CIAF Variables, Interfaces & Functions Reference

Comprehensive Naming Convention Standards

Authoritative Source of Truth
This document provides comprehensive reference for variable naming conventions, interfaces, and functions used throughout the Cognitive Insight AI Framework (CIAF) codebase. The goal is to ensure consistent naming patterns and provide clarity on variable relationships and cryptographic hierarchies.

Author: Denzil James Greenwood
Version: 1.0.0
Date: October 23, 2025
Document Type: Technical Reference

Usage Guidelines

- This document serves as the single source of truth for CIAF naming conventions
- All code implementations must conform to patterns specified herein
- Consult this reference when adding new variables or refactoring existing code
- Maintain cryptographic hierarchy relationships as documented

Classification: Technical Documentation
Distribution: Internal Development Team
Last Updated: October 24, 2025

Abstract

The Cognitive Insight AI Framework (CIAF) Variables Reference provides comprehensive documentation for naming conventions, interface patterns, and function signatures used throughout the CIAF codebase. This document establishes consistent naming standards for cryptographic variables, anchor hierarchies, receipt management, and API interfaces. It serves as the authoritative source for developers implementing CIAF components, ensuring code clarity, maintainability, and adherence to cryptographic security principles. The reference covers core naming patterns, suffix conventions, variable relationships, and best practices for maintaining consistency across the framework's distributed architecture.

Contents

1 Document Purpose and Scope

This document provides a comprehensive reference for variable naming conventions, interfaces, and functions used throughout the Cognitive Insight AI Framework (CIAF) codebase. The primary objectives are:

- **Consistency Enforcement:** Establish uniform naming patterns across all CIAF components
- **Clarity Provision:** Distinguish between related but distinct variable types (e.g., `modelAnchor` vs `modelAnchorRef`)
- **Hierarchy Documentation:** Define cryptographic relationships and derivation chains
- **Interface Standardization:** Specify function signatures and protocol implementations

1.1 Document Authority

This document serves as the **single source of truth** for CIAF naming conventions. All code implementations, documentation, and technical specifications must conform to the patterns and standards defined herein.

1.2 Version Control

Version	Date	Changes
1.0.0	October 23, 2025	Initial comprehensive reference

2 Naming Convention Principles

2.1 Core Naming Patterns

The CIAF framework employs a hierarchical naming system based on the following conventions:

snake_case Primary convention for variables, functions, and module names

PascalCase Used for classes, enums, and dataclasses

UPPER_CASE Used for constants and enum values

Descriptive suffixes Added to indicate variable purpose or type

2.2 Suffix Conventions

The framework uses standardized suffixes to indicate variable purpose and cryptographic properties:

Table 1: Standard Suffix Conventions

Suffix	Purpose
`_anchor`	Cryptographic anchor objects/bytes (internal form)
`_anchor_hex`	Hex-encoded anchor strings (external/AAD form)
`_anchor_ref`	Opaque reference/ID strings pointing to anchors
`_ref`	References or IDs pointing to anchors/objects
`_hash`	SHA-256 or other cryptographic hash values
`_digest`	Content-derived cryptographic digests
`_root`	Merkle tree root hashes
`_id`	Unique identifiers (usually strings)
`_hex`	Hex-encoded byte values
`_bytes`	Raw byte data
`_pem`	PEM-encoded cryptographic keys
`_metadata`	Structured metadata objects

3 Core Cryptographic Variables

3.1 Anchor-Related Variables

The CIAF framework implements a hierarchical anchor system for cryptographic integrity. The following variables represent different aspects of this system:

Listing 1: Master Anchors

```
# Master anchors (top-level derivation)
master_anchor: bytes                    # Root anchor derived from
    password + salt
master_password: str                    # Password for master
    anchor derivation

# Hierarchical anchors
dataset_anchor: bytes                   # Derived from
    master_anchor + dataset_hash
model_anchor: bytes                     # Derived from
    master_anchor + model_hash
capsule_anchor: bytes                   # Derived from
    dataset_anchor + capsule_id
```

Listing 2: Anchor Hex Representations

```
# Anchor hex representations
dataset_anchor_hex: str                 # Hex-encoded dataset anchor
model_anchor_hex: str                   # Hex-encoded model anchor
capsule_anchor_hex: str                 # Hex-encoded capsule anchor
```

Listing 3: Anchor References and IDs

```
# Anchor references/IDs
anchor_id: str                          # Unique identifier for
    anchor
anchor_ref: str                         # Reference to existing
    anchor
```

3.2 Hash Variables

Cryptographic hash variables follow standardized naming patterns to indicate their purpose and derivation:

Listing 4: Content Hashes

```
# Content hashes
dataset_hash: str                       # SHA-256 hash of dataset
    content
model_hash: str                         # SHA-256 hash of model
    parameters
content_hash: str                       # Generic content hash
schema_digest: str                      # Hash of data schema
params_root: str                        # Merkle root of model
    parameters
arch_root: str                          # Merkle root of model
    architecture
```

Listing 5: Cryptographic Digests

```
# Cryptographic digests
leaf_hash: str                          # Individual Merkle tree
    leaf
merkle_root: str                        # Merkle tree root hash
root_hash: str                          # Generic root hash
split_assignment_digest: str            # Hash of train/val/test
    split assignments
hp_digest: str                          # Hyperparameter
    configuration digest
env_digest: str                         # Training environment
    digest
```

3.3 Signature Variables

Digital signature variables maintain clear distinctions between different representations and key types:

Listing 6: Digital Signatures

```
# Digital signatures
signature: str                          # Base64-encoded Ed25519
    signature
signature_bytes: bytes                  # Raw signature bytes
merkle_signature: str                   # Signature over Merkle root
```

Listing 7: Keys and Key Management

```
# Keys and key management
```

```
private_key: Ed25519PrivateKey          # Ed25519 private key object
public_key: Ed25519PublicKey            # Ed25519 public key object
key_id: str                             # Unique key identifier
public_key_pem: str                     # PEM-encoded public key
private_key_pem: str                    # PEM-encoded private key
key_fingerprint: str                    # SHA-256 fingerprint of
   public key
```

4 LCM (Lazy Capsule Materialization) Variables

4.1 Core LCM Objects

The Lazy Capsule Materialization system uses specialized dataclasses and managers for different aspects of the AI lifecycle:

Listing 8: Anchor Objects (Dataclasses)

```
# Anchor objects (dataclasses)
dataset_anchor: LCMDatasetAnchor        # Dataset anchor with
   metadata
model_anchor: LCMModelAnchor            # Model anchor with metadata
training_anchor: LCMTrainingAnchor      # Training session anchor
deployment_anchor: LCMDeploymentAnchor # Deployment anchor
```

Listing 9: Manager Objects

```
# Manager objects
dataset_manager: LCMDatasetManager      # Dataset management
model_manager: LCMModelManager          # Model management
training_manager: LCMTrainingManager    # Training session management
deployment_manager: LCMDeploymentManager # Deployment management
```

4.2 Receipt Variables

Receipt variables represent different levels of audit detail and materialization:

Listing 10: Receipt Objects

```
# Receipt objects
lightweight_receipt: LightweightReceipt    # Minimal audit receipt
inference_receipt: InferenceReceipt        # Full inference audit
   record
training_receipt: TrainingReceipt          # Training session
   receipt
```

Listing 11: Receipt Fields

```
# Receipt fields
receipt_id: str                            # Unique receipt
   identifier
inference_id: str                          # Unique inference
   identifier
capsule_id: str                            # Capsule identifier
session_id: str                            # Training/inference
   session ID
```

```
committed_at: str                          # Receipt creation
   timestamp (RFC 3339 Z)
```

4.3 Split and Dataset Variables

Dataset management variables handle data organization and metadata:

Listing 12: Dataset Splits

```
# Dataset splits
train_split: DatasetSplit              # Training data split
val_split: DatasetSplit                # Validation data split
test_split: DatasetSplit               # Test data split
split_assignment: Dict[str, str]       # Record ID to split mapping
```

Listing 13: Dataset Metadata

```
# Dataset metadata
dataset_metadata: DatasetMetadata      # Comprehensive dataset info
split_metadata: SplitMetadata          # Split-specific metadata
schema_metadata: Dict[str, Any]        # Data schema information
```

5 API and Interface Variables

5.1 Protocol Interface Variables

The CIAF framework uses protocol-based interfaces for core cryptographic and storage operations:

Listing 14: Core Protocol Implementations

```
# Core protocol implementations
signer: Signer                         # Digital signature protocol
rng: RNG                              # Random number generator
   protocol
merkle: Merkle                        # Merkle tree protocol
anchor_deriver: AnchorDeriver         # Anchor derivation protocol
anchor_store: AnchorStore             # Anchor storage protocol
```

Listing 15: API Handler Protocols

```
# API handler protocols
dataset_api_handler: DatasetAPIHandler # Dataset API operations
model_api_handler: ModelAPIHandler     # Model API operations
audit_api_handler: AuditAPIHandler     # Audit API operations
```

5.2 Framework Objects

Governance framework variables provide domain-specific compliance implementations:

Listing 16: Governance Frameworks

```
# Governance frameworks
governance_framework: AIGovernanceFramework    # Base governance
banking_framework: BankingAIGovernanceFramework # Banking-specific
```

```
healthcare_framework: HealthcareAIGovernanceFramework #
    Healthcare-specific
government_framework: GovernmentAIGovernanceFramework #
    Government-specific
```

Listing 17: Organization and Configuration

```
# Organization and config
organization_id: str                    # Unique organization
    identifier
framework_version: str                  # Framework version string
governance_config: Dict[str, Any]       # Configuration parameters
compliance_history: List[Dict]          # Historical compliance
    events
```

6 Enum and Constant Variables

6.1 Core Enums

Enumeration variables provide type safety and standardized values:

Listing 18: Core Enums

```
# Record types
record_type: RecordType                # DATASET, MODEL, INFERENCE,
    ANCHOR, etc.

# Algorithms
hash_algorithm: HashAlgorithm          # SHA256, SHA3_256, BLAKE3
signature_algorithm: SignatureAlgorithm # ED25519, MOCK

# Consent management
consent_status: ConsentStatus          # GRANTED, DENIED, EXPIRED,
    etc.
consent_type: ConsentType              # EXPLICIT, IMPLIED,
    PARENTAL, etc.
consent_scope: ConsentScope            # DATA_PROCESSING, SHARING,
    etc.
```

6.2 Constant Variables

Framework constants ensure consistency across implementations:

Listing 19: Cryptographic Constants

```
# Cryptographic constants
SALT_LENGTH: int = 16                   # Salt length in bytes
PBKDF2_ITERATIONS: int = 100_000       # PBKDF2 iteration count
KDF_DKLEN: int = 32                    # Key derivation output length
HASH_OUTPUT_LENGTH: int = 64           # SHA-256 hex output length

# Default algorithms
DEFAULT_HASH_FUNCTION: str = "sha256"
DEFAULT_SIGNATURE_ALGORITHM: str = "ed25519"
DEFAULT_PUBKEY_ID: str = "ciaf_production_key_001"
```

```
# Schema versions
ANCHOR_SCHEMA_VERSION: str = "1.0"
MERKLE_POLICY_VERSION: str = "1.0"

# Prefixes
EVENT_ID_PREFIX: str = "evt"
```

7 Function Naming Patterns

7.1 Cryptographic Functions

Function naming follows consistent patterns that reflect their cryptographic purpose:

Listing 20: Hash Functions

```
# Hash functions
def sha256_hash(data: bytes) -> str
def blake3_hash(data: bytes) -> str
def sha3_256_hash(data: bytes) -> str
def compute_hash(data: bytes, algorithm: str) -> str
def hmac_sha256(key: bytes, data: bytes) -> str
```

Listing 21: Anchor Derivation Functions

```
# Anchor derivation functions
def derive_master_anchor(password: str, salt: bytes) -> bytes
def derive_dataset_anchor(master_anchor: bytes, dataset_hash: str)
    -> bytes
def derive_model_anchor(master_anchor: bytes, model_hash: str) ->
    bytes
def derive_capsule_anchor(dataset_anchor: bytes, capsule_id: str)
    -> bytes
```

Listing 22: Encryption Functions

```
# Encryption functions
def encrypt_aes_gcm(key: bytes, plaintext: bytes, aad: bytes) ->
    tuple
def decrypt_aes_gcm(key: bytes, ciphertext: bytes, nonce: bytes,
                    tag: bytes, aad: bytes) -> bytes
def make_aad(dataset_anchor_hex: str, capsule_id: str,
             policy_id: str) -> bytes
```

7.2 LCM Management Functions

LCM management functions follow consistent creation and manipulation patterns:

Listing 23: Manager Creation Functions

```
# Manager creation functions
def create_dataset_manager(policy: LCMPolicy, rng: RNG) ->
    LCMDatasetManager
def create_model_manager(policy: LCMPolicy, rng: RNG) ->
    LCMModelManager
def create_training_manager(model_anchor: LCMModelAnchor,
                            datasets: List) -> LCMTrainingManager
```

Listing 24: Receipt Generation Functions

```
# Receipt generation functions
def generate_lightweight_receipt(inference_data: Dict) ->
    LightweightReceipt
def materialize_full_receipt(lightweight_receipt:
    LightweightReceipt) -> InferenceReceipt
```

Listing 25: Validation Functions

```
# Validation functions
def validate_anchor_chain(anchor_chain: List[str]) -> bool
def verify_merkle_proof(leaf_hash: str, proof: List, root: str) ->
    bool
def validate_governance_requirements(system_id: str,
                                     requirements: Dict) -> Dict
```

7.3 API Functions

API functions implement standard CRUD patterns with consistent naming:

Listing 26: CRUD Operations

```
# CRUD operations
def create_dataset(dataset_id: str, metadata: Dict) -> Dict
def get_dataset(dataset_id: str) -> Optional[Dict]
def update_dataset(dataset_id: str, updates: Dict) -> Dict
def delete_dataset(dataset_id: str) -> bool
```

Listing 27: Assessment Functions

```
# Assessment functions
def assess_compliance(system_id: str, assessment_type: str) -> Dict
def generate_audit_report(system_id: str, report_type: str) -> Dict
def record_governance_event(event_type: str, event_data: Dict) ->
    str
```

8 Variable Relationship Patterns

8.1 Anchor Relationships

The cryptographic anchor system maintains clear hierarchical relationships:

Listing 28: Anchor Hierarchy

```
# Base anchor and its references
model_anchor: LCMModelAnchor                # Full anchor object with
    metadata
model_anchor_ref: str                       # Reference ID to the anchor
model_anchor_hex: str                       # Hex representation for
    AAD/binding
model_hash: str                             # Content hash used to
    derive anchor

# Chain relationships
master_anchor: bytes                        # Root of derivation chain
```

```
dataset_anchor: bytes                # Derived from master_anchor
   + dataset_hash
capsule_anchor: bytes                # Derived from
   dataset_anchor + capsule_id
```

8.2 Receipt Relationships

Receipt variables represent different stages of audit trail materialization:

Listing 29: Receipt Progression

```
# Receipt progression
lightweight_receipt: LightweightReceipt    # Minimal storage during
   operation
inference_receipt: InferenceReceipt        # Materialized full
   audit record
receipt_id: str                            # Unique identifier
   linking them
receipt_ref: str                           # Reference used in
   other contexts
```

8.3 Merkle Tree Relationships

Merkle tree variables maintain cryptographic proof relationships:

Listing 30: Merkle Tree Structure

```
# Tree construction
leaf_hash: str                       # Individual operation hash
merkle_path: List[Tuple[str, str]]   # Proof path from leaf to
   root
merkle_root: str                     # Root hash of the tree
merkle_signature: str                # Digital signature over root
```

9 Best Practices

9.1 Consistent Suffixing

- Always use `_anchor` for anchor objects/bytes
- Use `_ref` or `_id` for string references to anchors
- Use `_hex` when converting bytes to hex strings
- Use `_hash` for content-derived cryptographic hashes

9.2 Type Clarity

- Include type hints for all function parameters and returns
- Use descriptive variable names that indicate their purpose
- Distinguish between raw bytes, hex strings, and object references

9.3 Hierarchical Naming

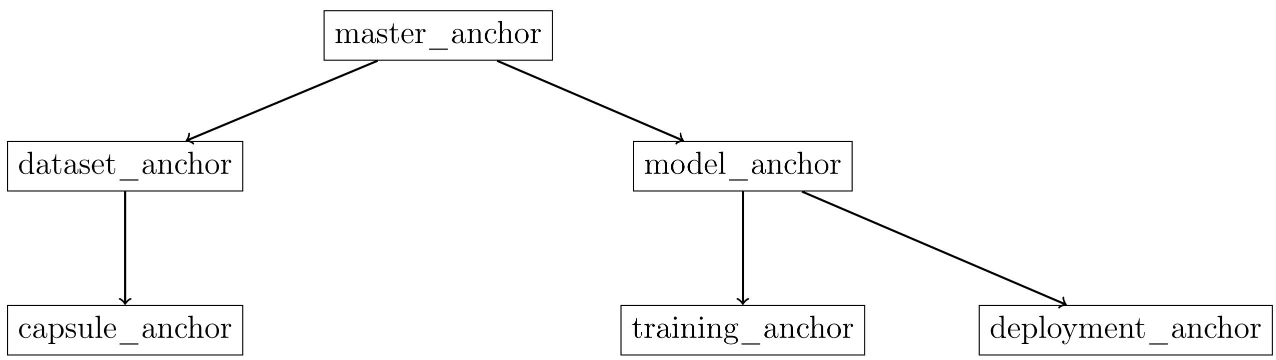
- Reflect the cryptographic hierarchy in variable names
- Use consistent prefixes for related operations (e.g., `derive_*_anchor`)
- Group related variables with common prefixes

9.4 Documentation

- Include docstrings explaining the relationship between variables
- Document the cryptographic properties of anchor variables
- Explain when to use different forms of the same conceptual data

10 Variable Cross-Reference Index

10.1 Primary Anchor Variables



10.2 Hash Chain Variables

- `dataset_hash` → used in `derive_dataset_anchor()`
- `model_hash` → used in `derive_model_anchor()`
- `params_root` → Merkle root of model parameters
- `arch_root` → Merkle root of model architecture

10.3 Receipt Chain Variables

- `lightweight_receipt` → minimal audit record
- `inference_receipt` → full materialized record
- `training_receipt` → training session record
- Connected by `receipt_id` and `inference_id`

10.4 API Object Variables

- `*_manager` objects handle lifecycle operations
- `*_api_handler` objects handle HTTP/API operations
- `governance_framework` objects handle compliance
- Connected through dependency injection patterns

11 Implementation Guidelines

11.1 New Variable Introduction

When introducing new variables to the CIAF codebase:

1. Consult this reference for appropriate naming patterns
2. Ensure suffix conventions align with variable purpose
3. Document cryptographic relationships in code comments
4. Update this reference document when new patterns are established

11.2 Code Review Checklist

During code reviews, verify:

- Variable names follow established suffix conventions
- Type hints are present and accurate
- Cryptographic hierarchy relationships are preserved
- Function naming patterns align with documented standards

11.3 Refactoring Guidelines

When refactoring existing code:

- Maintain backward compatibility where possible
- Update variable names to conform to current standards
- Preserve cryptographic security properties
- Update documentation to reflect changes

Conclusion

This Variables Reference serves as the comprehensive authority for CIAF naming conventions, ensuring consistency, clarity, and maintainability across the framework's distributed architecture. By adhering to these standards, developers can create code that is both cryptographically secure and easily understood by team members.

The hierarchical naming system reflects the underlying cryptographic relationships, making code review and security analysis more effective. Regular consultation of this reference during development and code review processes will maintain the high standards required for production AI governance systems.

Author Information

Denzil James Greenwood is the creator of the Cognitive Insight AI Framework and inventor of the Lazy Capsule Materialization process. This Variables Reference represents the canonical standards for CIAF development and serves as the authoritative guide for naming conventions across all framework components.

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

Copyright Notice