environment and reproducibility
24    random_seeds: Dict[str, int] # Random seeds for
      reproducibility
25    environment_info: Dict[str, str] # Environment
      configuration
26    framework_versions: Dict[str, str] # ML framework versions

```

```

27
28     # Audit and verification
29     training_start: str                # Training start timestamp
30     training_end: str                 # Training completion
    timestamp
31     merkle_root: str                  # Training verification
    root
32     signature: str                    # Digital signature
33     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:

```

1  @dataclass
2  class CapsuleHeader:
3      """
4      Comprehensive CIAF LCM state capsule for complete audit trails.
5      """
6
7      # Core metadata
8      capsule_version: str                # Capsule format version
9      generated_at: str                   # Generation timestamp
10     policy: Dict[str, Any]               # Serialized policy
    configuration
11
12     # Canonical stage anchors (A-H stages)
13     stage_a_dataset: Optional[Dict[str, Any]] = None        # Dataset
    anchor
14     stage_b_model: Optional[Dict[str, Any]] = None          # Model
    anchor
15     stage_c_training: Optional[Dict[str, Any]] = None       #
    Training session
16     stage_d_predeployment: Optional[Dict[str, Any]] = None  #
    Pre-deployment
17     stage_e_deployment: Optional[Dict[str, Any]] = None     #
    Deployment
18     stage_f_test_evaluation: Optional[Dict[str, Any]] = None # Test
    evaluation
19     stage_g_inference: Optional[Dict[str, Any]] = None      #
    Inference receipt
20     stage_h_roots: Optional[Dict[str, Any]] = None           # Merkle
    roots
21
22     def to_json(self, indent: int = 2) -> str:
23         """Convert to pretty-printed JSON for human readability."""
24         return json.dumps(asdict(self), indent=indent,
25                             sort_keys=True)
26

```



```

26     def to_compact_json(self) -> str:
27         """Convert to compact JSON for storage efficiency."""
28         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:

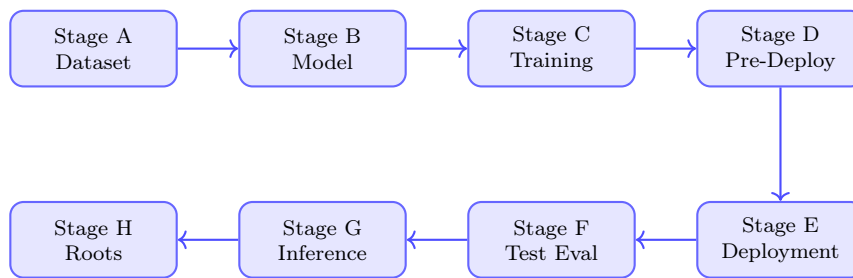


Figure 2: Eight-Stage CIAF Lifecycle Architecture

## 7 Protocol Interface Specifications

### 7.1 Cryptographic Protocol Interfaces

The framework defines protocol interfaces for swappable cryptographic implementations:

```

1 @runtime_checkable
2 class Signer(Protocol):
3     """Protocol for digital signature implementations."""
4
5     key_id: str # Signer key identifier
6
7     def sign(self, data: bytes) -> str:
8         """Sign data and return signature string."""
9         ...
10
11     def verify(self, data: bytes, signature: str) -> bool:
12         """Verify signature against data."""
13         ...

```

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

```

1 @runtime_checkable
2 class Merkle(Protocol):
3     """Protocol for Merkle tree implementations."""
4
5     def add_leaf(self, leaf_hash: str) -> str:
6         """Add leaf and return new root hash."""
7         ...
8
9     def get_root(self) -> str:
10        """Get the current Merkle root hash."""
11        ...
12
13    def get_proof(self, leaf_hash: str) -> List[Tuple[str, str]]:
14        """Get inclusion proof as (hash, position) tuples."""
15        ...
16
17    def verify_proof(self, leaf_hash: str, proof: List[Tuple[str,
18    str]], root: str) -> bool:
19        """Verify Merkle inclusion proof."""
20        ...

```

Listing 17: Merkle Protocol Interface

## 7.2 Storage Protocol Interfaces

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

```

1 @runtime_checkable
2 class AnchorStore(Protocol):
3     """Protocol for anchor storage implementations."""
4
5     def append_anchor(self, anchor: Dict[str, Any]) -> None:
6         """Append new anchor with WORM semantics."""
7         ...
8
9     def get_latest_anchor(self) -> Optional[Dict[str, Any]]:
10        """Get the most recent anchor from store."""
11        ...

```

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:

```

1 {
2   "$id": "https://cognitiveinsight.ai/schemas/capsule.schema.json",
3   "$schema": "https://json-schema.org/draft/2020-12/schema",
4   "title": "CIAF Audit Capsule",
5   "type": "object",
6   "required": [
7     "capsule_version",
8     "capsule_type",
9     "timestamp",
10    "record",
11    "proofs",
12    "anchor",
13    "verification"
14  ],
15  "properties": {
16    "capsule_version": { "type": "string", "const": "1.0" },
17    "capsule_type": { "type": "string", "const": "audit_proof" },
18    "timestamp": { "type": "string", "format": "date-time" },
19    "record": {
20      "type": "object",
21      "required": ["type", "metadata", "leaf_hash"],
22      "properties": {
23        "type": {
24          "enum": [
25            "dataset", "model", "inference",
26            "anchor", "monitoring", "compliance"
27          ]
28        },
29        "metadata": { "type": "object" },
30        "leaf_hash": { "type": "string" }
31      }
32    },
33    "proofs": {
34      "type": "object",
35      "required": ["merkle_path", "merkle_root",
36        "inclusion_proof_valid"],
37      "properties": {
38        "merkle_path": {
39          "type": "array",
40          "items": {
41            "type": "array",
42            "prefixItems": [
43              { "type": "string" },
44              { "enum": ["left", "right"] }
45            ],
46            "minItems": 2,
47            "maxItems": 2
48          }
49        },
50        "merkle_root": { "type": "string" },
51        "inclusion_proof_valid": { "type": "boolean" }
52      }
53    }
54  }

```

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. The diagram below shows three distinct phases of the CIAF lifecycle:

- **Training Phase:** Dataset anchors define model anchors, which are used in training sessions
- **Deployment Phase:** Training sessions create deployment anchors that generate inference receipts
- **Storage & Access Phase:** Inference receipts produce lightweight receipts that are stored in capsule headers, with materialization and integration processes providing data access

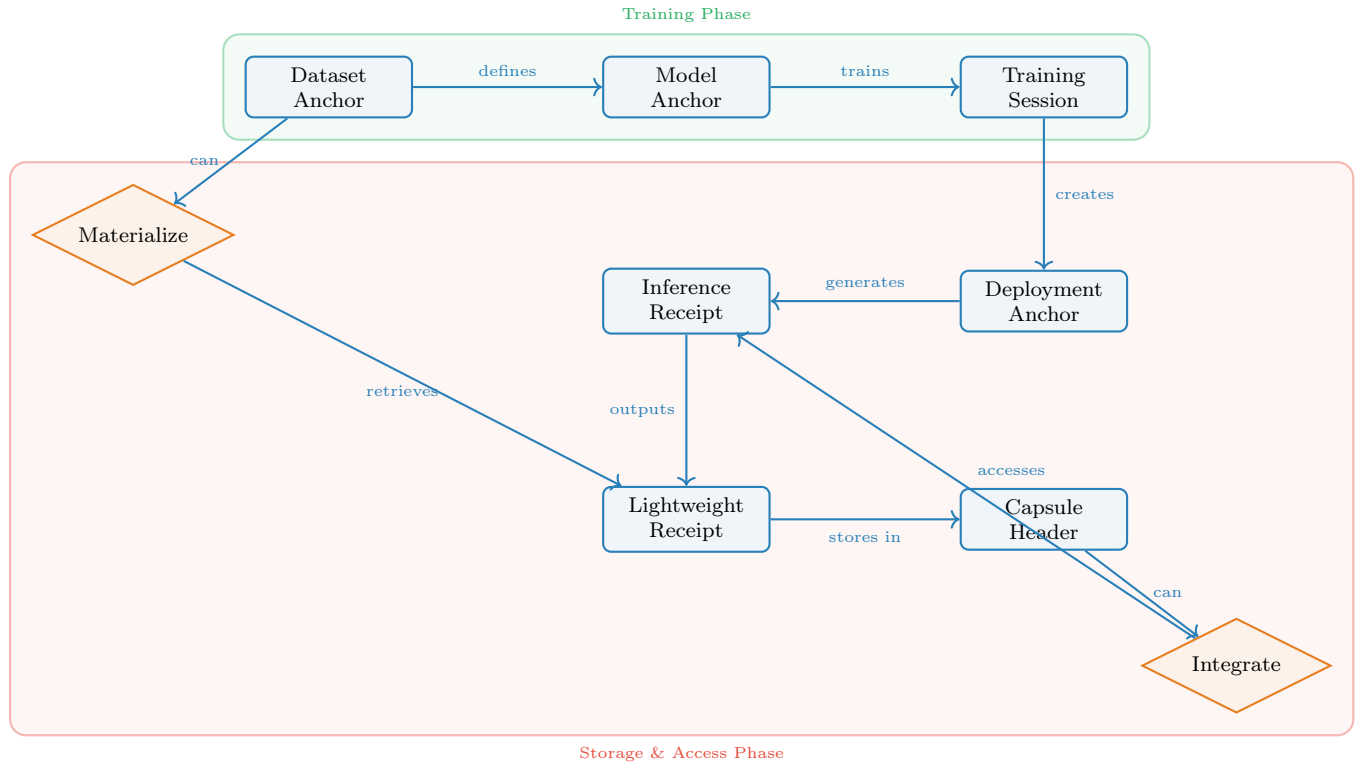


Figure 3: CIAF Data Structure Flow and Relationships

## 9.2 Cryptographic Verification Chain

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

$$\text{Dataset Hash} \rightarrow \text{Model Hash} \rightarrow \text{Training Hash} \quad (9)$$

$$\rightarrow \text{Deployment Hash} \rightarrow \text{Inference Hash} \quad (10)$$

$$\rightarrow \text{Capsule Hash} \rightarrow \text{Digital Signature} \quad (11)$$

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:

```
1 def canonical_json(obj: Any) -> str:
2     """
3     Create canonical JSON representation for cryptographic hashing.
4
5     Args:
6         obj: Object to serialize
7
8     Returns:
9         Canonical JSON string with sorted keys and no whitespace
10    """
11    return json.dumps(
12        obj,
13        sort_keys=True,                # Ensure deterministic key ordering
14        separators=(',', ':'),        # Remove whitespace
15        ensure_ascii=True,            # Use ASCII encoding
16        default=str,                  # Convert non-serializable types
17        to_strings
18    )
19
20 def canonical_hash(obj: Any) -> str:
21     """
22     Compute SHA-256 hash of canonical JSON representation.
23
24     Args:
25         obj: Object to hash
26
27     Returns:
28         Hexadecimal SHA-256 hash string
29    """
30    canonical_str = canonical_json(obj)
31    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:

```

1  class ReceiptValidator:
2      """Comprehensive validation for receipt data structures."""
3
4      @staticmethod
5      def validate_training_receipt(receipt: TrainingReceipt) ->
6      List[str]:
7          """
8              Validate training receipt structure and constraints.
9
10             Returns:
11                 List of validation error messages (empty if valid)
12             """
13             errors = []
14
15             # Validate UUID format
16             try:
17                 uuid.UUID(receipt.receipt_id)
18             except ValueError:
19                 errors.append("Invalid receipt_id format")
20
21             # Validate anchor formats (64-character hex strings)
22             for anchor_name, anchor_value in [
23                 ("model_anchor_ref", receipt.model_anchor_ref)
24             ]:
25                 if not re.match(r'^[a-f0-9]{64}$', anchor_value.lower()):
26                     errors.append(f"Invalid {anchor_name} format")
27
28             # Validate Merkle path
29             for i, path_element in enumerate(receipt.merkle_path):
30                 if not isinstance(path_element, list) or
31                     len(path_element) != 2:
32                     errors.append(f"Invalid Merkle path element at index
33                     {i}: must be [hash, position]")
34                 else:
35                     hash_val, position = path_element
36                     if not re.match(r'^[a-f0-9]{64}$', hash_val.lower()):
37                         errors.append(f"Invalid Merkle path hash at
38                         index {i}")
39                     if position not in ["left", "right"]:
40                         errors.append(f"Invalid Merkle path position at
41                         index {i}: must be 'left' or 'right'")
42
43             # Validate timestamp format
44             try:
45                 datetime.fromisoformat(receipt.committed_at.replace('Z',
46                     '+00:00'))
47             except ValueError:
48                 errors.append("Invalid timestamp format")

```

```
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:



Hash Computation :  $O(n)$  where  $n$  = serialized size (12)

Signature Generation :  $O(1)$  for Ed25519 (13)

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

Schema Validation :  $O(k)$  where  $k$  = field count (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):

```
1 {
2   "capsule_version": "1.0",
3   "capsule_type": "audit_proof",
4   "timestamp": "2025-10-22T00:00:00Z",
5   "record": {
6     "type": "inference",
7     "metadata": {
8       "request_id": "req_12345",
9       "model_anchor_ref": "mod_abcd1234",
10      "deployment_anchor_ref": "dep_efgh5678",
11      "committed_at": "2025-10-22T00:00:00Z",
12      "evidence_strength": "simulated"
13    },
14    "leaf_hash": "e3b0c44298fc1c149afbf4c8996fb924..."
15  },
16  "proofs": {
17    "merkle_path": [
18      ["9f86d081884c7d659a2feaa0c55ad015...", "right"],
19      ["b6d81b360a5672d80c27430f39153e2c...", "left"]
20    ],
21    "merkle_root": "3a7bd3e2360a3d...",
22    "inclusion_proof_valid": true
23  },
24  "anchor": { "capsule_id": "cap_001" },
25  "verification": {
26    "signature": "ed25519:abcd...xyz",
27    "timestamp": "2025-10-22T00:00:00Z",
28    "signer_id": "ciaf_production_key_001"
29  }
30 }
```

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

---

### Legal Notice

© 2025 Denzil James Greenwood

This technical specification, “*CIAF Data Structures: Technical Specification for Lazy Capsule Materialization*,”

is licensed under the [Creative Commons Attribution 4.0 International License \(CC BY 4.0\)](#).

All accompanying source code is released under the [Apache License 2.0](#).

Cognitive Insight™ and Lazy Capsule Materialization (LCM)™ are trademarks of Denzil James Greenwood.