ADAC 1.0 — Archival Digital Asset Container Format Specification

Version: 1.0
Status: Stable

Date: 2026

Abstract

The **Archival Digital Asset Container (ADAC)** format defines a self-describing, ZIP-based archival package for the long-term preservation and interchange of digital assets. A single ADAC container encapsulates master files, access derivatives, structured metadata, region annotations, non-destructive edit pipelines, domain-specific profile extensions, provenance history, and cryptographic fixity data — producing a portable, verifiable artifact that requires no external database, catalog, or service to be fully understood.

An ADAC container is a **living archive** — it is designed to be opened, enriched, and updated over its lifetime. Metadata can be refined, regions annotated, derivatives regenerated, and provenance events appended. The only content considered immutable is the master file(s); SHA-256 checksums are recomputed on every save to reflect the current container state while guaranteeing that master artifacts remain unaltered.

Table of Contents

---

1. Introduction

1.1 Problem Statement

Digital preservation demands more than storing files. A scanned document, audio recording, or photograph becomes meaningless without its descriptive context, provenance history, structural relationships, and integrity guarantees. Existing approaches scatter this information across databases, file systems, and proprietary

catalogs — creating fragile dependencies that break when systems are migrated, decommissioned, or lost.

1.2 Solution

ADAC solves this by packaging everything about a digital asset into a single, self-describing, **living** archive:

- **The artifact itself** — stored at full fidelity, uncompressed, in archival-grade format. Master files are the only immutable content in the container.
- **Its meaning** — descriptive, technical, administrative, and preservation metadata following established archival standards.
- **Its structure** — region annotations defining bounded areas of interest, with a mechanism for domain-specific enrichment via linked entities.
- **Its history** — a provenance log recording every significant action from capture to present.
- **Its integrity** — SHA-256 checksums recomputed on every save, verifiable at any future point without the original source files. Master file checksums specifically guarantee that the original artifacts have not been altered.
- **Its derivatives** — access copies, previews, and thumbnails linked back to their master.
- **Its domain context** — pluggable profile extensions for genealogy, legal, medical, or any other domain, without modifying the core format.
- **Its evolution** — the container is designed to be opened, enriched, and re-saved throughout its lifetime. Metadata is refined, regions are annotated, derivatives are regenerated, and provenance events are appended — all without disturbing the original master artifacts.

1.3 Design Philosophy

Principle	Description
Self-contained	Every ADAC container is a complete, portable unit. No external database, service, or catalog is required to understand its contents.
Living archive	An ADAC container is not a write-once artifact. It is designed to be opened, enriched, and re-saved throughout its lifetime — metadata refined, regions annotated, derivatives regenerated, provenance events appended. Only the master file(s) are immutable.
Preservation-first	Master files are stored without compression and are treated as immutable once ingested. Non-destructive edit pipelines preserve the original artifact. The format favors longevity over compactness.
Media-agnostic	ADAC supports images, audio, video, 3D models, and mixed-media assets through pluggable coordinate systems and region types.
Extensible without modification	Domain-specific profiles extend the container through an additive mechanism — they never modify core structures. Linked entities on regions provide per-region extensibility.
Verifiable	Cryptographic fixity data is recomputed on every save. Master file checksums guarantee that original artifacts have not been altered; checksums for other files reflect the current container state.
Open	The physical format is standard ZIP (ISO/IEC 21320-1:2015). Any tool that reads ZIP files can extract content from an ADAC container. Metadata is stored as human-readable JSON.
Forward-compatible	All JSON structures tolerate unknown properties. Future versions can add fields without breaking existing readers.

1.4 Scope

This specification defines:

- The physical container format (ZIP archive structure)
- Required and optional directory layout and file conventions

- JSON schemas for the manifest, core metadata, region annotations, edit pipelines, provenance log, and checksum manifest
- The profile extension mechanism for domain-specific metadata
- The linked entities mechanism for per-region extensibility
- The encryption descriptor model
- Fixity computation and verification procedures
- Validation rules, error codes, and conformance levels
- Media types and file extensions

This specification does **not** define:

- Domain-specific profile schemas (see the ADAC-Genealogy and ADAC-Legal specifications)
- Application-level APIs or programming interfaces
- User interface requirements
- Network protocols for container exchange

1.5 Audience

This specification is intended for:

- Software developers implementing ADAC readers and writers
- Archivists and digital preservation professionals evaluating the format
- Domain experts designing profile extensions
- Standards bodies evaluating ADAC for adoption

---

2. Conformance

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "MAY", and "OPTIONAL" in this document are to be interpreted as described in RFC 2119.

2.1 Conformance Levels

An ADAC container conforms to this specification at one of two levels:

Level	Requirements
Minimal	The container is a valid ZIP archive containing a valid `manifest.json` with at least one master entry whose referenced file exists, and a valid `metadata/core.json`.
Archival	Minimal requirements plus: provenance log present, checksum manifest present with valid SHA-256 hashes for all files, and all referenced region/edit/XMP files present.

An ADAC reader MUST be able to open any Minimal-conformant container. An ADAC reader SHOULD handle missing optional content gracefully (e.g., absent provenance, absent regions) without error.

2.2 Forward Compatibility

An ADAC reader MUST tolerate unknown JSON properties in all metadata files. Unknown properties MUST be preserved during round-trip serialization (read → modify → write). An ADAC reader MUST NOT reject a container solely because it contains unrecognized profile files, linked entity keys, or metadata fields.

---

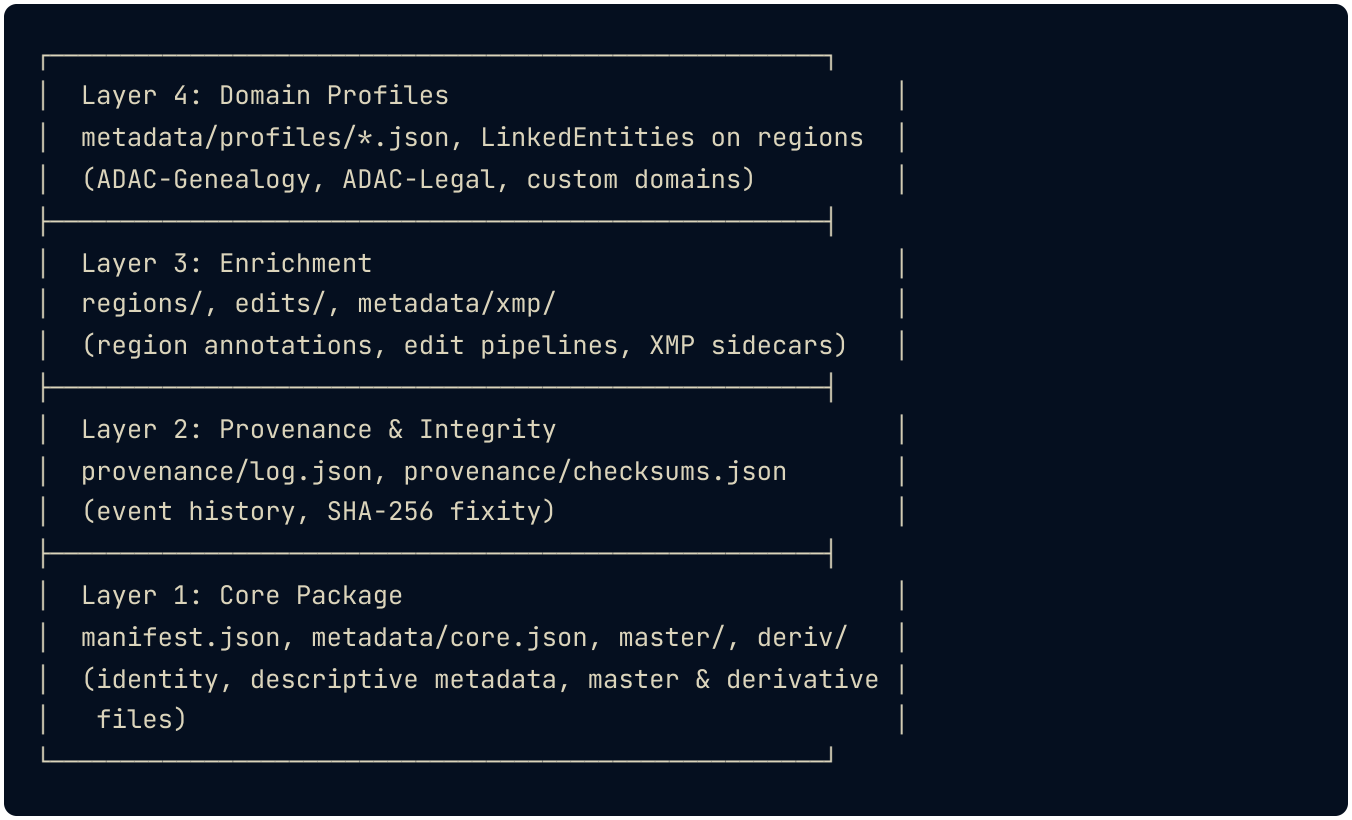
3. Terminology

Term	Definition
Container	A ZIP archive file conforming to this specification, typically with a `.adac` extension.
Master	The original, highest-fidelity digital artifact stored in the container (e.g., an uncompressed TIFF scan, a lossless WAV recording, a raw DNG capture).
Derivative	A file derived from a master for a specific purpose (e.g., a JPEG preview, an MP3 proxy, a thumbnail).
Manifest	The root JSON file (`manifest.json`) describing the container's identity, structure, and references to all content.
Core Metadata	Descriptive, technical, administrative, and preservation metadata about the container and its contents (`metadata/core.json`).
Region	A bounded area within a master artifact — a rectangular area on an image, a time segment in audio/video, or a coordinate range in a 3D model.
Linked Entity	A domain-specific JSON object attached to a region via a namespaced string key.
Profile	A domain-specific JSON metadata extension stored in `metadata/profiles/` and referenced in the manifest.
Provenance	A chronological record of actions performed on the artifact, from creation through present.
Fixity	The property of data remaining unchanged over time. Verified in ADAC through SHA-256 checksums.
Edit Pipeline	An ordered list of non-destructive editing operations recorded for a master, preserving the original artifact while describing transformations.
Encryption Descriptor	A metadata object indicating that a stored file contains pre-encrypted ciphertext. ADAC does not encrypt or decrypt — it stores opaque bytes and records descriptive metadata.

4. Architecture

4.1 Layered Model

ADAC is structured as four conceptual layers, each adding capability without requiring the layers above it:



A Minimal-conformant container requires only Layer 1. Layers 2–4 are additive and independently optional.

4.2 The Profile Extension Model

ADAC achieves domain extensibility through two complementary mechanisms:

1. **Profile files** — Self-describing JSON files in `metadata/profiles/` that carry domain-level metadata. A genealogy profile contains source citations; a legal profile contains case references. Each profile is opaque to readers that do not understand it.

2. **Linked entities** — Typed JSON objects stored in a dictionary on each region, keyed by namespaced strings (e.g., `"genealogy:person"`, `"legal:redaction"`). This allows per-region domain data to be attached without modifying the region schema itself.

Both mechanisms are designed for round-trip safety: readers that do not understand a profile or linked entity MUST preserve it unchanged during serialization.

4.3 The Fixity Model

Because an ADAC container is a living archive, the fixity model distinguishes between **immutable** content (master files) and **mutable** content (everything else).

ADAC computes SHA-256 checksums for every file in the container each time the container is saved. Checksums are stored in `provenance/checksums.json`. At any future point, a verifier can recompute all checksums and compare them against the stored values:

- **Master file checksums** serve as an immutability seal. A mismatch on a master file indicates unauthorized alteration or corruption of the original artifact — a serious integrity failure.
- **Non-master checksums** detect corruption or tampering of metadata, derivatives, and other supporting files since the last save.

This model allows the container to evolve — metadata refined, regions added, derivatives regenerated — while providing a strong guarantee that the original master artifacts remain bit-for-bit identical to the day they were ingested.

---

5. Physical Container Format

5.1 ZIP Archive

An ADAC container is a standard **ZIP archive** conforming to ISO/IEC 21320-1:2015. Any ZIP library on any platform can extract content from an ADAC container.

5.2 Directory Structure

```
<container>.adac
├── manifest.json                          (REQUIRED)
├── master/                                (REQUIRED — at least one file)
│   ├── master_0001.tif
│   ├── master_0002.tif
│   └── ...
├── metadata/                              (REQUIRED)
│   ├── core.json                          (REQUIRED)
│   ├── xmp/                               (OPTIONAL)
│   │   ├── master_0001.xmp
│   │   └── ...
│   └── profiles/                          (OPTIONAL)
│       ├── genealogy.json                 (ADAC-Genealogy profile)
│       ├── legal.json                     (ADAC-Legal profile)
│       └── ...
├── derivatives/                           (OPTIONAL)
│   ├── deriv_0001.jpg
│   └── ...
├── regions/                               (OPTIONAL)
│   ├── master-001.regions.json
│   └── ...
├── edits/                                 (OPTIONAL)
│   ├── master-001.edits.json
│   └── ...
└── provenance/                            (OPTIONAL — RECOMMENDED for archival)
    ├── log.json
    └── checksums.json
```

5.3 Path Conventions

- All paths within the container MUST use **forward slashes** (`/`) as separators, regardless of the host operating system.
- Paths are **case-sensitive**.
- Directory entries (paths ending with `/`) are informational and MUST be ignored during content processing.

5.4 Compression Rules

Content Type	Compression	Rationale
Master files	`Store` (no compression)	Archival best practice. Lossless re-compression of already-compressed formats (TIFF, DNG, WAV, FLAC) provides negligible size reduction but introduces a dependency on decompression correctness. Storing uncompressed ensures bit-perfect extraction and simplifies long-term preservation.
Derivative files	`Deflate` (optimal)	Derivatives are replaceable regenerated copies; compression saves space without archival risk.
JSON metadata	`Deflate` (optimal)	Small text files compress well and are easily regenerated.

6. Manifest

Path: `manifest.json` (container root)
Status: REQUIRED

The manifest is the entry point for any ADAC container. It establishes the container's identity, enumerates all master and derivative files, and references all metadata locations.

6.1 Schema

```
{
  "adacVersion": "1.0",
  "id": "550e8400-e29b-41d4-a716-446655440000",
  "createdOn": "2025-01-15T10:30:00Z",
  "createdBy": "Application Name v1.0",
  "description": "Optional human-readable description",
  "masters": [ ... ],
  "derivatives": [ ... ],
  "metadata": { ... }
}
```

6.2 Manifest Fields

Field	Type	Required	Description
`adacVersion`	`string`	REQUIRED	The ADAC specification version. MUST be `"1.0"` for containers conforming to this specification.
`id`	`string`	REQUIRED	A globally unique identifier for this container. UUID (RFC 4122) is RECOMMENDED.
`createdOn`	`string`	OPTIONAL	ISO-8601 timestamp of when the container was created.
`createdBy`	`string`	OPTIONAL	The application, system, or user that created the container.
`description`	`string`	OPTIONAL	A human-readable description of the container's purpose or contents.
`masters`	`MasterEntry[]`	REQUIRED	Array of master file entries. MUST contain at least one entry.
`derivatives`	`DerivativeEntry[]`	OPTIONAL	Array of derivative file entries.
`metadata`	`MetadataReference`	REQUIRED	References to metadata files within the container.

6.3 Master Entry

Each master entry describes a primary, highest-fidelity artifact stored in the container.

```
{
  "id": "master-001",
  "file": "master/master_0001.tif",
  "xmp": "metadata/xmp/master_0001.xmp",
  "regions": "regions/master-001.regions.json",
  "edits": "edits/master-001.edits.json",
  "encryption": null
}
```

Field	Type	Required	Description
`id`	`string`	REQUIRED	Unique identifier for this master within the container.
`file`	`string`	REQUIRED	Container-relative path to the master file. The file MUST exist at this path.
`xmp`	`string`	OPTIONAL	Container-relative path to the XMP sidecar for this master.
`regions`	`string`	OPTIONAL	Container-relative path to the region annotation file for this master.
`edits`	`string`	OPTIONAL	Container-relative path to the edit pipeline file for this master.
`encryption`	`EncryptionDescriptor`	OPTIONAL	Indicates the stored file is pre-encrypted ciphertext (see Section 17).

6.4 Derivative Entry

Each derivative entry describes a file produced from a master — a compressed preview, thumbnail, web-optimized copy, etc.

```
{
  "id": "preview-001",
  "file": "derivatives/deriv_0001.jpg",
  "sourceMasterId": "master-001",
  "purpose": "web-preview",
  "encryption": null
}
```

Field	Type	Required	Description
`id`	`string`	REQUIRED	Unique identifier for this derivative within the container.
`file`	`string`	REQUIRED	Container-relative path to the derivative file. The file MUST exist at this path.
`sourceMasterId`	`string`	OPTIONAL	The `id` of the master this derivative was produced from. SHOULD reference a valid master `id`.
`purpose`	`string`	OPTIONAL	The intended use (e.g., `"web-preview"`, `"thumbnail"`, `"print-proof"`).
`encryption`	`EncryptionDescriptor`	OPTIONAL	Indicates the stored file is pre-encrypted ciphertext (see Section 17).

6.5 Metadata Reference

The metadata reference object tells readers where to find each category of metadata within the container.

```
{
  "core": "metadata/core.json",
  "profiles": [
    "metadata/profiles/genealogy.json",
    "metadata/profiles/legal.json"
  ],
  "provenanceLog": "provenance/log.json",
  "checksums": "provenance/checksums.json"
}
```

Field	Type	Required	Description
`core`	`string`	REQUIRED	Path to the core metadata file.
`profiles`	`string[]`	OPTIONAL	Paths to domain-specific profile files.
`provenanceLog`	`string`	OPTIONAL	Path to the provenance log.
`checksums`	`string`	OPTIONAL	Path to the checksum manifest.

7. Core Metadata

Path: `metadata/core.json`
Status: REQUIRED

Core metadata provides descriptive, technical, administrative, and preservation information about the container and its contents. The descriptive fields align with the Dublin Core Metadata Element Set where applicable.

7.1 Schema

```
{
  "id": "550e8400-e29b-41d4-a716-446655440000",
  "title": "1870 Census, Licking County, Ohio — Page 42",
  "creator": "National Archives",
  "subject": "census, genealogy, Ohio",
  "description": "Page 42 of the 1870 Federal Census enumeration for Newark, Licking C
  "dateCreated": "2025-01-15T10:00:00Z",
  "source": "National Archives microfilm T9, Roll 1042, Frame 142",
  "format": "TIFF",
  "language": "en",
  "coverage": "United States, Ohio, Licking County, Newark",
  "tags": ["census", "1870", "ohio", "licking-county"],
  "rights": { ... },
  "technical": { ... },
  "administrative": { ... },
  "preservation": { ... }
}
```

7.2 Descriptive Fields

Field	Type	Required	Dublin Core	Description
`id`	`string`	REQUIRED	`dc:identifier`	MUST match the manifest `id`.
`title`	`string`	OPTIONAL	`dc:title`	A human-readable title for the artifact.
`creator`	`string`	OPTIONAL	`dc:creator`	The original creator of the artifact.
`subject`	`string`	OPTIONAL	`dc:subject`	Subject keywords (comma-separated or free text).
`description`	`string`	OPTIONAL	`dc:description`	Detailed description of the content.
`dateCreated`	`string`	OPTIONAL	`dc:date`	ISO-8601 timestamp of when the artifact was originally created.
`source`	`string`	OPTIONAL	`dc:source`	The source from which this artifact was derived.
`format`	`string`	OPTIONAL	`dc:format`	The primary format (e.g., `"TIFF"`, `"WAV"`, `"DNG"`).
`language`	`string`	OPTIONAL	`dc:language`	Language of the content (IETF BCP 47 RECOMMENDED).
`coverage`	`string`	OPTIONAL	`dc:coverage`	Geographic or temporal coverage.
`tags`	`string[]`	OPTIONAL	—	Classification tags for retrieval and categorization.

7.3 Rights

```
{
  "rights": {
    "statement": "Public domain",
    "holder": "National Archives",
    "license": "CC0-1.0",
    "accessRestrictions": "None"
  }
}
```

Field	Type	Description
`statement`	`string`	General rights statement.
`holder`	`string`	The entity holding rights to the artifact.
`license`	`string`	SPDX license identifier or free-text license description.
`accessRestrictions`	`string`	Access restriction details (e.g., `"None"`, `"Restricted — protective order"`, `"Embargoed until 2030"`).

7.4 Technical Metadata

Technical metadata captures the digitization or recording parameters of the artifact. Fields are OPTIONAL and media-dependent — populate only the fields relevant to the media type.

```
{
  "technical": {
    "scanner": "Epson Expression 12000XL",
    "dpi": 600,
    "bitDepth": 24,
    "colorSpace": "sRGB",
    "duration": "PT5M30S",
    "frameRate": 30.0,
    "sampleRate": 44100,
    "pageCount": 2
  }
}
```

Field	Type	Media	Description
`scanner`	`string`	Image	The scanning device or capture system.
`dpi`	`integer`	Image	Scanning resolution in dots per inch.
`bitDepth`	`integer`	Image	Bit depth per channel.
`colorSpace`	`string`	Image	Color space (e.g., `"sRGB"`, `"AdobeRGB"`, `"ProPhotoRGB"`, `"grayscale"`).
`duration`	`string`	Audio/Video	Duration (ISO 8601 duration or timecode string).
`frameRate`	`number`	Video	Video frame rate in frames per second.
`sampleRate`	`integer`	Audio	Audio sample rate in Hz.
`pageCount`	`integer`	Document	Number of pages (for multi-page/multi-master documents).

7.5 Administrative Metadata

Administrative metadata records the institutional context — which repository holds the original, under what accession or catalog number, and who digitized it.

```
{
  "administrative": {
    "repository": "InnoVadens Digital Archive",
    "accessionNumber": "2025-001-042",
    "catalogNumber": "CENSUS-OH-1870-042",
    "digitizedBy": "John Smith",
    "digitizedOn": "2025-01-15T10:00:00Z"
  }
}
```

Field	Type	Description
`repository`	`string`	The archival repository holding this artifact.
`accessionNumber`	`string`	Accession number within the repository.
`catalogNumber`	`string`	Catalog number for retrieval.
`digitizedBy`	`string`	Person or system that performed digitization.
`digitizedOn`	`string`	ISO-8601 timestamp of when digitization occurred.

7.6 Preservation Metadata

Preservation metadata provides summary counts that enable quick inventorying without scanning the entire manifest.

```
{
  "preservation": {
    "masterCount": 2,
    "derivativeCount": 1
  }
}
```

Field	Type	Description
`masterCount`	`integer`	Number of master files in the container.
`derivativeCount`	`integer`	Number of derivative files in the container.

8. Master Files

Directory: `master/`
Status: REQUIRED — at least one master file MUST be present

Master files are the primary, highest-fidelity digital artifacts. They represent the definitive record of the digitized source and are the **only content in the container**

considered immutable. Once a master file is ingested into an ADAC container, it MUST NOT be modified or replaced. All other container content — metadata, derivatives, regions, edits, provenance — is expected to evolve over the container's lifetime. SHA-256 checksums for master files serve as an immutability seal, enabling any future verifier to confirm that the original artifacts remain unaltered.

8.1 Naming Convention

When no explicit container path is specified, master files SHOULD be named using a zero-padded sequential pattern:

```
master/master_NNNN.<ext>
```

Where `NNNN` is a 1-based, zero-padded sequence number and `<ext>` is the original file extension (e.g., `master_0001.tif`, `master_0002.wav`).

8.2 Storage

Master files MUST be stored with `Store` (no compression) in the ZIP archive. This is a deliberate archival design decision:

- Archival-grade formats (TIFF, DNG, WAV, FLAC) are already efficiently encoded. Re-compression provides negligible size reduction.
- Uncompressed storage enables bit-perfect extraction using any ZIP tool.
- Eliminates a dependency on a specific compression algorithm's long-term availability.
- Simplifies fixity verification — the stored bytes are the artifact bytes.

8.3 Format Recommendations

ADAC does not restrict the media format of master files. However, for long-term preservation, the following formats are RECOMMENDED:

Media Type	Recommended Formats
Images	TIFF (uncompressed or lossless), DNG, PNG
Audio	WAV (PCM), FLAC, BWF
Video	FFV1 in MKV, uncompressed AVI
3D Models	glTF, OBJ, STL
Documents	PDF/A

8.4 Constraints

- Every master entry MUST have a unique `id` within the manifest.
- The master file MUST exist at the path referenced by its manifest entry.
- There is no limit on the number of masters per container.
- A master file MUST NOT be modified or replaced after initial ingestion. Its SHA-256 checksum, recorded in `provenance/checksums.json`, serves as an immutability seal.
- Additional masters MAY be added to the container in subsequent saves. Each new master receives its own checksum at the time of ingestion, which then becomes its immutability baseline.

---

9. Derivative Files

Directory: `derivatives/`
Status: OPTIONAL

Derivatives are files produced from masters for specific purposes — compressed previews, thumbnails, web-optimized versions, print proofs, etc. They are considered regenerable and are not archivally significant.

9.1 Naming Convention

Default pattern: `derivatives/deriv_NNNN.<ext>`

9.2 Storage

Derivative files are stored with `Deflate` (optimal) compression in the ZIP archive, since they are replaceable and compression provides meaningful space savings.

9.3 Constraints

- Every derivative entry MUST have a unique `id` within the manifest.
- The derivative file MUST exist at the path referenced by its manifest entry.
- The `sourceMasterId` SHOULD reference a valid master `id` in the same container.

---

10. Region Annotations

Directory: `regions/`
Status: OPTIONAL

Region annotations mark bounded areas within master artifacts. A region can be a rectangular area on a scanned image, a time segment in an audio or video recording, or a coordinate range in a 3D model. Regions provide the structural backbone for domain-specific enrichment through the linked entities mechanism.

10.1 File Naming

Each region annotation file is associated with a specific master and is named using the master's `id`:

```
regions/<masterId>.regions.json
```

10.2 Schema

```
{
  "mediaId": "master-001",
  "coordinateSystem": "pixel",
  "width": 4800,
  "height": 6400,
  "regions": [
    {
      "id": "region-001",
      "type": "boundingBox",
      "label": "Household entry",
      "bounds": {
        "x": 100.0,
        "y": 200.0,
        "width": 4600.0,
        "height": 80.0
      },
      "linkedEntities": {
        "genealogy:person": { ... },
        "legal:redaction": { ... }
      }
    }
  ]
}
```

10.3 Annotation-Level Fields

Field	Type	Required	Description
`mediaId`	`string`	OPTIONAL	The master identifier this annotation applies to.
`coordinateSystem`	`string`	OPTIONAL	Coordinate system (see Section 10.4). Defaults to `"pixel"`.
`width`	`integer`	OPTIONAL	Media width in pixels (for `pixel` system).
`height`	`integer`	OPTIONAL	Media height in pixels (for `pixel` system).
`duration`	`string`	OPTIONAL	Media duration (for `timecode` system).
`frames`	`integer`	OPTIONAL	Total frame count (for video).
`regions`	`Region[]`	REQUIRED	The annotated regions.

10.4 Coordinate Systems

ADAC supports three coordinate systems to accommodate different media types:

Value	Description	Relevant Bounds Properties
`pixel`	2D pixel coordinates on an image. Origin is top-left.	`x`, `y`, `width`, `height`
`timecode`	Time-based segments in audio or video.	`start`, `end`
`3d`	Three-dimensional coordinate system.	`x`, `y`, `z`, `width`, `height`

Implementations MAY define additional coordinate systems. Unknown coordinate systems MUST be preserved during round-trip serialization.

10.5 Region Types

Value	Description
`boundingBox`	A rectangular area defined by position and size.
`timeSegment`	A span of time defined by start and end timecodes.
`point`	A single point (x, y, or x, y, z).

Implementations MAY define additional region types. Unknown region types MUST be preserved during round-trip serialization.

10.6 Region Fields

Field	Type	Required	Description
`id`	`string`	REQUIRED	Unique identifier for this region within the annotation file.
`type`	`string`	REQUIRED	The region type (see Section 10.5).
`label`	`string`	OPTIONAL	A human-readable label for the region.
`bounds`	`Bounds`	OPTIONAL	The spatial or temporal bounds of the region.
`linkedEntities`	`object`	OPTIONAL	A dictionary of namespaced domain data objects (see Section 10.8).

10.7 Bounds Fields

Field	Type	Description
`x`	`number`	X coordinate (pixel or 3D origin).
`y`	`number`	Y coordinate (pixel or 3D origin).
`width`	`number`	Width (pixel-based or 3D bounds).
`height`	`number`	Height (pixel-based or 3D bounds).
`start`	`string`	Start timecode (time-based bounds).
`end`	`string`	End timecode (time-based bounds).
`z`	`number`	Z coordinate (3D bounds).

All bounds fields are OPTIONAL. Populate only the fields relevant to the coordinate system.

10.8 Linked Entities

The `linkedEntities` property on each region is a **dictionary** (JSON object) where:

- **Keys** are namespaced strings identifying the domain and data type (e.g., `"genealogy:person"`, `"legal:redaction"`, `"medical:diagnosis"`).
- **Values** are arbitrary JSON objects whose schema is defined by the domain profile that owns the namespace.

This mechanism allows any number of domain-specific profiles to attach metadata to the same region without conflict and without modifying the core ADAC region schema.

Round-trip safety: An ADAC reader that does not understand a particular linked entity key MUST preserve the key and its JSON value unchanged when the container is serialized back. This guarantees that domain data is never lost by generic tools.

Namespace convention: Keys SHOULD use the pattern `<domain>:<type>` (e.g., `"genealogy:person"`, `"legal:exhibit"`). This convention prevents collisions between profiles from different domains.

11. Edit Pipelines

Directory: `edits/`
Status: OPTIONAL

Edit pipelines describe non-destructive editing operations applied to a master. They record what transformations were (or should be) applied — crops, rotations, level adjustments, sharpening — without modifying the original master file.

11.1 File Naming

```
edits/<masterId>.edits.json
```

11.2 Schema

```
{
  "mediaId": "master-001",
  "pipelineVersion": "1.0",
  "coordinateSpace": "pixel",
  "referenceWidth": 4800,
  "referenceHeight": 6400,
  "operations": [
    {
      "id": "op-001",
      "type": "crop",
      "parameters": {
        "x": 100,
        "y": 200,
        "width": 800,
        "height": 600
      },
      "appliesToDerivativeIds": ["preview-001"]
    },
    {
      "id": "op-002",
      "type": "rotate",
      "parameters": {
        "angle": 90
      },
      "appliesToDerivativeIds": ["preview-001"]
    }
  ]
}
```

11.3 Pipeline Fields

Field	Type	Required	Description
`mediaId`	`string`	OPTIONAL	The master this pipeline applies to.
`pipelineVersion`	`string`	OPTIONAL	Version of the pipeline schema.
`coordinateSpace`	`string`	OPTIONAL	The coordinate space for spatial parameters. See Section 11.6. Defaults to `"pixel"`.
`referenceWidth`	`integer`	OPTIONAL	Reference width in pixels (REQUIRED when `coordinateSpace` is `"pixel"`). Spatial parameters are interpreted relative to this dimension.
`referenceHeight`	`integer`	OPTIONAL	Reference height in pixels (REQUIRED when `coordinateSpace` is `"pixel"`). Spatial parameters are interpreted relative to this dimension.
`operations`	`EditOperation[]`	REQUIRED	Ordered list of editing operations.

11.4 Edit Operation Fields

Field	Type	Required	Description
`id`	`string`	REQUIRED	Unique operation identifier within this pipeline.
`type`	`string`	REQUIRED	Operation type (e.g., `"crop"`, `"rotate"`, `"levels"`, `"sharpen"`, `"deskew"`).
`parameters`	`object`	OPTIONAL	Operation-specific parameters. The schema of this object depends on the operation `type`. Spatial parameters (coordinates, dimensions) are interpreted in the pipeline's `coordinateSpace`.
`appliesToDerivativeIds`	`string[]`	OPTIONAL	Derivative `id` values that were produced using this operation.

11.5 Design Rationale

Edit pipelines support the archival principle that the original artifact must be preserved unchanged. Rather than modifying the master file, transformations are recorded as

metadata. Applications can:

- Replay the pipeline to produce a derivative on demand.
- Display the original and edited versions side by side.
- Audit what transformations were applied and when.

11.6 Coordinate Spaces

Spatial parameters in edit operations (e.g., crop coordinates, deskew origin points) require a defined coordinate space so that pipelines can be replayed correctly against masters and derivatives at different resolutions.

Value	Description
`pixel`	Absolute pixel coordinates relative to the master's native resolution. When this space is used, `referenceWidth` and `referenceHeight` MUST be provided at the pipeline level so that implementations can scale parameters when applying operations to derivatives at different resolutions.
`normalized`	Resolution-independent coordinates in the range 0.0–1.0, where (0.0, 0.0) is the top-left corner and (1.0, 1.0) is the bottom-right corner. This space is inherently resolution-independent and does not require `referenceWidth` or `referenceHeight`.

If `coordinateSpace` is omitted, implementations MUST assume `"pixel"`. Implementations MAY define additional coordinate spaces; unknown values MUST be preserved during round-trip serialization.

Operations that do not involve spatial parameters (e.g., `"levels"`, `"sharpen"`) are unaffected by the coordinate space setting.

---

12. XMP Sidecars

Directory: `metadata/xmp/`
Status: OPTIONAL

XMP (Extensible Metadata Platform) sidecar files contain industry-standard metadata in XML format, compatible with Adobe tools, ExifTool, and the broader imaging ecosystem.

12.1 File Naming

Each XMP sidecar corresponds to a master file and is named using the master's base filename:

```
metadata/xmp/<masterBaseName>.xmp
```

The XMP file name is derived from the master's container path — the base file name with the `.xmp` extension (e.g., master file `master/master_0001.tif` → XMP sidecar `metadata/xmp/master_0001.xmp`).

12.2 Relationship to Core Metadata

XMP sidecars are complementary to core metadata. Core metadata (`metadata/core.json`) is ADAC's native, structured metadata format. XMP sidecars provide interoperability with the existing imaging ecosystem. When both exist, the core metadata is authoritative for ADAC consumers.

---

13. Profile Extensions

Directory: `metadata/profiles/`
Status: OPTIONAL

Profile extensions are the primary extensibility mechanism in ADAC. They allow domain-specific metadata to be added to a container without modifying the core specification.

13.1 Mechanism

1. A domain-specific JSON file is placed in `metadata/profiles/` (e.g., `metadata/profiles/genealogy.json`).

2. The file path is added to the manifest's `metadata.profiles` array.
3. Non-profile-aware readers ignore the file entirely.
4. Profile-aware readers locate the file by matching the filename against a well-known name.

13.2 Design Principles

- Profiles are **additive** — they MUST NOT modify core metadata structures or semantics.
- Profiles are **self-describing** — every profile file MUST include a `profileType` and `profileVersion` field at the root level.
- Profiles are **ignorable** — readers that do not recognize a profile type MUST NOT reject the container.
- Profiles are **composable** — a container MAY include multiple profiles simultaneously (e.g., both a genealogy profile and a legal profile).

13.3 Profile Root Fields

Every profile file MUST contain at minimum:

```
{
  "profileVersion": "1.0",
  "profileType": "<domain>"
}
```

Field	Type	Required	Description
`profileVersion`	`string`	REQUIRED	The version of the profile schema.
`profileType`	`string`	REQUIRED	A unique string identifying the profile domain (e.g., `"genealogy"`, `"legal"`, `"medical"`).

13.4 Well-Known Profiles

The following profiles are defined as companion specifications:

Profile Type	File Name	Specification
`genealogy`	`genealogy.json`	ADAC-Genealogy 1.0 — Genealogy Profile Format Specification
`legal`	`legal.json`	ADAC-Legal 1.0 — Legal Document Profile Format Specification

13.5 Custom Profiles

Any organization or domain may define custom profiles. To avoid collisions, custom profile types SHOULD use a reverse-domain prefix (e.g., `"com.example.radiology"`) or a clearly domain-specific term.

---

14. Provenance Log

Path: `provenance/log.json`
Status: OPTIONAL (RECOMMENDED for archival-grade containers)

The provenance log records a chronological sequence of events describing the lifecycle of the artifact — from initial capture or scanning, through editing and derivative creation, to export and distribution.

14.1 Schema

```
{
  "events": [
    {
      "id": "evt-001",
      "type": "scan",
      "timestamp": "2025-01-15T10:00:00Z",
      "actor": "John Smith",
      "software": "GeneaScan v1.0",
      "details": {
        "resolution": "600 dpi",
        "format": "TIFF",
        "scanner": "Epson Expression 12000XL"
      }
    },
    {
      "id": "evt-002",
      "type": "derivativeCreated",
      "timestamp": "2025-01-15T10:01:00Z",
      "actor": "GeneaScan v1.0",
      "software": "GeneaScan v1.0",
      "details": {
        "derivativeId": "preview-001",
        "format": "JPEG",
        "quality": 85
      }
    }
  ]
}
```

14.2 Event Fields

Field	Type	Required	Description
`id`	`string`	REQUIRED	Unique event identifier within this log.
`type`	`string`	REQUIRED	Event type. Well-known values include `"scan"`, `"import"`, `"edit"`, `"derivativeCreated"`, `"export"`, `"validate"`, `"save"`. Custom values are permitted.
`timestamp`	`string`	REQUIRED	ISO-8601 timestamp of when the event occurred.
`actor`	`string`	REQUIRED	Person, system, or service responsible for the action (e.g., `"Jane Doe"`, `"IngestService"`).
`software`	`string`	OPTIONAL	Software and version that executed the action (e.g., `"GeneaScan v1.0"`). In automated pipelines where the software is the actor, this MAY be omitted to avoid redundancy.
`details`	`object`	OPTIONAL	Event-specific key-value pairs. The schema is flexible and event-type-dependent.

14.3 Event Ordering

Events SHOULD be recorded in chronological order (earliest first). Consumers SHOULD NOT assume strict ordering and MAY sort by `timestamp` if needed.

14.4 Well-Known Event Types

The following event types are defined for interoperability. Implementations MAY define additional custom event types.

Event Type	Description
`scan`	A physical source was digitized to produce a master file.
`import`	A digital file was ingested into the container as a master.
`edit`	An edit pipeline was created or modified. When an application updates an edit pipeline file, it SHOULD also append a corresponding `"edit"` provenance event so the change is recorded chronologically.
`derivativeCreated`	A derivative file was generated from a master.
`export`	The container was written to persistent storage.
`validate`	A fixity or structural validation was performed on the container. The `details` object SHOULD include the validation result (e.g., `{"result": "passed"}` or `{"result": "failed", "errors": [...]}`) so that the verification history is auditable.
`save`	The container was re-saved after modification. Checksums were recomputed.

15. Checksum Manifest

Path: `provenance/checksums.json`
Status: OPTIONAL (RECOMMENDED for archival-grade containers)

The checksum manifest stores SHA-256 hashes for every file in the container, enabling integrity verification at any point in the future without the original source files. Because an ADAC container is a living archive, checksums are recomputed each time the container is saved. Master file checksums serve as an immutability seal — they MUST remain identical across every save. A changed master checksum indicates unauthorized alteration or corruption.

15.1 Schema

```
{
  "algorithm": "sha256",
  "files": [
    {
      "path": "manifest.json",
      "checksum": "a1b2c3d4e5f67890abcdef1234567890abcdef1234567890abcdef1234567890"
    },
    {
      "path": "master/master_0001.tif",
      "checksum": "f6e5d4c3b2a109876543210fedcba9876543210fedcba9876543210fedcba987"
    }
  ]
}
```

15.2 Fields

Field	Type	Required	Description
`algorithm`	`string`	REQUIRED	The hash algorithm. MUST be `"sha256"` for ADAC 1.0.
`files`	`FileChecksum[]`	REQUIRED	Array of path/checksum pairs.
`files[].path`	`string`	REQUIRED	Container-relative path to the file.
`files[].checksum`	`string`	REQUIRED	Lowercase hexadecimal SHA-256 hash of the file's contents.

15.3 Coverage

The checksum manifest SHOULD include a hash for every file in the container except itself. The `checksums.json` file MUST NOT contain its own hash (this would create a circular dependency).

15.4 Computation Timing

Checksums MUST be recomputed each time the container is saved — as each file's bytes flow into the ZIP archive, they are simultaneously fed to a SHA-256 hash computation. This single-pass approach avoids the need to re-read every file after writing.

Because the container is a living archive, non-master checksums will naturally change between saves as metadata, derivatives, and other supporting files are added, modified, or removed. Master file checksums, however, MUST remain stable across saves — any change indicates an integrity violation (see Section 8).

15.5 Write-Order Dependency

Because `manifest.json` references `provenance/checksums.json` by path, and `checksums.json` records a hash of `manifest.json`, there is an inherent ordering dependency during the write pass. Writers MUST follow this sequence:

1. Write all content files (masters, derivatives, metadata, profiles, regions, edits, XMP sidecars, provenance log) into the ZIP archive, computing SHA-256 hashes as each file is written.
2. Finalize `manifest.json` content (all paths and references are now known).
3. Write `manifest.json` into the archive and compute its SHA-256 hash.
4. Write `provenance/checksums.json` — containing hashes for all files including `manifest.json` — as the **last** entry in the archive.

`checksums.json` is always the final file written. It MUST NOT contain its own hash (see Section 15.3).

---

16. Fixity & Integrity Verification

Fixity verification compares stored checksums against freshly computed hashes to detect any bit-level tampering or corruption.

16.1 Verification Procedure

1. Open the ADAC container (ZIP archive).
2. Read `provenance/checksums.json`. If absent, fixity verification is not possible — report accordingly.
3. For each entry in the checksum manifest:
    1. Locate the corresponding ZIP entry by its `path`.
    2. If the entry does not exist in the archive, record it as a **missing file**.
    3. If the entry exists, read its entire content and compute the SHA-256 hash.
    4. Compare the computed hash (lowercase hexadecimal) against the stored `checksum` value (ordinal, case-sensitive comparison).
    5. If they differ, record it as a **mismatch**.
4. Produce a fixity report.

16.2 Fixity Report

A fixity report contains:

Field	Type	Description
`isValid`	`boolean`	`true` if no mismatches and no missing files.
`totalFiles`	`integer`	Total files listed in the checksum manifest.
`verifiedFiles`	`integer`	Files whose checksums matched.
`failedFiles`	`integer`	Files with hash mismatches.
`missingFiles`	`integer`	Files listed in the manifest but absent from the container.
`mismatches`	`Mismatch[]`	Details of each failure (path, expected hash, computed hash).

17. Encryption Descriptor

Status: OPTIONAL

An encryption descriptor is a purely informational metadata object that indicates a stored file is pre-encrypted ciphertext. **ADAC never performs encryption or decryption** — it stores opaque bytes bit-for-bit and records descriptive metadata so consumers know what they are looking at.

17.1 Schema

```
{
  "algorithm": "AES-256-GCM",
  "keyId": "vault://keys/archive-2025",
  "originalMediaType": "image/tiff"
}
```

17.2 Fields

Field	Type	Required	Description
`algorithm`	`string`	REQUIRED*	The encryption algorithm used to produce the ciphertext (e.g., `"AES-256-GCM"`, `"ChaCha20-Poly1305"`). Free-form informational string — ADAC does not interpret or validate it.
`keyId`	`string`	OPTIONAL	An opaque reference to the decryption key. The meaning is defined by the consumer's key management system (e.g., a UUID, a key vault path, a PGP fingerprint).
`originalMediaType`	`string`	OPTIONAL	The MIME type of the original unencrypted content (e.g., `"image/tiff"`). Since the stored bytes are ciphertext, the file extension may not reflect the actual media type.

* If an encryption descriptor is present, the `algorithm` field SHOULD be non-empty. An empty `algorithm` produces a validation warning.

17.3 Fixity Implications

When encryption is present, SHA-256 fixity checksums cover the **ciphertext** bytes — the bytes actually stored in the container. This is by design: the container verifies the integrity of what is stored, not the plaintext content. The consuming application is responsible for decryption using whatever key management system the `keyId` references.

---

18. JSON Conventions

All JSON files within an ADAC container MUST follow these conventions:

Convention	Requirement	Rationale
Property naming	camelCase	Aligns with JSON ecosystem conventions. `profileVersion`, not `ProfileVersion` or `profile_version`.
Formatting	Indented (2 or 4 spaces)	Human readability for archival inspection.
Null handling	Null-valued properties SHOULD be omitted	Reduces file size and avoids ambiguity between "null" and "absent".
Encoding	UTF-8 without BOM	Universal text encoding for long-term compatibility.
Unknown properties	MUST be preserved during round-trip serialization	Guarantees forward compatibility. A reader that encounters unknown fields MUST pass them through unchanged when writing.
Date/time format	ISO-8601	Unambiguous, timezone-aware, internationally recognized.
Numeric precision	IEEE 754 double-precision for floating-point	Standard JSON number representation.

---

19. Validation & Conformance Levels

ADAC validation checks containers for structural correctness. Findings are reported at three severity levels:

Severity	Meaning
Error	Required structure is missing or corrupt. The container is **non-conformant**.
Warning	Recommended content is missing or inconsistent. The container is technically valid but deviates from best practices.
Info	Optional content is absent. Informational only.

19.1 Error Codes

Code	Severity	Description
`ADAC-001`	Error	Container file does not exist.
`ADAC-002`	Error	File is not a valid ZIP archive.
`ADAC-010`	Error	`manifest.json` is missing or cannot be parsed as JSON.
`ADAC-011`	Error	Manifest `adacVersion` field is missing or empty.
`ADAC-012`	Error	Manifest `id` field is missing or empty.
`ADAC-020`	Error	Manifest contains no master file entries.
`ADAC-021`	Error	A master entry has an empty `id`.
`ADAC-022`	Error	A master file path referenced in the manifest does not exist in the container.
`ADAC-023`	Error	A region file referenced by a master entry does not exist in the container.
`ADAC-024`	Error	An edit file referenced by a master entry does not exist in the container.
`ADAC-025`	Error	An XMP file referenced by a master entry does not exist in the container.
`ADAC-030`	Error	A derivative file path referenced in the manifest does not exist in the container.
`ADAC-040`	Error	Core metadata file (`metadata/core.json`) is missing or cannot be parsed as JSON.

Code	Severity	Description
`ADAC-050`	Error	A profile file referenced in the manifest's `metadata.profiles` array does not exist in the container.
`ADAC-060`	Error	Provenance log referenced in the manifest does not exist in the container.
`ADAC-070`	Error	Checksums file referenced in the manifest does not exist in the container.
`ADAC-080`	Error	Checksums file cannot be parsed as JSON.
`ADAC-081`	Error	A file listed in the checksum manifest does not exist in the container.
`ADAC-082`	Error	SHA-256 checksum mismatch — stored hash does not match the computed hash.

19.2 Warning Codes

Code	Severity	Description
`ADAC-026`	Warning	A master entry has an encryption descriptor but the `algorithm` field is empty.
`ADAC-031`	Warning	A derivative references an unknown `sourceMasterId` (no matching master `id`).
`ADAC-032`	Warning	A derivative entry has an encryption descriptor but the `algorithm` field is empty.
`ADAC-041`	Warning	Core metadata `id` is empty.
`ADAC-042`	Warning	Core metadata `id` does not match the manifest `id`.
`ADAC-061`	Warning	No provenance log is referenced (RECOMMENDED for archival containers).
`ADAC-071`	Warning	No checksums file is referenced (RECOMMENDED for fixity verification).

19.3 Validation Options

Validators MAY support the following options:

Option	Default	Description
Verify checksums	`true`	Whether to recompute and verify all SHA-256 checksums.
Warn on missing provenance	`true`	Whether to produce `ADAC-061` when no provenance log is present.
Warn on missing checksums	`true`	Whether to produce `ADAC-071` when no checksums file is present.

20. Media Types & File Extensions

Format	Extension	MIME Type
ADAC Container	`.adac`	`application/vnd.adac.container`

All ADAC containers — regardless of which domain profiles they include — use the `.adac` file extension. The profile type is determined by reading the container's metadata, not by the file extension.

---

21. Security Considerations

21.1 No Built-In Encryption

ADAC does not perform encryption or decryption. When sensitive content requires confidentiality, the producing application MUST encrypt the content before placing it in the container and record the encryption descriptor (see Section 17). ADAC stores the ciphertext as opaque bytes.

21.2 No Signatures

ADAC 1.0 does not define a digital signature mechanism. The checksum manifest provides **integrity** (tamper detection) but not **authenticity** (proof of origin). Organizations requiring authenticity SHOULD apply external digital signatures (e.g., CMS/PKCS#7, PGP) to the container file.

21.3 Path Traversal

ADAC implementations MUST sanitize paths extracted from the ZIP archive to prevent path traversal attacks (e.g., entries containing `../` components that could escape the extraction directory). Paths within the container MUST be relative and MUST NOT contain parent directory references.

21.4 Zip Bombs

ADAC implementations SHOULD impose reasonable limits on decompressed file sizes and total entry counts to mitigate ZIP bomb attacks (e.g., deeply nested compression or extremely high compression ratios).

21.5 Sensitive Metadata

Profile extensions (legal, medical, etc.) may contain sensitive metadata — case numbers, party names, Social Security numbers, medical identifiers. Implementations that store ADAC containers SHOULD apply appropriate access controls at the file system or storage layer. The `confidentialityLevel` field in certain profiles is informational metadata, not an enforcement mechanism.

21.6 Regulatory and Compliance Frameworks

ADAC does not perform encryption, access control, or audit logging — it stores content and descriptive metadata about that content. Implementations that use ADAC containers for regulated content (e.g., healthcare records subject to HIPAA, personal data subject to GDPR, criminal justice information subject to CJIS) are solely responsible for ensuring compliance with applicable frameworks, including key management, access control, data retention, and audit logging external to the container format itself.

---

22. Interoperability

22.1 Round-Trip Preservation

Because ADAC containers are living archives — designed to be opened, enriched, and re-saved — the single most important interoperability requirement is **round-trip preservation**: when an implementation reads a container, modifies some content, and writes it back, **all data that the implementation did not intentionally modify MUST be preserved unchanged**. In particular, master files MUST be written back bit-for-bit identical. This includes:

- Unknown JSON properties in all metadata files
- Profile files the implementation does not understand
- Linked entity keys the implementation does not recognize
- Region annotations on masters the implementation did not modify

22.2 Forward Compatibility

ADAC is designed for decades of use. Future versions of this specification may add:

- New top-level fields in the manifest
- New fields in core metadata
- New coordinate systems for regions
- New region types
- New well-known profile types

Existing implementations that tolerate unknown properties (as REQUIRED by Section 2.2) will continue to function correctly with containers produced by future versions.

22.3 Tool Compatibility

Because ADAC containers are standard ZIP archives:

- Any ZIP tool can list and extract contents.
- Any text editor can inspect JSON metadata.
- Any image viewer can display extracted master files.
- No ADAC-specific software is required for basic access.

22.4 Profile Coexistence

Multiple profiles MAY coexist in the same container without conflict:

- Each profile occupies its own file in `metadata/profiles/`.
- Linked entity keys use namespaced strings to avoid collisions.
- A container could simultaneously carry genealogy, legal, and medical profiles, each adding domain metadata to the same masters and regions.

23. Complete Container Example

The following example shows a complete Archival-conformant ADAC container for a two-page scanned document:

23.1 Container Structure

```
census-1870-licking-oh-042.adac
├── manifest.json
├── master/
│   ├── master_0001.tif              (600 DPI TIFF, page 1)
│   └── master_0002.tif              (600 DPI TIFF, page 2)
├── metadata/
│   ├── core.json
│   ├── xmp/
│   │   ├── master_0001.xmp
│   │   └── master_0002.xmp
│   └── profiles/
│       └── genealogy.json           (ADAC-Genealogy profile)
├── derivatives/
│   └── deriv_0001.jpg               (JPEG preview of page 1)
├── regions/
│   └── master-001.regions.json
├── edits/
│   └── master-001.edits.json
└── provenance/
    ├── log.json
    └── checksums.json
```

23.2 Manifest (`manifest.json`)

```
{
  "adacVersion": "1.0",
  "id": "550e8400-e29b-41d4-a716-446655440000",
  "createdOn": "2025-01-15T10:30:00Z",
  "createdBy": "GeneaScan v1.0",
  "description": "1870 U.S. Federal Census, Licking County, Ohio — Page 42",
  "masters": [
    {
      "id": "master-001",
      "file": "master/master_0001.tif",
      "xmp": "metadata/xmp/master_0001.xmp",
      "regions": "regions/master-001.regions.json",
      "edits": "edits/master-001.edits.json"
    },
    {
      "id": "master-002",
      "file": "master/master_0002.tif",
      "xmp": "metadata/xmp/master_0002.xmp"
    }
  ],
  "derivatives": [
    {
      "id": "preview-001",
      "file": "derivatives/deriv_0001.jpg",
      "sourceMasterId": "master-001",
      "purpose": "web-preview"
    }
  ],
  "metadata": {
    "core": "metadata/core.json",
    "profiles": [
      "metadata/profiles/genealogy.json"
    ],
    "provenanceLog": "provenance/log.json",
    "checksums": "provenance/checksums.json"
  }
}
```

23.3 Core Metadata (`metadata/core.json`)


```
{
  "id": "550e8400-e29b-41d4-a716-446655440000",
  "title": "1870 Census, Licking County, Ohio — Page 42",
  "creator": "National Archives, Washington, D.C.",
  "subject": "census, genealogy, 1870, Ohio, Licking County",
  "description": "Pages 42–43 of the 1870 Federal Census enumeration for Newark, Lick
  "dateCreated": "2025-01-15T10:00:00Z",
  "source": "National Archives microfilm T9, Roll 1042, Frame 142",
  "format": "TIFF",
  "language": "en",
  "coverage": "United States, Ohio, Licking County, Newark",
  "tags": ["census", "1870", "ohio", "licking-county", "newark"],
  "rights": {
    "statement": "Public domain — U.S. Federal Government record",
    "holder": "National Archives and Records Administration",
    "license": "CC0-1.0",
    "accessRestrictions": "None"
  },
  "technical": {
    "scanner": "Epson Expression 12000XL",
    "dpi": 600,
    "bitDepth": 24,
    "colorSpace": "sRGB",
    "pageCount": 2
  },
  "administrative": {
    "repository": "InnoVadens Digital Archive",
    "accessionNumber": "2025-001-042",
    "catalogNumber": "CENSUS-OH-1870-042",
    "digitizedBy": "John Smith",
    "digitizedOn": "2025-01-15T10:00:00Z"
  },
  "preservation": {
    "masterCount": 2,
    "derivativeCount": 1
  }
}
```

23.4 Region Annotations (`regions/master-001.regions.json`)


```
{
  "mediaId": "master-001",
  "coordinateSystem": "pixel",
  "width": 4800,
  "height": 6400,
  "regions": [
    {
      "id": "region-001",
      "type": "boundingBox",
      "label": "John Smith — Head of household",
      "bounds": {
        "x": 100.0,
        "y": 200.0,
        "width": 4600.0,
        "height": 80.0
      },
      "linkedEntities": {
        "genealogy:person": {
          "givenName": "John",
          "surname": "Smith",
          "birthDate": "1828",
          "age": "42",
          "occupation": "Farmer",
          "birthplace": "Ohio",
          "relationshipToHead": "Self"
        },
        "genealogy:transcription": {
          "text": "Smith, John    42    M    W    Farmer    2000    Ohio",
          "confidence": 0.95,
          "transcriber": "GeneaScan AI v2.0",
          "transcribedOn": "2025-01-15T11:00:00Z",
          "originalLanguage": "en"
        }
      }
    },
    {
      "id": "region-002",
      "type": "boundingBox",
      "label": "Mary Smith — Wife",
      "bounds": {
        "x": 100.0,
```

```
        "y": 280.0,
        "width": 4600.0,
        "height": 80.0
      }
    }
  ]
}
```

23.5 Edit Pipeline (`edits/master-001.edits.json`)

```
{
  "mediaId": "master-001",
  "pipelineVersion": "1.0",
  "coordinateSpace": "pixel",
  "referenceWidth": 4800,
  "referenceHeight": 6400,
  "operations": [
    {
      "id": "op-001",
      "type": "deskew",
      "parameters": {
        "angle": -0.3
      },
      "appliesToDerivativeIds": ["preview-001"]
    },
    {
      "id": "op-002",
      "type": "levels",
      "parameters": {
        "blackPoint": 10,
        "whitePoint": 245,
        "gamma": 1.1
      },
      "appliesToDerivativeIds": ["preview-001"]
    }
  ]
}
```

23.6 Provenance Log (`provenance/log.json`)

```
{
  "events": [
    {
      "id": "evt-001",
      "type": "scan",
      "timestamp": "2025-01-15T10:00:00Z",
      "actor": "John Smith",
      "software": "GeneaScan v1.0",
      "details": {
        "scanner": "Epson Expression 12000XL",
        "resolution": "600 dpi",
        "format": "TIFF",
        "colorSpace": "sRGB"
      }
    },
    {
      "id": "evt-002",
      "type": "derivativeCreated",
      "timestamp": "2025-01-15T10:01:00Z",
      "actor": "GeneaScan v1.0",
      "software": "GeneaScan v1.0",
      "details": {
        "derivativeId": "preview-001",
        "format": "JPEG",
        "quality": 85
      }
    },
    {
      "id": "evt-003",
      "type": "export",
      "timestamp": "2025-01-15T10:05:00Z",
      "actor": "John Smith",
      "software": "GeneaScan v1.0",
      "details": {
        "outputPath": "census-1870-licking-oh-042.adac"
      }
    }
  ]
}
```

23.7 Checksum Manifest (`provenance/checksums.json`)

```
{
  "algorithm": "sha256",
  "files": [
    {
      "path": "manifest.json",
      "checksum": "a1b2c3d4e5f67890abcdef1234567890abcdef1234567890abcdef1234567890"
    },
    {
      "path": "master/master_0001.tif",
      "checksum": "1111111111111111111111111111111111111111111111111111111111111111"
    },
    {
      "path": "master/master_0002.tif",
      "checksum": "2222222222222222222222222222222222222222222222222222222222222222"
    },
    {
      "path": "metadata/core.json",
      "checksum": "3333333333333333333333333333333333333333333333333333333333333333"
    },
    {
      "path": "metadata/profiles/genealogy.json",
      "checksum": "4444444444444444444444444444444444444444444444444444444444444444"
    },
    {
      "path": "metadata/xmp/master_0001.xmp",
      "checksum": "5555555555555555555555555555555555555555555555555555555555555555"
    },
    {
      "path": "metadata/xmp/master_0002.xmp",
      "checksum": "6666666666666666666666666666666666666666666666666666666666666666"
    },
    {
      "path": "derivatives/deriv_0001.jpg",
      "checksum": "7777777777777777777777777777777777777777777777777777777777777777"
    },
    {
      "path": "regions/master-001.regions.json",
      "checksum": "8888888888888888888888888888888888888888888888888888888888888888"
    },
    {
      "path": "edits/master-001.edits.json",
```

```
      "checksum": "9999999999999999999999999999999999999999999999999999999999999999"
    },
    {
      "path": "provenance/log.json",
      "checksum": "aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa"
    }
  ]
}
```

24. References

24.1 Normative References

Reference	Description
ISO/IEC 21320-1:2015	Document Container File — Part 1: Core (ZIP-based containers).
RFC 2119	Key words for use in RFCs to Indicate Requirement Levels.
RFC 4122	A Universally Unique IDentifier (UUID) URN Namespace.
ISO 8601	Date and time — Representations for information interchange.
FIPS 180-4	Secure Hash Standard (SHA-256).
RFC 8259	The JavaScript Object Notation (JSON) Data Interchange Format.

24.2 Informative References

Reference	Description
Dublin Core Metadata Element Set	Vocabulary of fifteen properties for describing resources. ADAC core metadata aligns with Dublin Core where applicable.
OAIS Reference Model (ISO 14721)	Reference Model for an Open Archival Information System. Informs ADAC's preservation architecture.
BagIt (RFC 8493)	The BagIt File Packaging Format. A related but simpler packaging format for digital preservation.
PREMIS	Preservation Metadata: Implementation Strategies. Informs ADAC's provenance and fixity models.
XMP Specification	Extensible Metadata Platform. ADAC supports XMP sidecars for imaging ecosystem interoperability.
IETF BCP 47	Tags for Identifying Languages. Recommended for ADAC language fields.
SPDX License List	Software Package Data Exchange license identifiers. Recommended for ADAC rights.license fields.
ADAC-Genealogy 1.0 Format Specification	Genealogy Profile — the genealogy domain extension for ADAC.
ADAC-Legal 1.0 Format Specification	Legal Document Profile — the legal domain extension for ADAC.

25. Version History

Version	Date	Description
1.0	2026	Initial release.