Universal Manifest v0.2 Specification

Published Specification

1 April 2026

This version: https://universalmanifest.net/spec/v0.2/

Latest published version: https://universalmanifest.net/spec/v0.3/

History: Changelog

Editors: Universal Manifest Working Group

> **Coming from v0.1?** See the Migration Guide before upgrading to v0.2.

Abstract

This specification defines a JSON-LD-based file format known as the **Universal Manifest**, a portable state capsule. Formulated as a hybrid synthesis of web publication metadata and web application parameters, this format provides developers and issuers with a standardized envelope to convey linked-data identity references, role permissions, device registrations, and pointers to canonical data sources.

The Universal Manifest is specifically designed for local-first environments (e.g., venue edges, public displays) where consumers must tolerate partial connectivity and rely on cached, verifiable state. Using this standard, user agents, smart displays, and network edges can securely interoperate without requiring a continuous cloud connection, facilitating seamless cross-context experiences.

Status of This Document

This section describes the status of this document at the time of its publication.

Universal Manifest v0.2 is the current published version of the Universal Manifest specification, covering the v0.1 base format and v0.2 extensions. The v0.1 format defines the core JSON-LD envelope, lifecycle, and caching semantics. The v0.2 version introduces a normative signature profile (JCS + Ed25519), an identity binding framework with a tiered trust model, and a `claims[].claimProof` field for claim authenticity proofs, along with conventions for cross-DID binding and agent delegation.

Because Universal Manifest is still in the v0.x line, minor-version bumps may still include breaking changes when those changes are reflected by a new version folder. Implementors should review the v0.1 to v0.2 migration guide before upgrading.

Future changes are managed through the RFC mechanism and the breaking-change policy. This document uses layout formats established by major web standard groups to present normative requirements cleanly, and may be cited as the published Universal Manifest v0.2 specification.

For implementers: use the Implementation Guide, run the Standalone Conformance Suite, review Conformance v0.2, and compare against the TypeScript reference implementation.

1. Universal Manifest

A **Universal Manifest** is a JSON-LD document acting as a cross-platform data envelope. It blends the semantic linkability of a Web Publication Manifest with the applied processing lifecycle of a Web App Manifest.

1.1. Examples

The following is an example of a minimal Universal Manifest:

Example 1: Minimal Universal Manifest payload

```
{
  "@context": [
    "https://universalmanifest.net/ns/v0.1"
  ],
  "@id": "urn:uuid:123e4567-e89b-12d3-a456-426614174000",
  "@type": ["um:Manifest"],
  "manifestVersion": "0.1",
  "subject": "did:example:123",
  "issuedAt": "2026-03-01T00:00:00Z",
  "expiresAt": "2026-03-02T00:00:00Z"
}
```

1.2. JSON-LD Core Members

1.2.1. `@context` member

The `@context` member establishes the semantic definitions of terms used within the manifest. It **MUST** include the Universal Manifest namespace (e.g., `https://universalmanifest.net/ns/v0.1`).

1.2.2. `@id` member

The `@id` member provides a primary identifier. Issuers **SHOULD** generate `@id` as an opaque, offline-safe identifier (e.g., `urn:uuid:<uuidv4>`).

1.2.3. `@type` member

The `@type` member indicates the document type classifying the resource. It **MUST** include the value `um:Manifest`.

1.3. Identities & Lifespans

1.3.1. `subject` member

The `subject` member specifies the primary entity (user, app, venue) the manifest describes. It **MUST** contain a stable identifier URI (e.g., a Decentralized Identifier / DID).

1.3.2. `issuedAt` and `expiresAt` members

The `issuedAt` and `expiresAt` members formulate the bounding constraints (TTL) for the manifest's validity. Both parameters are formatted as ISO 8601 / RFC 3339 date-time strings.

1.4. Structural State Members

The manifest structure relies on domain-specific members akin to Web Publication linkages.

1.4.1. `facets` member

The `facets` member organizes extended functional blocks (`um:Facet`), packaging specific verifiable capabilities, metadata subsets, or configuration modules.

1.4.2. `claims`, `consents`, `devices`, and `pointers`

These arrays group specific operational contexts representing permissions, deployed hardware targets, and external data reference pointers connected to the `subject`.

1.5. Signature Integrity

1.5.1. `signature` member

The `signature` member encapsulates cryptographic proofs of the manifest payload. In v0.1, its format is intentionally permissive and serves as a placeholder for interoperable profiles (reserved for subsequent iterations).

1.6. v0.2 Signature Profile: JCS + Ed25519

Version 0.2 introduces the first normative signature profile. This profile constrains the `signature` member to a deterministic, portable format suitable for local-first verification on constrained devices.

Signature profiles are additive. Future versions MAY introduce additional profiles (e.g., post-quantum algorithms or W3C Data Integrity proofs). Consumers verify the profiles they support and safely ignore unknown profiles.

1.6.1. Canonicalization and Algorithm

The v0.2 profile uses **JSON Canonicalization Scheme** (JCS, RFC 8785) for canonicalization and **Ed25519** for signing. Signature values are encoded as **base64url**.

This combination provides deterministic signing input without requiring JSON-LD expansion or RDF canonicalization, matching the specification's "state capsule" usage where the JSON shape is the primary contract.

1.6.2. Signature Shape

The `signature` object **MUST** contain the following fields for this profile:

- `signature.algorithm` — **MUST** be `"Ed25519"`.
- `signature.canonicalization` — **MUST** be `"JCS-RFC8785"`.
- `signature.keyRef` — URI reference to verification key material (recommended: DID URL or HTTPS URL).
- `signature.value` — base64url-encoded Ed25519 signature over the canonical bytes.

The following fields are **OPTIONAL**:

- `signature.publicKeySpkiB64` — base64-encoded SPKI DER public key bytes for offline/fixture/local-first verification.
- `signature.created` — ISO 8601 date-time indicating when the signature was produced.
- `signature.statusRef` — URI to revocation/status material for this signature.
- `signature.revocationCursor` — monotonic status cursor/version string for cache-aware revocation checks.

Additional fields **MAY** exist for future profiles, but consumers **SHOULD** rely on `algorithm` + `canonicalization` to decide whether they can verify a given signature.

The `signature` property is **not included** in the signing input to avoid circularity. The fields `statusRef` and `revocationCursor`, when present, are metadata for revocation-aware policy checks and do not alter the signing input.

Example 2: v0.2 signature object

```
{
  "signature": {
    "algorithm": "Ed25519",
    "canonicalization": "JCS-RFC8785",
    "keyRef": "did:key:z6MkAlice#keys-1",
    "publicKeySpkiB64": "MCowBQYDK2VwAyEA...",
    "created": "2026-04-01T10:00:00Z",
    "value": "base64url-encoded-signature-bytes"
  }
}
```

1.6.3. Signing Input Procedure

To compute the signature for this profile:

1. Start with the complete manifest JSON object.
2. Remove the `signature` property entirely.
3. Canonicalize the remaining object using JCS (RFC 8785), producing a UTF-8 byte sequence.
4. Compute the Ed25519 signature over those bytes.
5. Set the `signature` property on the manifest with the fields defined above.

This yields a stable, portable verification input for any implementation that supports JCS + Ed25519.

1.6.4. Verifier Checklist

A verifier implementing this profile **MUST**:

1. Confirm the document is a v0.x Universal Manifest (required fields present and `@type` includes `um:Manifest`).
2. Enforce TTL: reject for use if `now > expiresAt`; sanity-check `issuedAt <= expiresAt`.
3. Determine signature profile support: require `signature.algorithm === "Ed25519"`, `signature.canonicalization === "JCS-RFC8785"`, and a non-empty `signature.value`.
4. Obtain a public key: if `signature.publicKeySpkiB64` exists, a consumer **MAY** use it directly (SPKI DER bytes, base64); otherwise, resolve `signature.keyRef` to public key material (method-specific, e.g., DID resolution or HTTPS fetch).
5. Recompute the signing input (remove `signature`, JCS canonicalize).
6. Verify the Ed25519 signature over the canonical bytes.

If verification fails, the manifest **MUST** be rejected for use (but **MAY** be retained for debugging).

1.6.5. Profile Identification

Consumers **MUST** treat the pair `signature.algorithm` + `signature.canonicalization` as the explicit profile identity.

- If the pair is unsupported, the manifest MUST be rejected when strict verification is required by policy.
- Consumers MUST NOT reinterpret unknown pairs as the baseline profile.

1.6.6. Revocation-Aware Verification Extension

For consumers claiming revocation-aware verification:

1. If `signature.statusRef` is present, resolve status from that URI (or a configured equivalent).
2. If `signature.revocationCursor` is present, use it to prevent stale-status acceptance and to drive cache revalidation policy.
3. If revocation status cannot be determined and policy requires active status, the manifest MUST be rejected for use.

Consumers that do not implement revocation-aware verification MUST report revocation status as `unchecked` and MUST NOT claim revocation-aware conformance.

2. Entities and Facets

Universal Manifest adopts a compositional pattern allowing nested structures (`facets` mapping to specific `entities`), drawing from semantic web standards for deeply interlinked resources.

2.1. um:Facet Module

A facet is a composable part grouped within a manifest's envelope. A facet MUST explicitly declare `@type: um:Facet`. It MAY optionally declare a `name` for display purposes, a `ref` routing to its authoritative source, and an `entity` holding the payload parameters (`um:Entity`).

2.2. um:Entity Base

The `um:Entity` acts as the base classification for all embedded configurations, representations, or localized states. It MUST be resolvable through an `@id` URI and typed accordingly (`@type` array).

3. Manifest Lifecycle and Caching

Parallel to the Web Application Manifest lifecycle, the Universal Manifest must be systematically processed, applied, and occasionally evicted from client edges.

3.1. Processing the Manifest

When a user agent or smart edge encounters a Universal Manifest, it **MUST**:

1. Parse the JSON document for semantic syntax errors.
2. Confirm the existence of required properties (`@context`, `@id`, `subject`, `issuedAt`, `expiresAt`).
3. Discard any unknown fields seamlessly to preserve forward compatibility.

3.2. Caching Formulation

For constrained devices and public displays:

1. **TTL Ejection**: Caches **MUST** immediately evict or reject payloads where the system clock surpasses `expiresAt`.
2. **Telemetry Minimization**: Centralized logging platforms **SHOULD** stream only the `@id` string (and potentially a content hash), bypassing the full JSON payload to conserve bandwidth.
3. **Identifier Rotation**: Identifiers (`@id`) **SHOULD** be rotated iteratively per-issuance to avert heuristic tracking.

4. Conformance

4.1. Consumer Behavior

A consumer platform parsing the manifest **MUST** validate structural parameters and securely ignore unknown elements without raising fatal invocation errors. Freshness (via `expiresAt` TTL constraints) is an absolute rejection gateway. Implementers **MUST** verify `issuedAt <= expiresAt`.

4.2. Issuer Behavior

Issuers generating the manifest **MUST** assign a globally stable identifier URI for the `subject` (preferably an established DID) and a random URI for the manifest root (`@id`). To shield clients from unbounded trust windows, issuers **MUST** strictly bound `expiresAt` to a sensible interaction lifetime (e.g., hours or days).

4.3. Standalone Conformance Suite

Implementations validate conformance claims natively by testing against the official `conformance/` suite—which includes fixture validation (accepting valid stubs and correctly isolating/flagging malformed artifacts like missing contexts or expired logic trees).

Implementations claiming v0.2 conformance **MUST** pass the standalone conformance suite and should publish their claimed level using the guidance at https://universalmanifest.net/conformance/v0.2/.

5. Extensibility & Profiles

Echoing the extensibility models of generic W3C recommendations, proprietary manifest members can be injected via fully qualified URIs inside the linked `@context`. Consumers ignoring unrecognized properties guarantees that domain-specific profiles won't shatter cross-system interoperability.

6. Security Considerations

Warning: Version 0.1 intentionally defers universal cryptographic standardization. Version 0.2 introduces a normative signature profile and an identity binding framework, but higher-assurance binding mechanisms (multi-signature, zero-knowledge proofs) are deferred to future versions.

6.1. Signature Limitations

Because v0.1 lacks strict prescriptive rules on canonicalization formatting and algorithm bounds, tamper-protection cannot be definitively guaranteed using baseline specifications. Implementers migrating to production applications **MUST** implement additive signature topologies documented in v0.2.

6.2. TTL Enforcement

Bounding the `expiresAt` timeline dictates the primary line of defense against presentation replay spoofing.

6.3. Resource Limits

Denial-of-Service vectors originating from disproportionately inflated arrays or recursion logic SHOULD be countered with hard limits on payload ingestion:

- Maximum memory serialization threshold: 1 MB
- Maximum recursion/nesting depth: 10 layers
- Array length maximums: 1,000 bounds

6.4. Identity Binding and Claim Authenticity

6.4.1. Bag of Claims Limitation

A Universal Manifest may contain claims from multiple issuers and references to multiple DIDs under a single `subject`. The v0.2 signature proves that the signer produced the manifest. It does **not** prove:

- That the signer controls the `subject` DID (subject-signer binding).
- That the `subject` controls all DIDs mentioned in claims or facets (cross-DID control).
- That an issuer actually issued a claim listed in the manifest (claim authenticity). The `claims[].issuer` field is a string assertion, not a verified provenance chain.

Relying parties MUST NOT treat the presence of claims in a signed manifest as proof that those claims are authentic or that multiple DIDs are controlled by the same entity.

6.4.2. Tiered Trust Model

The specification defines four trust tiers for claim verification. Each tier is strictly additive — a higher-tier manifest also satisfies all lower-tier requirements. Higher tiers provide stronger guarantees but impose more user ceremony. The specification does NOT mandate a minimum tier; relying parties choose based on their threat model and acceptable user friction.

Tier 0 — Signature-only. Zero friction. Claims are self-asserted by the manifest signer. No external `claimProof` material is present. Suitable for low-stakes use cases where the relying party has an out-of-band trust relationship with the signer. Tier 0 **MUST NOT** be used as sufficient assurance for Sybil-critical decisions.

Tier 1 — Attested or claimProof-backed. Low friction. Some or all claims carry external `claimProof` material (Verifiable Presentations) or an attested cross-DID binding claim (`identity.crossDidBinding`). Relying parties can verify specific claims against their issuers or evaluate attester trust. Consumers claiming Tier 1 assurance **MUST** enforce attester trust policy and freshness/expiry checks on the proof material used for Tier 1 acceptance. Suitable for medium-stakes use cases (social identity, reputation, basic access control).

Tier 2 — Cryptographic binding. Medium friction. Cross-DID control is cryptographically proven via multi-signature binding or zero-knowledge proofs. Each DID independently proves control by signing. Suitable for high-stakes Sybil-resistance use cases. (Future: v0.3+.)

Tier 3 — Multi-party ceremony. High friction. Multiple keyholders (potentially different people, different locations) must co-sign. Analogous to multi-sig wallets. Suitable for the highest-stakes organizational and financial contexts. (Future: v0.4+.)

Relying parties **MUST** define their required trust tier based on their threat model. Relying parties **MUST NOT** extend trust from one DID in a manifest to another DID in the same manifest unless binding proof material (Tier 1 or Tier 2) is present for that specific DID pair.

6.4.3. `claims[].claimProof` Field

The `claimProof` field is an **OPTIONAL** property on any claim object. It carries a Verifiable Presentation or attestation proof demonstrating claim issuance to the manifest subject. This field enables Tier 1 verification.

The field is named `claimProof` rather than `evidence` to avoid collision with the W3C Verifiable Credentials Data Model v2.0 `evidence` property (VCDM section 9.2), which has overlapping but distinct semantics.

`claimProof` **MAY** be an embedded object (a VP or attestation proof) or a string (URI reference to a VP or attestation endpoint).

When present as an embedded object, the consumer **SHOULD** verify the VP proof chain: (a) VC signature validates to the stated issuer, (b) VP signature validates to the holder, (c) holder DID matches the manifest subject.

When present as a URI string, the consumer MAY fetch the VP for verification when network access is available. Consumers that cannot resolve the URI SHOULD report the claim as `claimProof-unresolved` rather than `verified`.

VPs used as `claimProof` SHOULD include domain (audience binding) and challenge (nonce) parameters to prevent cross-manifest replay.

Size limits: maximum 50 KB per embedded VP; maximum 500 KB total VP payload across all claims in one manifest.

6.4.4. `identity.crossDidBinding` Claim Convention

The `identity.crossDidBinding` claim type provides a pragmatic, trust-delegated mechanism for asserting that multiple DIDs are controlled by the same entity. It works within the existing `claims[]` array and requires no schema changes. This convention is non-normative in v0.2.

Example 3: Cross-DID binding claim

```
{
  "@type": "identity.crossDidBinding",
  "boundDids": ["did:key:z6MkAlice", "did:plc:alice-bsky"],
  "attester": "did:web:verify.example",
  "attestationMethod": "AT Protocol handle resolution",
  "attestedAt": "2026-03-15T10:30:00Z",
  "expiresAt": "2026-06-15T10:30:00Z"
}
```

Required fields:

- `@type` — MUST be `"identity.crossDidBinding"`.
- `boundDids` — Array of DID strings asserted as controlled by the same entity. MUST contain at least 2 DIDs. One MUST match the manifest `subject`.
- `attester` — DID or URI of the entity attesting the binding.
- `attestationMethod` — Human-readable description of the verification method used.
- `attestedAt` — ISO 8601 timestamp of when the attestation was produced.

Optional fields:

- `claimProof` — URI or structured object pointing to the attestation proof.
- `expiresAt` — Attestation expiry. Relying parties **SHOULD** reject expired attestations.

Relying parties **MUST NOT** treat the presence of a binding claim as proof of common control unless they trust the attester. Relying parties **SHOULD** maintain a configurable attester trust list. Multiple binding claims for overlapping DID sets are independent assertions, not cumulative proof.

6.4.5. `requiredTrustTier` Declaration

A manifest **MAY** declare the minimum trust tier required for specific claims, facets, or the manifest as a whole via the `requiredTrustTier` field. This field is an integer (0–3) indicating the minimum verification tier a relying party **MUST** satisfy before acting on the associated data.

- **Manifest-level:** a top-level `requiredTrustTier` sets the floor for the entire manifest.
- **Claim-level:** a `requiredTrustTier` on an individual claim applies to that claim only.
- **Facet-level:** a `requiredTrustTier` on a facet applies to that facet only.

If a claim carries `requiredTrustTier: 2` but the relying party can only verify at Tier 1, the relying party **MUST** treat that claim as unverified. If absent, the default is 0 (no minimum required). The manifest-level value sets the floor; claim/facet-level values can only raise it, not lower it.

This convention is non-normative in v0.2; it is expected to become normative in v0.3 after implementation experience.

Example 4: Manifest with requiredTrustTier

```
{
  "requiredTrustTier": 1,
  "claims": [
    {
      "@type": "personhood.worldId.verification",
      "issuer": "did:web:worldcoin.org",
      "requiredTrustTier": 2
    }
  ]
}
```

6.4.6. Bilateral Exchange

In any transaction or interaction, both parties present a Universal Manifest to each other. Trust verification is inherently bilateral:

- Alice presents her UM to a venue; the venue verifies Alice's claims at the trust tier the venue requires.
- The venue presents its UM to Alice; Alice verifies the venue's claims at the trust tier Alice requires.
- A peer-to-peer exchange has both sides presenting and verifying simultaneously.

The effective trust tier for an interaction is the maximum of what either party demands. Asymmetric requirements are valid — each party sets its own `requiredTrustTier` independently. Two devices **MAY** exchange manifests via local transport (NFC, BLE, QR) and each independently verify the other's claims at the declared tier without a server.

When asymmetric verification outcomes occur (for example, one party cannot satisfy the other party's required tier), each party **MUST** evaluate policy independently. For Sybil-critical or otherwise high-risk actions, parties **MUST** fail closed (deny the action) when required tier checks are not satisfied. For lower-risk actions, parties **MAY** degrade to a restricted mode that excludes trust-transitive or high-impact operations.

6.5. Agent Delegation

The `um:agentDelegation` pointer type declares when a manifest subject has delegated session authority to an AI agent, bot, or proxy. This enables platforms to distinguish human-controlled sessions from delegated ones. This convention is non-normative in v0.2.

The `um:agentDelegation` pointer is placed in the manifest's `pointers` array.

6.5.1. Structure

Required fields:

- `@type` — **MUST** be `"um:agentDelegation"`.
- `delegateType` — One of `"ai-agent"`, `"bot"`, `"proxy"`, or `"human-delegate"`.
- `delegatedBy` — DID of the delegating subject. **MUST** match the manifest `subject`.
- `delegatedAt` — ISO 8601 date-time when delegation was granted.

- `expiresAt` — ISO 8601 date-time when delegation expires. Platforms MUST reject expired delegations.

Optional fields:

- `delegateId` — DID or identifier of the delegate entity.
- `scope` — Array of capability strings the delegate may exercise.
- `livenessEndpoint` — URI for real-time liveness/delegation status queries.

Example 5: Agent delegation pointer

```
{
  "@type": "um:agentDelegation",
  "delegateType": "ai-agent",
  "delegateId": "did:key:z6MkAgentBot",
  "delegatedBy": "did:key:z6MkAlice",
  "delegatedAt": "2026-04-01T10:00:00Z",
  "expiresAt": "2026-04-01T11:00:00Z",
  "scope": ["spatial.session", "social.messaging"]
}
```

6.5.2. Platform Guidance

Platforms SHOULD display delegation status to other users when present. Platforms MAY require human-only sessions for high-stakes actions (financial, governance). Platforms MAY query the `livenessEndpoint` for real-time status when available. Platforms MUST treat the delegation pointer as static for the manifest's TTL if `livenessEndpoint` is absent.

7. Privacy Considerations

- **Opaque Identifiers**: Generating the `@id` via random-distribution UUIDv4 shields session metadata from enumerability.
- **Minimal Disclosure**: Universal Manifest acts as the fundamental passport. Issuers are strongly advised against over-bundling capabilities. Only embed contextually relevant facets.
- **Subject Privacy**: Static DID correlations across disparate endpoints create persistent observation footprints. The use of pairwise / pseudonymous DIDs can significantly improve

ecosystem privacy.

Privacy-Binding Tension. Universal Manifest supports both privacy-preserving pairwise DIDs and cross-DID binding mechanisms. These goals are in fundamental tension: stronger binding enables correlation, while stronger privacy prevents binding verification. The specification does not fully address this tension at the protocol level. Instead, it provides mechanisms for both and requires relying parties to define their trust tier based on their specific threat model. Relying parties MUST NOT require cross-DID binding unless their use case demands Sybil resistance or trust transitivity. Subjects SHOULD use the minimum binding tier that satisfies their relying parties' requirements.

Data Protection. The privacy considerations in this specification identify relevant data protection provisions but do not constitute a Data Protection Impact Assessment. Deployers operating under GDPR or equivalent frameworks MUST conduct their own assessment for cross-DID binding operations, cross-border attester data flows, and mandatory binding requirements.

8. References

8.1. Normative References

The following project governance documents control how published Universal Manifest specification changes, deprecations, incidents, releases, and RFC intake are handled.

1. Breaking-Change Policy
2. Deprecation Policy
3. Incident Response and Rollback
4. Release Cadence
5. Spec Improvement Queue

8.2. Informative References

The following standards and prior art inform the identity binding, tiered trust model, and signature profile design in this specification.

1. **NIST Special Publication 800-63-3: Digital Identity Guidelines** (June 2017, updated 2024 supplement). Defines Identity Assurance Levels (IAL 1–3) and Authenticator Assurance Levels

(AAL 1–3). The UM tiered trust model adapts NIST's tiered assurance framework to a local-first, portable document context.

2. **Regulation (EU) 2024/1183: European Digital Identity Framework (eIDAS 2.0)**. Defines assurance levels (Low, Substantial, High) for electronic identification. Relevant for GDPR alignment and European adopters.
3. **OpenID for Verifiable Presentations (OID4VP)** — Draft specification, OpenID Foundation (2024). Defines VP presentation with audience binding and nonce parameters. The `claims[].claimProof` VP recommendations (domain + challenge) align with OID4VP patterns.
4. **Presentation Exchange 2.0** — Decentralized Identity Foundation, v2.0.0 (2023). Defines how relying parties express credential/claim requirements. The `requiredTrustTier` field is a simplified analog.
5. **Well Known DID Configuration** — Decentralized Identity Foundation, v0.2 (2023). Prior art for cross-DID linking via `/.well-known/did-configuration.json` Domain Linkage Credentials. The `identity.crossDidBinding` convention is analogous but manifest-embedded rather than domain-hosted.
6. **RFC 9449: OAuth 2.0 Demonstrating Proof of Possession (DPoP)** — IETF (September 2023). Defines proof-of-possession binding for OAuth2 tokens. Directly analogous to the subject-signer binding problem addressed by the tiered trust model.
7. **Verifiable Credential Status List v2021** — W3C CCG (2022); superseded by Bitstring Status List v1.0 (W3C, 2024). Privacy-preserving credential revocation and freshness checking. Target mechanism for `claims[].claimProof` credential status verification.
8. **Verifiable Credentials Data Model v2.0** — W3C Recommendation (March 2025). Core data model for VCs and VPs. The `claims[].claimProof` field is inspired by but distinct from VCDM's `evidence` property (Section 9.2) — deliberately renamed to avoid semantic collision.
9. **ISO/IEC 29115:2013 — Information technology — Security techniques — Entity authentication assurance framework** (2013). International standard defining four entity authentication assurance levels (LoA 1–4). Additional prior art for the tiered trust model.
10. **RFC 8785: JSON Canonicalization Scheme (JCS)** — IETF (June 2020). Defines the deterministic JSON serialization used as the canonicalization step in the v0.2 signature profile.