Universal Manifest v0.5 — EXT-OPT

Optional-Feature Profiles

v0.5.0. Published as tag v0.5.0 of UniversalManifest/universal-manifest, with the same wire files as release candidate 1 (v0.5.0-rc.1). It is a mechanism release, not a completed implementation or conformance qualification. Immutable versions in UniversalManifest/universal-manifest are normative; main is the working version. Earlier versions were reviewed in the OMA3 Technical Working Group, and publication asserts no OMA3 ratification.

8 November 2026

This document: /spec/v0.5/ext-opt/

Document set: Universal Manifest v0.5.0 · What changed in v0.5.0

Artifacts: JSON Schema · JSON-LD context

History: https://universalmanifest.net/spec/ (version index and changelog)

Author: Grig Bilham — inventor, architect, and author of the Universal Manifest.

Copyright © 2026 SumSet Tech LLC and the Universal Manifest contributors. This specification is available under the Apache License, Version 2.0. It is free and remains free. Earlier published revisions retain their original licence notices.

---

v0.5.0. Published as tag `v0.5.0` of `UniversalManifest/universal-manifest`, with the same wire files as release candidate 1 (`v0.5.0-rc.1`). Wire version `0.5`; namespace `https://universalmanifest.net/ns/v0.5`; reading routes `/spec/v0.5/`. It is a mechanism release, not a completed implementation or conformance qualification. Feature-specific PREVIEW and conditional requirements remain in force.

This is a **companion profile** to the Universal Manifest v0.5 Base specification. It collects the capabilities that are **not rungs on the trust-tier ladder** but are independently optional — features a conformance claim *adds* orthogonally to its tier. Read the Base first; this document assumes its terminology, envelope, and evaluation sequence.

Each part below is self-contained. The Base *names* each of these features and points here; nothing in this document changes the Tier-0/Tier-1 conformance surface. Parts marked PREVIEW are under review and revisable before the schema locks.

EXT-OPT: Optional-Feature Profiles

Companion to: Base specification (Tier-0 / Tier-1 Core). **Registry categories:** spans `domain`, `signature`, and others (per feature).

Contents

- O1. Abstract Data Model & Format Independence (incl. CBOR-LD encoding, ADM conformance, context-integrity production)
- O2. statusRef Resolution Schema (incl. the current policy-state profile PREVIEW)
- O3. Receipt as a First-Class Manifest Class PREVIEW
- O4. Bilateral Session Model PREVIEW
- O5. Federation PREVIEW
- O6. Profile Registration Mechanism PREVIEW
- O7. Agent-Delegation Scope Registry PREVIEW
- O8. Category Trust and `trustWeight` PREVIEW
- O9. Extended Structural Schemas PREVIEW (sensor consent, two-component devices, actorState)
- O10. Cryptographic Requirements Summary PREVIEW
- O11. Optional-Feature Matrix (consolidated) (incl. the conformance-claim template PREVIEW)
- O12. Asset-Bound Manifests and Represented-Person Consent Profile PREVIEW
- O13. Signed Referral-Route Attribution and Route-Scoped Disclosure Profile PREVIEW
- O14. GeoPose Primary Pose Value Profile PREVIEW
- O15. Signature Profile B: JCS + ES256 (the additive hardware-custody signature suite; normative, recognized on the wire)
- Appendix OA. Manifest Class Registry Snapshot PREVIEW

✧ ✧ ✧

O1. Abstract Data Model & Format Independence

The Base names the abstract data model and treats JSON-LD as the reference encoding; this part specifies the model in full, the production-rule contract, the CBOR-LD compact encoding, and the conformance and context-integrity production details.

O1.1. Abstract Data Model

A Universal Manifest is defined in two layers. The **abstract data model** specifies the manifest's types, properties, and semantics independently of any serialization. A **production rule** specifies how that abstract model is written into — and read back from — a concrete wire format. This separation, modeled on W3C DID Core [DID-CORE], lets the manifest gain new encodings without changing what a manifest *means*. Member definitions and examples are presented in the JSON-LD reference encoding as an editorial convenience, not a normative restriction: every requirement stated in terms of a JSON-LD member applies to the corresponding abstract property in any conformant encoding.

The abstract data model describes a manifest as a *map* of abstract properties to values, using the format-agnostic core data types of [INFRA]. The abstract properties constituting a Universal Manifest (each production rule defines how a name is expressed in its format):

- **type** (set of strings, REQUIRED) — MUST include `um:Manifest`. Expressed as `@type` in JSON-LD.
- **id** (string, REQUIRED) — globally unique opaque identifier. Expressed as `@id`.
- **contextReference** (list, REQUIRED) — reference to the term definitions fixing property meaning. Expressed as `@context` in JSON-LD; carried as a compression context reference in CBOR-LD.
- **manifestVersion** (string, REQUIRED); **subject** (string, REQUIRED); **issuedAt**, **expiresAt** (datetime, REQUIRED).
- **facets** (list, OPTIONAL); **claims**, **consents**, **pointers**, **devices** (lists, OPTIONAL); **requiredTrustTier** (integer, OPTIONAL).
- **signature** (map, REQUIRED) — integrity proof over the payload.
- **presentationProof** (map, OPTIONAL); **livenessAttestation** (map, OPTIONAL) — credential-binding members (EXT-T1).
- **actorState** (map, OPTIONAL, PREVIEW); **postQuantumSignature** (map, OPTIONAL, PREVIEW — EXT-T2).

The semantics of these properties — the evaluation sequence, the tiered trust model, consent matching, signature verification, and all conformance obligations — are defined against this abstract data model and do not depend on which production rule serialized a given manifest. Extensions and future versions that define new root-level members MUST define them as abstract properties with a representation in each production rule they are used with; unknown members are preserved as unprocessed entries. A receipt-class manifest (Base §3.3.2) additionally carries the receipt field surface of Base §3.3.1 as abstract properties under the same name-per-production mapping rule. The integrity proof is computed over the bytes of a specific production (O1.5), so a signature is bound to one encoding: converting a signed manifest between encodings requires re-signing; the *meaning* is preserved across encodings, the *signature bytes* are not.

O1.2. Production Rules

A **production rule** is a bidirectional mapping between the abstract data model and a concrete byte representation. *Production* serializes; *consumption* parses back. A production rule MUST: (1) define a media type identifying the encoding; (2) specify how each abstract property is represented, including required and optional properties; (3) guarantee that consumption of a produced representation yields an abstract manifest equal to the original (lossless round-trip) for all properties defined by this specification. This specification defines two production rules: JSON-LD (reference) and CBOR-LD (compact). Future versions or profiles MAY define additional production rules (e.g., plain-JSON or COSE-based) without altering the abstract data model. Evaluators MUST NOT assume the absence of a production rule from this list means a manifest is non-conformant; they reject representations whose media type or encoding they do not support, exactly as they reject unsupported signature profiles.

O1.3. JSON-LD Production Rule (Reference Encoding)

The JSON-LD production rule is the reference encoding and the default for interoperation. A manifest in this encoding is a JSON-LD document [JSON-LD] as described in Base §1.2–§1.6. Media type `application/um+ld+json` (also valid `application/ld+json`). The mapping is direct: abstract **type**, **id**, **contextReference** → `@type`, `@id`, `@context`; all other abstract properties → JSON members of the same name; abstract datetimes → RFC 3339 [RFC3339] strings; integers/decimals/booleans/strings/lists/maps → their JSON counterparts (JSON numbers for integers and decimals, JSON `true`/`false` for booleans). A lone `@context` string is consumed as a one-element `contextReference` list; producers SHOULD emit the array form (normalization does not affect the signature, which covers the concrete bytes of the consumed production).

O1.4. CBOR-LD Production Rule (Compact Encoding)

The CBOR-LD production rule is a second, compact binary encoding defined to demonstrate that the abstract data model is genuinely format-independent. It targets constrained channels — QR codes, NFC tags, low-bandwidth/embedded transports — where JSON-LD verbosity is costly. Media type `application/um+cbor-ld`. CBOR-LD [CBOR-LD] compresses a JSON-LD document into CBOR [RFC8949] by using the manifest's JSON-LD context to replace string term names with compact integer tokens:

1. **Production (encode).** Begin with the JSON-LD production; using the context identified by **contextReference**, map each defined term to its integer token, then emit as CBOR. The **contextReference** is carried as a context identifier (not inlined term strings).
2. **Consumption (decode).** Read the context identifier, load the corresponding context, reverse the integer-to-term mapping, and reconstruct the JSON-LD document; interpreting it under O1.3 yields the abstract manifest.

This mapping is bidirectional and semantically lossless for all abstract properties: a manifest produced in CBOR-LD and then consumed MUST yield the same abstract manifest as the JSON-LD production from which it was derived. Conformance is identical across the two encodings; only the bytes on the wire differ. CBOR-LD decoding requires the manifest's context to be available to the decoder; deployments on offline or constrained channels SHOULD pre-provision the context for the relevant version, since it cannot be assumed retrievable at decode time. Term tokenization is deterministic only when both parties use the same context version, which the versioned namespace makes explicit.

Signing (candidate per-production rule). The CBOR-LD production is signed under its own production rule, per the per-production architecture of O1.5 — a Profile-A analogue defined against the CBOR bytes:

- **Signing input.** Starting from the CBOR-LD document, remove the same signing-input exclusions as Base §1.6.3 — the **signature** property and, when present, **presentationProof** and **postQuantumSignature** — then encode the result under the Core Deterministic Encoding Requirements of RFC 8949 §4.2.1 [RFC8949]. The signing input is those deterministic-encoding bytes. This parallels Profile A exactly — parse the consumed production, remove the exclusions, canonicalize, sign or verify — with CBOR deterministic encoding playing the role JCS plays for JSON.
- **Algorithm and profile identity.** Ed25519 over that input. The signature object carries the same members as Profile A (Base §1.6.2), with the candidate `canonicalization` value `"CBOR-RFC8949-CDE"`; the `algorithm` + `canonicalization` pair (`"Ed25519"` + `"CBOR-RFC8949-CDE"`) identifies the profile per Base §1.6.5.

- **Verification.** A verifier of a consumed CBOR-LD manifest recomputes the signing input from the bytes it consumed — parse, remove the exclusions, re-encode deterministically — and verifies against them, never against another production's bytes (O1.6).

Because the signing input is the tokenized form, the signature is bound to the context version that fixes the token mapping; the pinned-context and context-hash disciplines of O1.7 therefore apply before signing or verifying. **Prohibited implementation (explicit).** Verifying a manifest's *JSON-LD-production* signature against bytes reconstructed from a consumed CBOR-LD document is prohibited: O1.6 requires verifying the integrity proof against the bytes of the production actually consumed, and O1.5 forbids a signature from traveling across a re-encoding. A manifest converted from JSON-LD to CBOR-LD MUST be re-signed under this rule before it is presented in CBOR-LD (and symmetrically, a manifest re-encoded back to JSON-LD is re-signed under Profile A). With this rule stated, a signed — hence conformant — CBOR-LD manifest is constructible by design, as O1.4's identical-conformance sentence requires; the rule remains a candidate in v0.5.

Preview: A complete CBOR-LD profile — the registered context-to-token table, finalization of the candidate signature production above, and conformance fixtures — will be finalized after maintainer review. Input is requested on whether CBOR-LD is the right compact encoding to standardize first.

O1.5. Signing and Production

The integrity proof is computed over the canonical bytes of a specific production. Signature Profile A (Base §1.6) is defined against the JSON-LD production: the signing input is the JCS-canonicalized JSON-LD document with the signing-input exclusions removed (the **signature** property and, when present, **presentationProof** and **postQuantumSignature**); the production consumed for signing is constrained to I-JSON with duplicate member names rejected (Base §1.6.3), which is what makes the JCS input well-defined. Each production rule MUST either reuse a profile defined against the abstract model or specify how the integrity proof is computed over its own canonical bytes, so a verifier can recompute the signing input deterministically. Both production rules defined by this specification have a stated integrity-proof computation: Signature Profile A for JSON-LD, and the candidate Profile-A analogue over RFC 8949 Core Deterministic Encoding bytes for CBOR-LD (O1.4). Because a signature is bound to the bytes of one production, a manifest re-encoded under a different production rule MUST be re-signed under that production's signing rule. *(Open question: whether a future profile should sign a canonicalization of the abstract model — making one signature portable across encodings — is the strategic open question of the format-independence design; this draft builds in per-production signing, and the abstract-model option stands as the recorded question for the next version.)*

O1.6. Conformance to the Abstract Data Model

The behavioral requirements of Base §4.1–§4.3 are stated against the abstract data model and apply identically regardless of encoding. A conformant implementation MUST support at least one production rule and SHOULD support the JSON-LD reference encoding; MUST consume supported representations into the abstract data model before applying behavioral requirements; MUST reject an unsupported encoding rather than misinterpreting it; MUST preserve every abstract property across a production/consumption round-trip when it both produces and consumes an encoding; and MUST verify the integrity proof against the bytes of the production it consumed. **Supporting only the JSON-LD reference encoding is sufficient for full conformance**; supporting CBOR-LD is OPTIONAL.

Carrying external standards (informative). The same wrapper discipline extends beyond UM's own encodings. A manifest can carry — as a facet or claim value — or point to (via `pointers`) an artifact governed by another standard: C2PA content-credential material [C2PA], an OGC GeoPose payload [OGC-GEOPOSE] (O14), a W3C Verifiable Credential, or a comparable external artifact. Universal Manifest complements such carrier and provenance standards; it does not replace them. The carried artifact travels unmodified — UM never re-signs, transforms, or reinterprets its bytes — and it remains independently verifiable under its own standard's rules: a receiver performs two independent checks, the UM envelope evaluation defined by this document set and the carried standard's own verification, and passing one never substitutes for the other. An evaluator that has verified the UM envelope has not thereby verified the carried artifact; absent a registered carry profile, a carried standard the evaluator cannot evaluate is preserved as sealed/opaque or unprocessed content under the existing fail-closed baselines (Base §3.3.1), never reported as verified. A dedicated external-standard-carry profile — a registered carried-standard verification-result receipt field and an unknown-carried-standard disposition value — is a registered-profile candidate (O6) whose wire shape is not defined by this draft.

O1.7. Context Integrity (production detail)

Term meanings — and, for CBOR-LD, tokenization — depend on the content of the versioned namespace context. The manifest signature covers the context *reference* (the URI string), not the context *document content*: if the served context changes or an attacker controls its resolution, term semantics and CBOR-LD round-trips can change silently under a still-valid signature. To prevent this, the versioned namespace URI identifies an **immutable** context: once published, the context document for a version MUST NOT change. Evaluators SHOULD ship or pin the context for each supported version rather than fetching it at evaluation time, and MUST NOT fetch contexts from untrusted resolvers at verification time. CBOR-LD deployments MUST use the pinned context,

since tokenization correctness depends on byte-identical context versions on both sides. *(The Base states the evaluator-facing requirement in Base §6.10; this is the production/encoding detail.)*

Hash computation. The context content hash named by Base §6.10 and reproduced in Base Appendix B is a multihash [MULTIFORMATS]; the multihash function MUST be SHA-256 (multihash function code `0x12`) for v0.5, the same function required for the receipt-chain `prevHash`. The hash input is the exact published byte sequence of the versioned context document — its UTF-8 text with LF (U+000A) line endings and a single trailing LF — with no re-serialization, JSON canonicalization, whitespace normalization, or transcoding applied before hashing. Where the hash appears in text-bearing positions it is expressed as a multibase base58btc string (`z` prefix); binary positions MAY carry the raw multihash bytes. The hash is encoding-independent and pins the context for every production rule: a JSON-LD evaluator verifies it before trusting term definitions (Base §6.10), and a CBOR-LD deployment MUST verify it over those identical bytes before deriving or applying a term-token table (O1.4).

✧ ✧ ✧

O2. statusRef Resolution Schema

This part specifies the resolution protocol for the `signature.statusRef` field (Base §1.6.6, Base §3.4). `signature.statusRef` resolves the status of the **manifest instance** identified by `manifestId`; it does **not** convey the revocation status of any key (key revocation is determined separately via the key's DID document verification-method state, or a `claimProof` entry's own `statusRef`). These distinct objects of revocation MUST NOT be conflated.

O2.1. Resolution Procedure

The entire manifest `signature` object is excluded from the envelope signing input (Base §1.6.3); **neither `signature.statusRef` nor `signature.revocationCursor` is authenticated by that signature**. This base status response has no independent signature. HTTPS authenticates the contacted transport endpoint, not the holder's authorization of that endpoint or its authority over the manifest, consent, key, or governed action. Before a response can establish relied-upon status or current permission, the evaluator MUST independently establish a trusted binding between the status authority, target, and applicable freshness rules. A response's self-declared `manifestId`, an embedded locator, or successful TLS alone is insufficient.

When `signature.statusRef` is present and revocation-aware verification is implemented, the evaluator follows its authorized resolution policy: use HTTP `GET` only for a permitted HTTPS locator, require `application/json`, validate the response schema and target, and apply status semantics only after the independent authority checks above. A non-HTTPS, unreachable, unsupported, or unvalidated endpoint yields `unchecked`; it never establishes `active`. No implicit arbitrary-URI fetch authority is created by carrying the locator. The optional-pointer rule remains: an authoritative source can establish current local permission without a public status request, while a stale replica cannot.

Cursor semantics. A cursor is comparable only under the ordering defined by the selected status profile for its authority and target. Evaluators MUST NOT order opaque tokens lexicographically or assume numerically shaped tokens from different authorities share a sequence. A profile using a minimum cursor MUST define syntax, comparison, authority/target binding, and rollover. Without those definitions and trusted binding, a claimed `revocationCursor` minimum is unsatisfied, not ignored; a different opaque token alone does not mean newer. A newer trusted local observation MUST NOT be rolled back by older evidence under the same defined ordering.

HTTP cache semantics. An ETag is a transport cache validator with HTTP's own syntax, not an authenticated status assertion or an ordering rule. Conditional requests SHOULD send a stored HTTP ETag in `If-None-Match`, rather than copying an arbitrary status cursor. A profile mapping cursor to ETag MUST define the encoding and comparison. `304 Not Modified` can revalidate a cached representation under HTTP rules, but MUST NOT grant new current permission, establish authority, or extend relied-upon status evidence beyond its independently established validity bounds. A `nextCheck` hint does not extend those bounds either.

This clarification supplies integrity limits and fail-closed behavior, not the protected signed-status/current-state mechanism. Its carrier, authority proof, signature suite, and high-privacy resolution mechanism remain unresolved; no new wire member or implicit signature is introduced here.

O2.2. Response Schema

A status endpoint MUST return a JSON object with `manifestId` (MUST match the manifest's `@id`), `status` (one of `"active"`, `"revoked"`, `"suspended"`; MUST be present), and `updatedAt` (RFC 3339; MUST be present). It MAY include `reason` (human-readable), `cursor` (opaque status metadata; ordering only under O2.1), and `nextCheck` (an ISO 8601 duration, e.g. `"PT1H"`, recommending revalidation). `updatedAt` and `nextCheck` are endpoint assertions; consumers MUST NOT use them to extend current permission without the trusted authority/target and validity checks of O2.1.

O2.3. Status Semantics

- `"active"` — the independently authorized status source reports that this manifest instance is not revoked or suspended. The evaluator continues the remaining evaluation checks; this value does not itself prove a signature, credential, key authority, consent, or current permission.
- `"revoked"` — permanently invalidated by the holder; the evaluator MUST reject with outcome `"rejected"` and `revocationStatus: "revoked"`.
- `"suspended"` — temporarily suspended; the evaluator SHOULD treat as `"accepted-with-warnings"` and record `revocationStatus: "suspended"`; MAY apply local policy on whether to process.

O2.4. Error Handling

When the status endpoint is unreachable or errors, evaluators MUST: on **404**, record `revocationStatus: "unchecked"` with reason `um:reason:status:endpoint-unknown-manifest`; on **503 or network failure**, record `"unchecked"` with reason `um:reason:status:endpoint-unavailable`; on **other 4xx/5xx**, record `"unchecked"` with the HTTP status code in the reason. Evaluators in offline mode MUST record `"unchecked"` with reason `um:reason:status:offline`. Deployments MAY use a W3C Bitstring Status List endpoint [VC-STATUS] as the `statusRef` target, provided the evaluator can parse the response into the status semantics above; this is a SHOULD-level alternative that mitigates the status-resolution correlation risk (O5 and the status-resolution correlation consideration in Base §7).

Endpoint failure, unsupported status processing, or failed authority/target validation is recorded as `unchecked`; it does not establish `active` status. Manifest acceptance follows Base §1.6.6. If revocation status cannot be established and policy requires active status, the manifest MUST be rejected for use. When policy does not require active status, evaluators MUST NOT reject the manifest solely because the endpoint is unreachable. In either case an operation requiring current permission MUST be denied until an authorized current-state path establishes it. This does not authorize processing a facet whose required consent cannot be verified (Base §3.1.4).

O2.5. Status Endpoint Conformance

A **conformant status endpoint** (Base §4.6) MUST respond with the response schema and status semantics above (including the `status` values and the `cursor`/`nextCheck` fields), MUST key its response to the `manifestId` queried, and SHOULD support conditional requests. Resolver

conformance (the federated-status class of O5) is deferred pending the open decision on whether federation moves to a companion specification.

(Worked example: statusRef response — see Cookbook.)

O2.6. Current Policy-State Profile PREVIEW

Candidate registered profile (candidate identifier `um:profile:trust:current-policy-state`, registration via O6; the identifier — including its `trust` category segment — is provisional, since category assignment is a registration-time decision). **Design-stage:** the members and value tokens below are candidates whose wire shape may change; they are **not yet** in the published `schema.json`/`schema.jsonld` or the conformance fixture suite, and per the Base conformance discipline (Base §4.4) they carry no conformance tests until the design is finalized.

The bounded composition in Base §3.4.1 describes checks relevant to "is this consent/authorization state still current?", conditional on independently established authority and target binding. This design-stage profile proposes three pieces that composition leaves open. It does not supply the unresolved protected signed-status/current-state mechanism, adopt an authenticated carrier, or turn transport integrity into current permission. It applies to any governed asset or action — 3D assets, scans, media, digital replicas, embodied-person representations, and agentic actions alike.

Supersession pointer. A status response (O2.2) MAY additionally carry `supersededBy` — a URI identifying a proposed successor record. It is meaningful only alongside `status: "revoked"` or `"suspended"`; the closed `status` enum is unchanged. An evaluator implementing this design-stage profile may record the pointer or request the successor from the holder, but MUST NOT treat the superseded record's grants as current. `supersededBy` carries no independent signature in this base response, and `signature.statusRef` is likewise not authenticated by the envelope signature. Both require independently established authority/target binding before reliance (O2.1). A successor confers nothing until it is evaluated under the ordinary sequence. Evaluators MUST NOT automatically dereference `supersededBy` at evaluation time: it can expose a request-forgery and correlation path to an endpoint-influenced target. Retrieval remains through the holder or the deployment's authorized presentation channel; a locator is not authority to trust or fetch its target.

Policy-window taxonomy. A deployment, profile, or consent condition MAY declare a `policyWindow` (ISO 8601 duration) for a governed action: the maximum age of current-state evidence (a status response, a consent record's validation, a delegation liveness check) the evaluator may act on. Within the window, cached evidence MAY be used, subject to the `nextCheck` and `revocationCursor` rules of O2.1. Beyond the window, or when state cannot be resolved at all,

the evaluator MUST NOT proceed on stale state and MUST apply the declared **unresolved-state disposition**: `deny` (refuse the governed action; the default when no disposition is declared — fail closed) or `escalate` (withhold automatic processing and route the decision to a designated out-of-band authority; escalation is a refusal-to-proceed-automatically, never a grant). Where the declared disposition is `escalate` but no escalation authority is designated or reachable, the evaluator MUST apply `deny`: an unconfigured escalation route degrades to the fail-closed default, never to automatic processing. The disposition governs the *governed action*, not manifest acceptance. Denying or escalating the action does not itself determine whether the manifest is accepted: evaluation also follows Base §1.6.6 and O2.4. When policy requires active status that cannot be established, the manifest MUST be rejected for use; otherwise, endpoint unreachability alone MUST NOT reject the manifest. Receipt production records the action's disposition and the applicable manifest result. The disposition vocabulary is a registry, extensible via O6.

Policy-window carrier (candidate). A consent-carried policy window is expressed as consent `conditions` tokens (Base §1.4.4), in the same plain-string condition shape as the `route-class:<value>` convention (O13.4; O6 registry-token note): **`policy-window:<duration>`** (e.g. `policy-window:PT1H`, the ISO 8601 duration) declares the window for governed actions under that consent entry, and **`policy-disposition:deny`** / **`policy-disposition:escalate`** declares the unresolved-state disposition (absent a disposition token, `deny` — the fail-closed default above). At most one `policy-window:` token and one `policy-disposition:` token may appear on one entry; an entry carrying more than one of either is unenforceable and fails closed under the Base's condition rule. A `policy-disposition:` token without a `policy-window:` token is well-formed: it declares the entry's unresolved-state disposition — applied whenever required current-state evidence cannot be resolved at all — while adding no age bound. A deployment- or profile-declared window is configuration, declared out-of-band (for example in the deployment's conformance claim, O11.1, or the profile's registration entry) and evaluated identically. Interoperation degrades safely: to an evaluator that does not implement this profile the tokens are unrecognized conditions, so the governed facet already fails closed under Base §1.4.4 — a non-implementer can never process *more* than an implementer. **Default when no policy window is declared** by any carrier: no additional currency bound applies — the existing machinery alone governs (TTL, `nextCheck`/`revocationCursor`, consent validity windows, per Base §3.4.1); this profile adds no implicit window.

Reason codes and receipt evidence. This profile registers the reason-code candidates `um:reason:policy:superseded`, `um:reason:policy:unresolved`, `um:reason:policy:window-expired`, and `um:reason:policy:escalated` (O3.3), and the receipt event-class candidates `policy-state-resolved`, `policy-state-stale`, and `policy-state-escalated` (O3.2), so the disposition of a governed action — resolved, unresolved, stale, superseded, denied, or escalated — is machine-readable in the receipt (Base §3.3).

Honest limits. The profile proposes auditable, fail-closed dispositions; it does not make a status endpoint truthful or authoritative. HTTPS alone supplies transport integrity, while reliance still requires the independent authority/target and validity checks of O2.1. It does not erase or relabel state already acted on under a then-valid record (consistent with Base §7.2), and does not by itself provide resolver-operator privacy (see Base §7). A `supersededBy` pointer links an old instance to its successor, weakening instance unlinkability for that pair. Successor identifiers can be sensitive response content and SHOULD be restricted to authorized queriers where appropriate. No protected current-state proof or privacy result is established by this candidate prose.



O3. Receipt as a First-Class Manifest Class PREVIEW

A receipt is itself a **first-class manifest class** (Base §1.0.1, Base §3.3.2): a signed, portable record that can be chained, retained, and independently verified, carrying the common envelope members with `@type` including both `um:Manifest` and `um:Receipt`, plus the receipt fields of Base §3.3.1. The envelope's `subject` on a receipt manifest is the **evaluator** that produced it: the receipt is a record of the evaluator's own processing act, and the manifest it processed is identified by the `manifestId` receipt field, not by `subject` (this is the practice the fixture suite and Cookbook already follow). This promotion is additive; an evaluator that only emits inline receipts remains conformant, and the chain-integrity members are OPTIONAL unless a receipt is part of a sequence.

O3.1. Chain Integrity

Receipts in a session or audit sequence form a hash-linked chain. A chained receipt MUST include `chainId` (a URI identifying the chain; for a bilateral session it MUST equal the `sessionId` of O4; it is the stable identifier tying a sequence together — `exchangeId` is per exchange round and does not serve this purpose), `seq` (a monotonically increasing non-negative integer; the first receipt has `seq` 0), and `prevHash` (the hash of the immediately preceding receipt's canonical signing bytes, encoded as a multibase/multihash string [MULTIFORMATS]; the multihash function MUST be SHA-256 for v0.5; omitted for `seq` 0).

A chained receipt SHOULD be signed with a **session-scoped signing key** (for example, `deviceCapability.sessionSigningKey` of O9.2, or another ephemeral key bound to the session) rather than a long-lived identity key, so a receipt chain does not become a long-lived correlator. Because a session-scoped key cannot be authorized through the subject's DID document the way

EXT-T1 §T1.5 authorizes manifest-signing keys, a session-scoped receipt-signing key MUST be introduced by a **`sessionKeyAuthorization`** — a candidate member of the chain's first (`seq` 0) receipt manifest, carrying the authorization object inline or a URI resolving to it. The authorization object's candidate shape: `sessionKey` (REQUIRED — the session public key, as a JWK or a multibase-encoded public key; it MUST be the key that signs the chained receipts, e.g. the `deviceCapability.sessionSigningKey`); `chainId` (REQUIRED — MUST equal the chain's `chainId`); `validFrom` and `validUntil` (REQUIRED — RFC 3339; the window within which this key may sign receipts of the chain); and `signature` (REQUIRED — a Signature Profile A signature over the authorization object, computed by applying the Base §1.6.3 procedure to the object with its own `signature` member excluded, produced by a key authorized for the **evaluator's** DID — a key appearing under `authentication` or `assertionMethod` in the DID document of the `evaluatorId` party). A verifier attributing a session-signed chain MUST verify the authorization — the evaluator-key signature, the `chainId` match, that the chained receipts' signing key equals `sessionKey`, and that each receipt was signed within the validity window — before attributing the chain to the evaluator; on failure, chain *integrity* may still verify link by link, but the chain is unattributable and MUST NOT be attributed to the evaluator. Evaluators verifying a chain MUST confirm each `seq` increments by one without gaps and each `prevHash` matches the prior receipt; a broken link MUST be reported and the chain after the break treated as unverified.

Verifying an archived chain vs. acting on a receipt. A chained receipt is a durable audit record, and its links age past their manifest TTLs. The freshness gate (Base §3.1.2) and cache eviction (Base §3.2) govern *acting on* a receipt's content as current state; they do not govern verifying an archived chain. Chain verification — signature validity, `seq` continuity, `prevHash` linkage, and the `sessionKeyAuthorization` — proceeds over expired links unchanged: an auditor records an expired link as *expired* (its content is stale for action) but MUST NOT treat expiry as a broken link under the broken-link rule above. Deployments retaining chains for audit SHOULD bound retention per Receipt Minimization (Base §7).

O3.2. Typed Event Vocabulary

A receipt manifest MAY carry an `events` array recording typed lifecycle events. Each event object MUST carry an `eventType` from the registry below and an `at` (RFC 3339) timestamp; it MAY carry an event-specific `reason` (O3.3) and a `subjectRef` identifying the affected facet, claim, consent, or device. Recognized event classes: `manifest-arrived`, `manifest-verified`, `manifest-rejected` (envelope lifecycle); `facet-processed`, `facet-sealed`, `facet-consent-denied` (facet outcomes); `consent-granted`, `consent-withdrawn`, `consent-expired` (consent transitions); `claim-verified`, `claim-failed`, `binding-verified`, `binding-failed` (trust outcomes); `avatar-retrieved`, `avatar-substituted` (presence/avatar disclosure outcomes; presence is a

locator, not authorization); `facet-key-rotated`, `facet-key-shredded` (facet-encryption key-lifecycle events — Base §2.3.5); `session-unlocked` (the opening of an unlock window — O4.5); `session-unlock-refused` (an unlock or locked-facet access withheld because the asserted assurance class was below the facet's `requiredAssuranceClass` floor — EXT-T1 §T1.3.3; carries reason `um:reason:trust:assurance-insufficient`); `session-completed` (session terminal event). This is a registry, not a closed enumeration: new classes are added through the profile registration mechanism (O6). Evaluators encountering an unrecognized `eventType` MUST preserve the event but MUST NOT act on it.

Key-lifecycle and unlock events MAY carry an OPTIONAL `facetKeyRef` field identifying the affected facet-encryption key by its `kid` as an **identifier, never key material**. `facetKeyRef` is distinct from `signature.keyRef` (the signature-verification key URI) and from the receipt `keyRefResolution` field — distinct referents, distinct names. A `facet-key-shredded` event records the emitter's **assertion and timing** of key destruction, **not** cryptographic proof that copies are gone; consumers MUST NOT treat it as retroactive revocation (paralleling the Sealed-Entry Forward Secrecy discipline, Base §7.2). Key-rotation/shred histories are bounded by the `events` array limits of the Base resource limits (Base §6.3: maximum array length 1,000 entries / maximum manifest size 1 MB), so an evaluator is not forced into unbounded replay. Key-lifecycle events are subject to Receipt Minimization (Base §7: carry only `facetKeyRef` as an identifier, minimum detail) and SHOULD ride session-scoped-signed receipt chains (O3.1), so the shred/rotation stream is not a long-lived cross-interaction correlator.

O3.3. Structured Reason Registry

Receipt `reason` values (on facet statuses, consent statuses, and events) SHOULD be drawn from a structured reason registry using `um:reason:<category>:<code>` naming (for example `um:reason:consent:withdrawn`, `um:reason:crypto:no-decryption-key`, `um:reason:crypto:key-rotated`, `um:reason:crypto:key-shredded`, `um:reason:trust:tier-unsupported`, `um:reason:trust:assurance-insufficient`). Structured reasons make receipts machine-comparable. Free-text reasons remain valid where no registry code applies. New reason codes are registered through the profile registration mechanism (O6).

O3.4. Transparency-Log Anchoring (optional)

A receipt manifest MAY be anchored to an append-only transparency log following the Certificate Transparency 2.0 model [RFC9162]. When anchored, the receipt MAY carry a `transparencyAnchor` object with `logId`, `inclusionProof` (a Merkle inclusion proof; its structure is a wire-freeze decision to be bound to the [RFC9162] inclusion-proof structure — log identity, tree

size, leaf index, audit path — and until then it is carried as an opaque candidate object), and `anchoredAt` (RFC 3339). Anchoring is OPTIONAL and does not increase per-event cost for deployments that do not use it; it provides tamper-evident, independently auditable history. Evaluators that do not implement transparency anchoring MUST preserve the `transparencyAnchor` field without acting on it.

(Worked example: chained receipt manifest with typed events — see Cookbook.)

Preview: Promoting the Receipt to a first-class class — with the `seq`/`prevHash` chain, typed event vocabulary, structured reason registry, and optional CT-style anchoring — is built on the editors' default (promote in v0.5; additive). Input is requested on the canonical event-class and reason-code registries and on the hash/multibase encoding for `prevHash`.



O4. Bilateral Session Model PREVIEW

This part specifies the protocol-layer session model for bilateral exchanges, extending the manifest-level bilateral exchange of Base §6.4.6 with session objects, paired receipt correlation, and exchange identifiers.

O4.1. Session Object

A bilateral session is a time-bounded interaction between two participants who each present and evaluate the other's manifest. A session object MUST contain `@type` (`"um:BilateralSession"`), `sessionId` (a globally unique URI; MUST be present), `exchangeId` (a correlation identifier shared by both parties in a single exchange round; both include the same `exchangeId` in their receipts; MUST be present), `participants` (an array of exactly two participant objects, each with a `did` and `role` ∈ {`"initiator"`, `"responder"`}), `initiatedAt` (RFC 3339), `expiresAt` (RFC 3339; evaluators MUST reject session operations after this time), and `state`. A session comprises exactly one exchange round; a new exchange between the same parties is a new session with new `sessionId`/`exchangeId`. `sessionId` and `exchangeId` MUST each be generated with at least 128 bits of entropy and MUST NOT encode party identifiers, since both are correlation tokens the parties rely on. **Session operations** — the acts governed by the session lifecycle and its TTL — are: presenting or accepting a manifest under the session, producing or accepting a receipt bearing the session's `exchangeId`, and the completion acknowledgment (O4.2). After `expiresAt`,

the session is `"expired"` and a participant MUST NOT perform or accept any of these under that `sessionId`/`exchangeId`; receipts already exchanged remain valid records.

O4.2. Session Lifecycle

States, strictly forward (no return to a previous state): `"initiated"` (initiator created the session and sent `sessionId`/`exchangeId`); `"manifests-exchanged"` (both presented manifests; each SHOULD reference `exchangeId` in its presentation context); `"receipts-exchanged"` (both produced and shared receipts; each receipt MUST include `exchangeId`); `"completed"` (both acknowledged the other's receipt); `"expired"` (TTL elapsed without `"completed"`; evaluators MUST treat as terminated). `"expired"` MAY be entered from any active state. The `"completed"`-state acknowledgment mechanic — how a party signals that it received the counterparty's receipt — is transport-defined (O4.4): a transport binding MAY use a message type, an HTTP response, or an equivalent; the session model requires only that both acknowledgments occur before the session is `"completed"`.

O4.3. Paired Receipt Correlation

A receipt produced as part of a bilateral session MUST include an `exchangeId` matching the session's, enabling pairing: Party A's receipt (evaluating B's manifest) correlates with Party B's receipt (evaluating A's manifest) via the shared identifier. A receipt exchanged in a bilateral session MUST include `evaluatorId` and SHOULD carry `receiptSignature`, so the exchanged receipts are attributable and tamper-evident.

O4.4. Transport Independence

The session model is transport-agnostic. Sessions MAY be conducted over any transport that can carry manifest payloads in a supported production rule — JSON-LD or (once finalized) CBOR-LD on constrained channels: local transports (NFC, BLE, QR scan), network transports (HTTPS, WebSocket), or hybrids. Transport-specific bindings MAY be defined in profile documents. The session object MUST NOT assume any specific transport capability.

Authentication and signature verification establish only their verified counterpart and content bindings; a relay can forward signed traffic without altering it. Channel binding without a distance-bounding result MUST NOT be represented as proof of physical proximity. Missing channel binding is reported according to the selected method, without inventing a distance claim.

O4.5. Unlock Window PREVIEW

An **unlock window** is a time-bounded grant that opens a holder-enumerated set of facets for a single subject. It is **not** a `um:BilateralSession` (O4.1): it has no `participants`, no `exchangeId`, and is a single-subject construct. An unlock window is a recognized composition of an existing consent carrying `unlock.window` in its `scope` (dot form, per the agent-delegation scope convention of O7), with `grantedAt`, `expiresAt`, an optional `withdrawnAt`, and the enumerated facet set (`unlockWindowFacets`, defined below). It is not a new manifest object.

The window's consent members. The unlock-window consent is an ordinary consent entry (Base §1.4.4) whose `scope` contains the `unlock.window` token, extended by one member this profile defines: **`unlockWindowFacets`** — an array of facet `@id` strings explicitly enumerating the facet set the window opens. `unlockWindowFacets` is REQUIRED on a consent whose `scope` contains `unlock.window`; it MUST be explicit and bounded (literal `@id` values only — no wildcard or pattern forms), and, like the OPTIONAL unlock-policy members below, it is carried on the consent object inside the signed payload. A consent carrying `unlock.window` without `unlockWindowFacets` is malformed for this profile, and the window MUST NOT be opened (fail closed). An entry of `unlockWindowFacets` that matches no facet of the presented manifest is inert — a window cannot open what is not projected. The consent's **`facetRef`** (REQUIRED by the Base consent shape) MUST name one facet of the enumerated set: it designates an anchor member, and the Base's scope/purpose/validity matching applies to the entry as usual — but the set the window opens is governed by `unlockWindowFacets`, never by `facetRef` alone. *(Wire status: `unlockWindowFacets` and the unlock-policy members are grade-1 structural PREVIEW — present in the published v0.5 context and, for the enumeration, exercised by a shape fixture — while the enforcement semantics of this part remain revisable, per the two-grade PREVIEW policy.)*

An unlock window MUST be sound on the following axes:

1. **Bounded / enumerated facet set.** The window MUST explicitly enumerate the facet set it opens (by `@id`, in `unlockWindowFacets` — defined above). It MUST NOT open an unbounded or implicit set. This prevents the "one ceremony unlocks everything" anti-pattern that per-facet isolation (Base §2.3.5) exists to prevent.
2. **Never lowers a per-facet floor or isolation.** The window MUST NOT lower or bypass any per-facet `requiredLiveness` floor (EXT-T1 §T1.3.1) or any isolation obligation (Base §2.3.5). A facet whose floor is unmet stays withheld even inside the window.
3. **Re-attestation on freshness decay.** When the opening liveness attestation's `freshnessClass` degrades below a facet's floor during the window, the window MUST require re-attestation for that facet; a single attestation MUST NOT grant a window outliving its freshness class.

4. **Anti-replay binding.** The window MUST bind to a fresh `presentationProof` / nonce (EXT-T1 §T1.2), with audience binding, so static signed bytes cannot be replayed to re-open a window. This is the same anti-replay primitive a write inside the window reuses; a write is additionally per-write replay-guarded (Base §3.1.4).
5. **Fail closed on unobservable lifetime conditions.** Idle-timeout and lock-on-sleep are runtime-only conditions the receipt cannot prove; only `grantedAt`/`expiresAt`/`withdrawnAt` are auditable bounds. Where a lifetime condition is unobservable, the evaluator MUST fail closed. The window bounds the *authorization window*, not in-use plaintext (see the in-use-plaintext note, Base §7.2).
6. **Receipt-side liveness reference.** The opening ceremony is recorded as a `session-unlocked` event in the typed event vocabulary (O3.2), referencing the opening liveness attestation — the *liveness reference* adds no consent member. The window itself is carried by the `unlock.window` scope token plus the `unlockWindowFacets` enumeration and the OPTIONAL unlock-policy members (the consent-object members this profile defines); the receipt-side record adds nothing further to the consent object.

What opening means — the window never substitutes for consent (fail closed). Opening an unlock window is an *unlock ceremony plus a time-boxed authorization window*; it is not a consent grant. Membership in `unlockWindowFacets` does not authorize any operation on a facet: each enumerated facet is processed only under a consent entry of its own that governs it — Stage 4 (Base §3.1.4) applies unchanged, so an enumerated facet with no governing consent entry stays `"consent-missing"` and MUST NOT be processed even while the window is open. What the window adds, for the enumerated set and only within `grantedAt`–`expiresAt` (as tightened by the unlock-policy bounds below): (1) the **opening ceremony** — the fresh `presentationProof`/liveness binding of axes 3–4 — gating access to locked-tier facets (EXT-T1 §T1.3.3); (2) the **time box** — outside an open window, a locked-tier facet of the set reverts to locked/withheld even where its own consent entry remains valid; and (3) the **auditable record** — the `session-unlocked` event of axis 6. A write inside the window additionally requires its own write-consent and per-write replay guard (Base §3.1.4).

Unlock-policy fields (locked-tier auditability). When an unlock window opens a locked-tier facet (EXT-T1 §T1.3.3), the unlock-window consent MAY additionally carry these OPTIONAL unlock-policy members so the time-boxed grant is auditable and tightly bounded:

- `idleLifetime` — an ISO 8601 duration (e.g. `"PT10M"`) after which an idle window SHOULD re-require the unlock gesture;
- `absoluteLifetime` — an ISO 8601 duration (e.g. `"PT12H"`) capping the window regardless of activity; it MUST NOT extend the consent's own `expiresAt` (the consent `expiresAt` remains the hard auditable bound);

- `lockOnSleep` — a boolean; when `true`, the runtime SHOULD close the window when the device sleeps/locks.

These fields **state policy intent and defaults+floor, not a single mandated number**: a conformant deployment SHOULD default to a short idle lifetime and a bounded absolute cap (representative practice: a ~10-minute idle timeout with a ~12-hour absolute cap; a software-class unlock that derives a key from a passphrase SHOULD use a memory-hard KDF — e.g. Argon2id with at least 19 MiB of memory — for the `software` rung). The exact minutes and KDF parameters are tunable policy.

This is **additive clarification, not a contradiction of axis 5**: `idleLifetime`, `absoluteLifetime`, and `lockOnSleep` are runtime-only conditions whose enforcement the receipt cannot prove. Axis 5 stands — only `grantedAt`/`expiresAt`/`withdrawnAt` are auditable bounds, and **where a lifetime condition is unobservable the evaluator MUST fail closed.** Declaring these fields commits the runtime to the policy; it does not make the runtime condition retroactively auditable.

(Worked example: bilateral session object — see Cookbook.)

Open question: Whether the session model remains in the core specification or is published as a separate companion specification is open; bilateral sessions are protocol-layer concerns that may evolve independently of the manifest format.

✧ ✧ ✧

O5. Federation PREVIEW

This part specifies the federation model: how multiple resolver operators coordinate to provide status checks, cache invalidation, and availability across a distributed network.

Resolver coordination. A federated resolver network consists of two or more resolver operators that synchronize manifest status. Resolvers MUST identify themselves using DIDs and MUST establish mutual trust through bilateral attestation: each operator issues an attested claim about the *peer's* resolver identity — an ordinary attester-asserted claim whose `subject` is the peer's DID (a candidate `identity.peerAttestation` claim type, registrable via O6), or an equivalent mutual-attestation pattern. `identity.crossDidBinding` is **not** that pattern and MUST NOT be used for peer trust: it asserts that multiple DIDs are controlled by the *same* entity (Base §6.4.4), which is precisely not the relationship between two independent resolver operators. Resolver discovery is out of scope; deployments MAY use a well-known URI (`/.well-known/um-resolvers`), a DID service endpoint, or out-of-band configuration.

Status check distribution. When an evaluator resolves a `statusRef` URI, it MAY query multiple authorized federated resolvers. Only responses meeting the independent authority/target and validity requirements of O2.1 participate in status selection. Among those responses, `"revoked"` overrides `"suspended"`, which overrides `"active"`; a disagreement SHOULD record `um:reason:status:federation-inconsistent`. An untrusted resolver's assertion is not authoritative merely because it is restrictive. `"unchecked"` applies when no usable authorized response establishes status, including unreachable or unvalidated resolvers; follow O2.4. Federation does not supply the unresolved protected status-authority mechanism.

Cache invalidation. When a manifest's status changes, the issuing resolver MUST propagate the change to all federated resolvers; the federation SHOULD achieve propagation within 60 seconds for revocation events. During the propagation window, different evaluators may receive different answers. Evaluators SHOULD treat cached status as provisional and re-check per `nextCheck`.

Availability and failover. The federation uses eventual consistency. Evaluators MAY query an authorized alternative resolver when the primary `statusRef` endpoint is unavailable; discovery alone does not authorize a resolver. In split-brain scenarios, select the most restrictive usable response under the authority/target and validity rules above and record `um:reason:status:federation-inconsistent` for disagreement. If no response meets those rules, status is `"unchecked"`, even when an untrusted endpoint returned bytes. A replica or failover response does not by itself establish current permission.

Manifest forwarding. When a manifest is forwarded across federation boundaries, the forwarding agent MUST preserve all fields (unknown fields survive the evaluation sequence). Forwarding metadata MUST NOT be added to the manifest payload, because the payload's bytes are covered by the signature; a forwarding agent MAY convey a `forwardedVia` array (resolver DIDs in path order) in transport-level metadata or a wrapper object that carries the manifest unmodified (the wrapper's shape is transport-profile-defined; this profile constrains only that the carried manifest bytes are unmodified) — such metadata is informational and MUST NOT affect evaluation. Forwarded manifests MUST be re-verified by the receiving evaluator; the forwarding agent's handling does not substitute for the receiver's own evaluation.

Open question: Whether federation should be published as a separate companion specification is open; it introduces protocol-layer complexity that may evolve independently of the manifest format.

✧ ✧ ✧

O6. Profile Registration Mechanism PREVIEW

This part defines the formal registration process for new cryptographic, trust, and domain profiles that extend the Universal Manifest specification, following IANA-style registry conventions. It is the mechanism the Base §5 extension model relies on.

Profile identification. Each registered profile MUST have a canonical identifier following `um:profile:<category>:<name>`. Categories: `signature` (e.g., `um:profile:signature:jcs-ed25519`, `um:profile:signature:jcs-es256` — O15), `trust` (e.g., `um:profile:trust:zkp-bbs-linked-secret`, `um:profile:trust:locked-tier-portable-unlock` — EXT-T1 §T1.3.3), `domain` (domain-specific manifest class profiles, e.g., `um:profile:domain:receipt`), `binding` (credential binding profiles, e.g., `um:profile:binding:sd-jwt-kb`). Each identifier MUST be paired with a human-readable name and version string; identifiers are globally unique and MUST NOT be reused after deprecation. This process also governs the subsidiary value registries referenced elsewhere: receipt event classes (O3.2), receipt reason codes (O3.3), agent-delegation scope values (O7), liveness `proofType` values (EXT-T1 §T1.3), and additional JWE algorithm pairs (Base §2.4).

Registration process (five steps). In this section, **the maintainers** means the maintainers of the canonical `UniversalManifest/universal-manifest` repository, acting through its public process. (1) **Proposal** — a profile author submits a specification to the maintainers as an issue or pull request on that repository, including a scope statement, normative requirements (RFC 2119 keywords), security considerations, an interoperability assessment, and at least one conformance test fixture. (2) **Review** — the maintainers review for completeness, security, and interoperability, and MUST verify the profile does not conflict with existing profiles at the same extension point. (3) **Conformance integration** — the author provides fixtures; the maintainers integrate them into the conformance suite. (4) **Registration** — upon the maintainers' public acceptance, the profile receives its canonical identifier and is added to the registry. (5) **Maintenance** — the author maintains the spec and tests; the maintainers MAY deprecate the profile if maintenance lapses, security issues arise, or a superseding profile is registered.

Conflict resolution. Two profiles defining incompatible semantics for the same extension point MUST NOT both be registered without resolution; the maintainers MUST resolve by merging, selecting one, or defining a compatibility boundary. **Deprecation.** The maintainers MAY mark a profile `"deprecated"` (with a date and superseding-profile reference); evaluators SHOULD warn on deprecated-profile references but MUST NOT reject manifests solely for referencing one until the profile is declared `"sunset"` at a major-version boundary. **Registry hosting.** The registry MUST be published as a standalone document at `https://universalmanifest.net/registry/profiles/`, updated independently of the

specification version; each entry contains the identifier, name, version, status (`"active"`, `"deprecated"`, `"sunset"`), specification URI, and registration date.

Registry-token representation in the JSON-LD encoding (informative). The published v0.5 context gives the registry-token families used across this document set three distinct JSON-LD shapes, and profile authors should account for them at registration time. (1) Receipt `reason` codes (`um:reason:<category>:<code>`, O3.3) ride a plain, uncoerced term — values are inert literal strings, with shipped wire precedent in the conformance fixtures (`um:reason:crypto:key-rotated` / `um:reason:crypto:key-shredded`); design-stage candidate families such as the O2.6 `um:reason:policy:*` and O13.5 `um:reason:route:*` codes follow that same pattern unchanged if finalized. (2) Consent `conditions` values — including the O13.4 `route-class:<value>` candidate convention — are members of an unordered set of plain strings; they undergo no compact-IRI expansion and are compared as opaque tokens. (3) `profile` identifiers (`um:profile:<category>:<name>`) are `@id`-coerced, so a value expands as a compact IRI under the version-independent `um:` prefix; a registered profile identifier is thereby a globally unique IRI, not a bare label. The term-shape asymmetry — reason codes and conditions as opaque strings, profile identifiers as IRIs — is deliberate at this stage: reason codes and conditions are matched by evaluators as tokens, while profile identifiers name registry entries that benefit from global identity. Whether a design-stage candidate token family enters the versioned context, and in which of these shapes, is a context-terms decision taken at wire freeze through this registration process; appearance of a candidate token in a design-stage draft confers no context registration.

The `signature` category and the recognized-suite set. The `signature` category is where signature suites register, and it is the forward path by which the recognized-suite set of Base §1.6 grows. Both suites v0.5 recognizes hold identifiers in this category: `um:profile:signature:jcs-ed25519` (Signature Profile A, the mandatory-to-implement baseline) and `um:profile:signature:jcs-es256` (Signature Profile B, O15). Today that set is materialized on the wire as the `algorithm` enumeration of the published `schema.json`, and a suite is recognized by the five-step process above together with that one-line enumeration edit; the registration is the governing act, and the enumeration is its wire projection. When this registry is published as the standalone document named above, the enumeration becomes the projection of a registry that is edited independently of the specification version — the same relationship, differently hosted. Either way the set is open by construction: any number of suites MAY be registered, and no re-architecture is needed to admit the next one (the post-quantum migration of EXT-T2 §T2.3 is the case this path is being rehearsed for). This part remains PREVIEW. A local draft registry (`registry-profiles.md`) accompanies the specification; its existence does not establish that every listed candidate is registered or active. Recognition explicitly stated in the Base or a normative extension remains effective under that specification. Additional entries become active only through the review, conformance integration and registration steps defined here, with recorded

version, status, specification reference, registration date and registering authority. Provisional names and editorial placeholders remain distinguishable from active registrations.

Open question: Whether low-risk profiles (e.g., new domain entity types) should follow a lighter-weight registration with designated expert review instead of the full process is under evaluation. A second open question is whether a future version should make `schema.json` defer to the published registry rather than enumerate the recognized suites; the enumeration works unchanged if it does not, and becomes the registry's wire projection if it does.

✧ ✧ ✧

O7. Agent-Delegation Scope Registry PREVIEW

This part defines a registry of recognized scope values for the `scope` field in `um:agentDelegation` pointers (Base §6.5), with a namespace convention and registration procedure.

Namespace convention. Scope values MUST follow a dot-separated `<domain>.<capability>` naming (the domain identifies the functional area; the capability the specific authority). Examples: `spatial.session`, `social.messaging`, `commerce.transaction`. Custom scope values defined by profile documents SHOULD use a reverse-DNS prefix to prevent collisions (e.g., `com.example.custom-capability`).

Core scope values. All evaluators recognizing `um:agentDelegation` pointers MUST understand these: `spatial.session` (participate in spatial computing sessions on behalf of the subject); `spatial.navigation` (navigate spatial environments); `social.messaging` (send and receive messages); `social.presence` (represent the subject's presence status); `commerce.transaction` (initiate transactions, within any spending limits declared by profile-defined members — this specification itself defines no spending-limit member); `identity.attestation` (present the subject's identity claims to verifiers — does **not** grant authority to create or modify claims).

Evaluator behavior for unrecognized scopes. An evaluator encountering unrecognized scope values MUST: (1) record them in the receipt; (2) restrict the delegate to only the recognized scopes (the delegate MUST NOT exercise capabilities for unrecognized scopes); (3) if the pointer contains **only** unrecognized scopes, SHOULD treat the delegation as having no effective scope and record this.

Registration procedure. New scope values are registered through the profile registration mechanism (O6); a registration MUST include the scope string, a description, the domain it governs, and any constraints (e.g., "requires `requiredTrustTier` >= 1").

(Non-normative: Mastercard's Verifiable Intent pattern — a three-layer SD-JWT structure for bounded-scope agent delegation — is compatible, each layer mapping to a scope constraint. Open question: whether scope values should use URI-based naming for stronger uniqueness is under evaluation.)

✧ ✧ ✧

O8. Category Trust and `trustWeight` PREVIEW

A recurring failure mode in service directories is treating a self-declared service category (for example "emergency-services" or "verified-merchant") as if it were attested. v0.5 separates the *claim* of a category from its *attestation* by splitting the single self-declared category field into four distinct category-trust claim objects and by replacing the v0.1/v0.3 `interpretedAs: "hint-only"` enum with a typed `trustWeight` field an evaluator can act on uniformly.

The `trustWeight` field. Any claim or category-trust object MAY carry a `trustWeight` field replacing the older `interpretedAs: "hint-only"` convention. `trustWeight` is one of: `"hint"` (self-asserted, unattested; evaluators MUST NOT grant trust-transitive or high-impact authority on a `hint`-weighted value alone — the typed successor to `interpretedAs: "hint-only"`); `"attested"` (backed by a Verifiable Credential or attestation from a named issuer; evaluators verify per the claim-proof process — EXT-T1 §T1.5 — before relying on it); `"authoritative"` (issued by an identity service provider or registry the evaluator treats as authoritative for the category by local policy). When `trustWeight` is absent, evaluators MUST default to `"hint"`. Evaluators encountering the deprecated `interpretedAs` member (from v0.1/v0.3) MUST ignore it; in its absence `trustWeight` defaults to `"hint"`, preserving the prior `interpretedAs: "hint-only"` semantics. The `interpretedAs` member is removed from the v0.5 vocabulary.

Category trust split. The single self-declared category is replaced by four distinct *category-trust objects* (claim-level objects carried in the `claims` array, distinguished by their `@type`; not top-level `um:Facet` objects), each carrying the common claim members of Base §1.4.3 plus `trustWeight`, plus the value-carrying members named here: `operatorIdentity` (the operating entity's identity, signed by an identity service provider — the operator's DID or URI is carried in the base claim `subject` member, with the attesting provider as `issuer`; no additional value member is needed; RECOMMENDED for any non-trivial service); `serviceCategoryClaim` (the self-declared category — a REQUIRED **`category`** member, a string naming the category, drawn from a registered category vocabulary where one exists (O6); always `trustWeight: "hint"` unless paired with a `categoryAttestation`); `categoryAttestation` (a Verifiable Credential attesting the category from a recognized authority — carrying the same REQUIRED **`category`** member, whose value

MUST equal the paired `serviceCategoryClaim`'s `category` for the pairing to hold; REQUIRED for high-impact categories — defined at the profile level, e.g. emergency, medical, financial, child-directed); `urgencyClaim` (any urgency/priority assertion — a REQUIRED **`urgency`** member, a string token with candidate registry values `routine`, `elevated`, `emergency` (extensible via O6), carried as a separate signed claim so urgency cannot be smuggled in via the category field). Normative rules: an evaluator MUST NOT act on a `serviceCategoryClaim` for a high-impact category without a verified `categoryAttestation` (high-impact membership is determined from the `category` value against the profile-level list); MUST verify `categoryAttestation` credentials for revocation per O2; and MUST treat `operatorIdentity` and `serviceCategoryClaim` as independent. The high-impact category list is profile-level, not fixed by this specification, consistent with eIDAS 2.0 [eIDAS2] assurance-level practice.

Preview: The historical v0.4 editorial default proposed `trustWeight` and category-trust semantics. `trustWeight` remains specification-only in this v0.5 candidate and is absent from its context; the value-shape decision and fixture/schema coverage remain prerequisites to carrying it losslessly. Its stated optional semantics and fail-closed requirements remain; this candidate does not register the missing term or claim delivery.

✧ ✧ ✧

O9. Extended Structural Schemas PREVIEW

The Base names three structural-state extensions as reserved members (Base §1.4.4, §1.4.6, §1.4.7); this part specifies them.

O9.1. Derived-Variant Sensor Consent Vocabulary

Spatial-computing sensors expose data at very different sensitivities depending on processing. A single boolean "eye-tracking allowed" consent cannot distinguish "raw gaze vector stream" from "region-of-interest hit only." v0.5 extends the `consents` vocabulary (Base §1.4.4) with **derived-variant** consent keys: each sensor class is qualified by a derivation tier so holders consent to the minimum-sensitivity variant a use case needs. Non-breaking: the v0.1 boolean keys remain valid and are interpreted as the most-permissive variant only when a deployment explicitly opts in; absent an explicit grant for a derived variant, evaluators MUST fail closed.

A derived-variant consent key uses dot-separated `sensor.<class>.<signal>.<derivation>` naming, aligned with the Khronos OpenXR sensor classes [OPENXR]. Eight sensor classes (eye,

hand, face, body, depth, audio, rgb-camera, environment) are each qualified by a derivation tier from `raw` (least processed, most sensitive) through progressively more-derived variants (e.g. `roi-derived`, `event-derived`, `presence-derived`). Examples: `sensor.eye.gaze.raw` (raw gaze vector stream); `sensor.eye.gaze.roi-derived` (region-of-interest hit-test results only); `sensor.hand.pose.raw` (full hand skeleton); `sensor.hand.pose.gesture-derived` (recognized discrete gestures only). Each derived-variant consent entry MUST carry the `purpose` field of Base §1.4.4, binding the grant to a stated purpose so a grant for one purpose cannot be silently reused for another (consent-creep prevention, per GDPR Article 5(1)(b) and ISO/IEC 27560:2023). The `purpose` value SHOULD be drawn from the W3C Data Privacy Vocabulary where an applicable term exists. An evaluator MUST match the most specific derived-variant key granted: a grant for `sensor.eye.gaze.roi-derived` MUST NOT be read as granting `sensor.eye.gaze.raw`. A grant for a less-derived (more sensitive) variant MAY be treated by policy as also authorizing strictly more-derived variants of the same signal, but a deployment doing so MUST document the implication in its conformance claim.

Carrier (candidate). The derived-variant sensor key is carried **folded into consent `scope`**: each key (`sensor.<class>.<signal>.<derivation>`) is a consent-`scope` operation token (Base §1.4.4), following the same dot-token discipline as the agent-delegation scopes (O7) and the write verbs (Base §3.1.4) — no dedicated `sensorConsent` member is defined. A sensor grant is therefore an ordinary consent entry whose `scope` contains the sensor token and whose `purpose` carries the mandatory purpose binding above; its `facetRef` names the facet through which the sensor data is exposed or into which it is written (a deployment MAY model a sensor stream as its own facet for this purpose). Stage-4 matching applies literally, and the token that authorizes a source sensor read — including for the derived-write precedence rule of Base §7.2 — is the most specific granted token per the specificity rule above. The tokens are design-stage candidates and enter the context/registry only at wire freeze (O6).

(Worked example: derived-variant sensor consent with purpose binding — see Cookbook.)

Preview: Built on the editors' default (add in v0.5; small, non-breaking). The carrier question is resolved at design stage to scope-folding (above); input is requested on the full derivation-tier enumeration per sensor class.

O9.2. Two-Component `devices` Schema

The `devices` array (Base §1.4.6) registers hardware endpoints (XR headsets, NFC readers, smart displays, wearable sensors) associated with the subject. v0.3 reserved this member; v0.5 defines device entries as a **two-component split** mirroring the layered device-attestation models of FIDO, TPM 2.0, WebAuthn [WEBAUTHN] (Level 2 as the normative anchor; Level 3 is a W3C

Recommendation dated 25 August 2026), Android Hardware Attestation, and Apple DeviceCheck: a long-lived hardware provenance component and a session-scoped capability component. Separating them lets a session-only manifest advertise device capabilities without leaking a long-lived hardware identifier. Each entry is a device object that MAY carry a `deviceAttestation`, a `deviceCapability`, or both; it MUST include a `@type` of `"um:Device"` and an `@id`. The `deviceAttestation` and `deviceCapability` sub-objects are device *components*, not `um:Facet` objects.

`deviceAttestation` (long-lived, manufacturer-signed). When present it MUST contain `deviceClass` (a string, e.g. `"xr-headset"`, `"nfc-reader"`, `"wearable-sensor"`), `modelHash` (a hash identifying the model/firmware baseline, suitable for matching against a manufacturer registry without revealing a per-unit serial), and `attestation` (a Verifiable Credential or platform attestation object signed by the manufacturer or attestation authority — e.g. a WebAuthn/FIDO device attestation or a TPM 2.0 [TPM2] quote; evaluators MUST verify it against a configurable manufacturer trust anchor before relying on the device's hardware claims). It MAY carry `manufacturer` (DID/URI) and `attestedAt` (RFC 3339). Because hardware identifiers are long-lived and potentially correlatable, holders SHOULD omit `deviceAttestation` from session-only or pseudonymous manifests and carry only `deviceCapability`.

A deployment profile relying on `modelHash` MUST define the model artifact bytes or canonical serialization, digest algorithm and output encoding, version and any preprocessing, and how the value is authenticated by the evidence relied upon. Without that definition and evidence binding, an evaluator MUST treat the value as unverified and MUST NOT use it to raise assurance or authorize a model-specific operation. Equality with an untrusted holder-supplied string is not model-integrity verification. This PREVIEW note selects no global digest, model canonicalization, or default artifact-discovery mechanism.

`deviceCapability` (session-scoped, user-signed). Describes what the device can do for the current session, signed by the subject (or a session-scoped key), not the manufacturer. When present it MAY contain `sensors` (sensor-class identifiers, e.g. `"eye-tracking"`, `"hand-tracking"`, `"depth"`, `"rgb-camera"`, using OpenXR sensor-class naming), `openxrExtensions` (OpenXR extension identifiers the device/runtime supports), `positioningTier` (e.g. `"3dof"`, `"6dof"`, `"world-scale"`), `processingConstraints` (compute/thermal/power limits an evaluator should respect), `privacyModes` (e.g. `"on-device-only"`, `"roi-derived-only"`), and `sessionSigningKey` (a session-scoped ephemeral public key used to sign per-session device assertions; MUST NOT be a long-lived hardware identifier; generated per session to avoid cross-session linkage). Evaluators MUST NOT treat `deviceCapability` claims as hardware-attested unless a corresponding verified `deviceAttestation` is present on the same entry; absent such attestation, `deviceCapability` is a self-declared, session-scoped assertion (analogous to a hint-weight claim — O8).

The `devices` array is part of the signed payload: holders include it when signing, and evaluators MUST include it when recomputing the signing input and MUST NOT discard it before verification or during Arrive-stage unknown-field handling (it is a named structural member). Evaluators that do not implement the device components MUST preserve the array and record entries as present-but-unprocessed.

Device-local observations and holder-signed platform state are self-declared evidence to a remote evaluator. Neither local non-exportability observations, device-genuineness tokens, a claimed hardware backend, nor storage of a hardware-wrapped blob alone establishes verified provenance for the authenticator that performed the gesture. The evaluator MUST apply EXT-T1 §T1.3.3 to evidence it actually validated and MUST NOT raise the class from the holder's assertion. Verified hardware-backed user verification and verified device-bound provenance remain separate findings.

A declaration about correlation is a hint for minimization, not evidence of privacy. A device-model string does not uniquely identify every unit, but combinations of model, boot state and other metadata can fingerprint; the existing bounded selective-minimum-disclosure rules apply to this metadata. A remote receipt MUST NOT imply that a local assertion was independently verified.

(Worked example: device entry with both components — see Cookbook.)

Preview: The two-component split is built on the editors' default (define in v0.5), because the wire shape is not yet fielded and this is the most consequential wire-shape decision in this version. Attestation transparency logs for device attestations remain outside this candidate; the historical v0.4 draft flagged them for later work, without selecting or qualifying a mechanism. Input is requested on the sensor-class and OpenXR-extension vocabularies and on session-key generation requirements. Two member-precision decisions are reserved to the wire freeze so they are not lost: the `modelHash` computation rule (which bytes are hashed — e.g. a manufacturer-published model/firmware descriptor — and the matching procedure against a manufacturer registry) and the `sessionSigningKey` representation (JWK vs. multibase); until fixed, both are deployment-profiled.

O9.3. `actorState` Member

The `actorState` member (Base §1.4.7) is an OPTIONAL top-level object declaring who is operating the session the manifest is presented in: the human principal, a delegated agent, or a hybrid. It bridges the `um:agentDelegation` pointer (Base §6.5) — which declares that delegation *exists* — to the session-state semantics of who is *currently acting*. RECOMMENDED whenever an `um:agentDelegation` pointer is present.

When present, `actorState` MUST contain `principal` (the DID of the human or legal entity the manifest represents; MUST match the manifest `subject`; an evaluator finding `actorState.principal` not equal to `subject` MUST record this as a verification failure and MUST NOT rely on the manifest for delegated-authority decisions). When present, it MAY contain an `executor` object describing who is operating on the principal's behalf: `type` (one of `"human"`, `"agent"`, `"hybrid"`; MUST be present when `executor` is present); `delegateId` (the DID of the operating agent; REQUIRED when `type` is `"agent"` or `"hybrid"`); `delegationRef` (a reference to the `um:agentDelegation` pointer by `@id` that authorizes this executor; when present, evaluators MUST confirm the referenced pointer exists, is unexpired, and names `delegateId` as its delegate); `lastVerifiedAt` (RFC 3339 of the last interactive verification of the principal; meaningful only when `type` is `"human"` or `"hybrid"`; relates to the liveness model in EXT-T1 §T1.3). When `executor` is absent, evaluators MUST treat the session as principal-operated (`type: "human"` with no delegate). A `"hybrid"` executor MUST be treated as agent-operated by any gate that distinguishes human-operated from agent-operated sessions (for example a human-only-session requirement — Base §6.5.2): partial human participation does not satisfy a human-only gate (fail closed). `actorState` is additive and non-breaking; v0.3 manifests omit it, and evaluators that do not implement it record it as present-but-unprocessed.

(Worked example: actorState with an agent executor — see Cookbook.)

Preview: Built on the editors' default (add as OPTIONAL, RECOMMENDED with `um:agentDelegation`). It is a wire-shape addition. Input is requested on whether `actorState` should become REQUIRED when any delegation pointer is present.



O10. Cryptographic Requirements Summary PREVIEW

This consolidates the cryptographic requirements scattered across the signature profiles (Base §1.6, O15), encrypted facets (Base §2.4), the tiered trust model (Base §6.4.2), the ZKP profiles (EXT-T2 §T2.1), the ceremony model (EXT-T3), credential binding (EXT-T1), and post-quantum signatures (EXT-T2 §T2.3) into a single reference. This is a summary; the normative requirement for each algorithm lives in its referenced section, so this table can be updated as profiles are added without editing those sections.

Requirement level	Algorithms / cryptosuites	Reference
Mandatory-to-implement	Ed25519 over JCS-RFC8785 (Signature Profile A); SD-JWT Key Binding (KB-JWT); JWK Thumbprint; ECDH-	Base §1.6, EXT-T1 §T1.1, Base

Requirement level	Algorithms / cryptosuites	Reference
	ES+A256KW / A256GCM for encrypted facets	§2.4
Recommended	BBS+ signatures and BBS+ derived proofs (BLS12-381); WebAuthn device/liveness attestation (explicitly versioned assertion validation; Level 3 Recommendation for BE/BS semantics)	EXT-T2 §T2.1, EXT-T1 §T1.3
Optional (recognized additive signature suite)	ECDSA over NIST P-256 with SHA-256 (`ES256`) over JCS-RFC8785 — Signature Profile B, the hardware-custody suite; raw 64-byte `R || S`, no low-S rule, no RFC 6979 requirement	O15
Optional (Tier 2 profiles) †	Groth16 (HD-derivation proofs); BBS+ linked-secret equality proofs	EXT-T2 §T2.1
Future	FROST threshold Schnorr/Ed25519 [RFC9591]; threshold BBS+; ML-DSA / SLH-DSA / FN-DSA post-quantum signatures	EXT-T3, EXT-T2 §T2.3

† Items marked "Optional (Tier 2 profiles)" are optional at the specification level but become mandatory *within* the named optional profile: Groth16 is REQUIRED for an implementation that supports Profile 2B. The mandatory-to-implement row is likewise scoped to the module that uses each algorithm: Ed25519 over JCS-RFC8785 is unconditional for every implementation; the JWE pair is mandatory for evaluators claiming encrypted-facet support; SD-JWT Key Binding and JWK Thumbprint are mandatory for implementations claiming credential-binding support.

Preview: Built on the editors' default (add as a reference summary). Input is requested on whether any "recommended" item (notably BBS+ for privacy-preserving Tier 2) should be elevated to mandatory-to-implement for v0.5.



O11. Optional-Feature Matrix (consolidated)

This consolidates Base §4.7 with the modules this companion introduces. An implementation may leave an optional module unimplemented only while observing every applicable mandatory baseline. Partial implementation MUST enumerate implemented and unmet obligations; a partial declaration does not waive a required check or establish conformance to a class whose requirements remain unmet. Declared support, structural validation, or partial support MUST NOT raise a document's verification result, assurance class, or trust tier.

Feature	Conformance	Mandatory baseline when not implemented
Signature Profile B (JCS + ES256 — O15)	OPTIONAL	Verify Signature Profile A, the mandatory baseline (Base §1.6); reject a Profile-B-signed manifest recording `signatureCheck: "unsupported-profile"` — never reinterpret it under Profile A.
Encrypted-facet decryption (Base §2.3)	OPTIONAL	Sealed-entry handling REQUIRED: record present-but-sealed; never infer content.
Revocation-aware verification (O2)	RECOMMENDED	Record `revocationStatus: "unchecked"`; do not grant revocation-gated trust.
Credential binding (EXT-T1)	OPTIONAL (REQUIRED for Tier 1+)	Record `holderBindingStatus: "absent"`; cap relied-upon claims at Tier 0.
Per-facet floors (`requiredLiveness`, `requiredAssuranceClass` — EXT-T1 §T1.3.1 / §T1.3.3, PREVIEW)	OPTIONAL (PREVIEW)	A floor-bearing facet MUST be withheld (recorded `assuranceInsufficient` for either unmet floor, with the actual failure reason), never processed — fail closed, without silent downgrade.
CBOR-LD encoding (O1.4)	OPTIONAL	Support JSON-LD reference encoding; reject unsupported encodings.
Transparency anchoring (O3.4)	OPTIONAL	Verify the manifest signature and proofs directly; do not require a transparency log.
Tier-2 cryptographic binding (EXT-T2)	OPTIONAL	Record `"trustTierUnsupported"` for Tier-2-required items; do not downgrade.
Tier-3 ceremony (EXT-T3)	OPTIONAL	Record `"trustTierUnsupported"` for Tier-3-required items; do not downgrade.
Bilateral session model (O4)	OPTIONAL	Manifest-level bilateral exchange (Base §6.4.6) still applies.
Federation (O5)	OPTIONAL	Single-endpoint statusRef resolution (O2) applies.
Current policy-state profile (O2.6, PREVIEW design-stage)	OPTIONAL	The bounded composition of Base §3.4.1 applies: TTL, statusRef, consent windows, fail-closed.

Feature	Conformance	Mandatory baseline when not implemented
Represented-person consent profile (O12, PREVIEW design-stage)	OPTIONAL	Existing fail-closed rules apply: an unrecognized claim type is unprocessable above Tier 0 (Base §1.4.3); an unrecognized or unenforceable consent condition fails closed (Base §1.4.4).
Referral-route attribution profile (O13, PREVIEW design-stage)	OPTIONAL	An unrecognized pointer type is recorded but not acted on (Base §1.4.5); a consent conditioned on a route class the evaluator cannot verify fails closed (Base §1.4.4).
GeoPose primary pose value profile (O14, PREVIEW design-stage)	OPTIONAL	A GeoPose-bearing manifest still follows ordinary UM behavior: unknown claims/pointers are preserved but not acted on, unverified provenance caps trust, and missing or unverifiable pose data MUST NOT be invented.

O11.1. Conformance-Claim Contents (candidate template) PREVIEW

Design-stage. Requirements across this document set oblige a deployment to *document* choices in its conformance claim, but no claim format has been defined. This subsection names the candidate template so those obligations have a single landing place; the claim-document format itself is a registration-time deliverable (O6) and adds no manifest wire member.

Truthful obligation-level reporting. Irrespective of document format, a capability claim MUST identify the specification/profile version, implementation version, supported encodings and suites, each claimed obligation by exact source section/requirement, its state (`implemented`, `partial`, or `not-implemented`), actual implemented checks, unmet checks, the mandatory baseline for those unmet checks, finite limits, and supporting executed evidence. `partial` requires both implemented and unmet checks; `implemented` cannot hide unmet checks within the claimed obligation; `not-implemented` must state the concrete applicable baseline. Obligation identifiers MUST be unambiguous and non-duplicated. An unexecuted cryptographic primitive MUST be stated explicitly even when parsing, policy, or error-handling code exists.

Reports MUST distinguish normative checks, structural PREVIEW checks, and design-stage candidates, and identify the checks actually executed. Matching a verdict alone does not prove that required cryptographic or behavioral checks ran. Numerical totals MUST NOT imply verification of

opaque proofs, placeholders, signatures, or isolation properties that were not checked. A design-stage candidate without an operative contract is not executed, not a passing structural test. Evidence for shape validation cannot prove a cryptographic requirement. These rules govern claim content and evidence; they create no new manifest member, capability-negotiation carrier, authenticated wire advertisement, or signature requirement for such a carrier.

A v0.5 conformance claim SHOULD be a published, structured document stating, at minimum:

1. **Identity and class** — the implementation and version; the conformance class(es) claimed (evaluator, holder, bilateral participant, status endpoint — Base §4); the production rules supported (O1.2).
2. **Modules and PREVIEW features** — each obligation's implemented/partial/not-implemented state and mandatory baseline for unmet checks, grouped by the optional modules above; identify which PREVIEW features the claim includes (per the two-grade PREVIEW policy; design-stage profiles MUST NOT be claimed — Base §4.4).
3. **Documented deviations and policies** — every choice the normative text requires a claim to document: a widened clock-skew tolerance (Base §3.1.2); resource-limit deviations (Base §6.3); purpose-vocabulary hierarchy use (Base §1.4.4); the sensor derivation-implication policy, where a less-derived grant is read as also authorizing more-derived variants (O9.1); and any stricter VP size limits (EXT-T1 §T1.5.1).
4. **Isolation claim** — where per-facet isolation is claimed (Base §2.3.5), the declared isolated facet set (the named set, or the deployment rule identifying it); this is the carrier of the conformance-claim flag that section defines.
5. **Suite evidence** — the exact source/index/input identities, implementation/runtime, checks actually executed, results and limits, alongside the claimed level (Base §4.4). Passing a finite suite is bounded evidence, not certification of every normative obligation.



O12. Asset-Bound Manifests and Represented-Person Consent Profile

PREVIEW

Candidate registered profile (`um:profile:domain:represented-person-consent`, registration via O6). **Design-stage:** the class, members, and value tokens below are candidates whose wire shape may change; they are **not yet** in the published `schema.json`/`schema.jsonld` or the conformance fixture suite, and per the Base conformance discipline (Base §4.4) they carry no conformance tests until the design is finalized.

O12.1. Motivation and Role Vocabulary

A portable digital asset can contain a real person: a likeness, a body or face scan, a performance capture, a voice, a synthetic replica. When the asset ports between worlds, the geometry travels; the consent of the person inside it may not. Ownership records answer *who controls the asset*; they do not answer *whether the person represented inside it agreed, and to what*. This profile carries that answer with the asset, as a Universal Manifest record.

The profile distinguishes roles that other systems routinely collapse. **Represented person** — the person depicted, embodied, scanned, performed, voiced, or replicated inside the asset. **Asset owner** — the party controlling or possessing the asset. **Token owner** — the holder of a token (e.g., an NFT) referencing the asset. **Rights holder** — a party with an IP interest in the asset. **Operator** — the platform or runtime handling the asset. **Holder/presenter** — the party presenting the manifest (Base §1.0). These MAY coincide in a given deployment, but an evaluator implementing this profile MUST NOT treat asset ownership, token ownership, rights holding, or operator authority as evidence of represented-person consent: only a consent record whose grantor is (or verifiably acts for) the represented person is represented-person consent.

O12.2. Asset-Bound Manifest Class

An **asset-bound manifest** is a manifest class (Base §1.0.1) whose `subject` is the asset's stable identifier URI rather than a person. Its characteristic members are: a **representation facet** describing what or whom the asset represents, plus the represented-person consent record of O12.3. Like every manifest class, it reuses the common envelope unchanged — signature, TTL, consents, claims, receipts, and the evaluation sequence all apply as specified in the Base.

Signing anchor (non-DID subject). An asset URI is typically not a DID and resolves to no DID document, so the subject-DID key-authorization step of the Tier-1 verification chain (EXT-T1 §T1.5.2, step 2) cannot apply to this class as written. For an asset-bound manifest, the candidate rule is: the signing key's authorization is checked against the **signer's own** DID document — `signature.keyRef` SHOULD be a DID URL so the controller DID is identifiable; the evaluator resolves that controller's DID document and confirms the key appears under the `authentication` or `assertionMethod` verification relationship, exactly as step 2 checks the *subject's* document for person-subject manifests. The signature thereby proves *who produced the record*, not that the producer holds any authority over the asset (the Base §6.4.1 limits apply): the signer's relationship to the asset — owner, operator, rights holder, or the represented person's delegate — is an assertion, and where it must be verifiable it is carried as an ordinary claim with `claimProof`. Represented-person consent verifiability is unaffected by the non-DID subject: it rides the consent-

grant claim (O12.3), whose Tier-1 verification anchors on the represented person as the claim's issuer, not on the manifest `subject`.

O12.3. Represented-Person Consent Record

The profile expresses represented-person consent through existing member kinds, specialized — not through a new pipeline:

- **Representation facet.** A facet (candidate `@type` value: `um:RepresentedParty` on its entity) identifying the representation. Both of its characteristic members are REQUIRED: `representedParty` (DID or URI of the represented person, or an opaque pairwise identifier where the person's identity must itself stay private), and `relationship` (a value naming how the person is inside the asset; candidate registry values: `likeness`, `scan`, `performance-capture`, `voice`, `synthetic-replica`, `composite`, `training-derived`; extensible via O6). A representation facet missing either member does not identify a representation, and a consent record referencing it fails closed.
- **Consent entry.** A standard `consents[]` entry (Base §1.4.4) whose `grantor` is the represented person (or their verifiable delegate) and whose `facetRef` names the representation facet. Because an asset-bound manifest's `subject` is the asset, the Base's absent-`grantor` default (grantor = manifest `subject`) would name the asset itself; a represented-person consent entry therefore MUST carry an explicit `grantor`, and an entry without one is not represented-person consent. Its `scope` uses the `<domain>.<capability>` convention (Base §3.1.4) with candidate use verbs: `use.display`, `use.port`, `use.transform`, `use.regenerate`, `use.train`, `use.commercial`, `use.agentic-action`. All existing consent semantics apply unchanged: fail closed on unlisted operations and unrecognized conditions, purpose matching, validity windows, `withdrawnAt`.
- **Consent-grant claim.** A claim (candidate `@type`: `consent.representedPerson`) whose `issuer` is the represented person (or their delegate) and whose `claimProof` carries the verifiable grant (e.g., a Verifiable Presentation), giving the record Tier-1 verifiability under the existing claim-proof path (Base §6.4.3, EXT-T1). Absent a verifiable grant, the record is a Tier-0 self-assertion by the manifest signer and MUST be treated as such (Base §6.4.1). Verifiability and identity privacy pull against each other here, and the tension is stated rather than hidden: Tier-1 verification anchors on an identified issuer, so a grant claim issued under the represented person's own DID re-identifies the person to every evaluator that verifies it — however `representedParty` is expressed. Where the person's identity must stay private *and* the grant must verify, the grant anchors on a verifiable delegate (evaluators resolve the delegate's DID, not the person's), or the identifying members ride sealed entries readable only

by designated recipients (Base §2.3); the unlinkable-verification route is the EXT-T2 track, and this profile does not claim it.

Grant↔entry correspondence (fail closed). The Stage-4-operative record is the consent *entry*; the verifiable record is the consent-grant *claim*. When a deployment relies on the grant for Tier-1 verifiability of represented-person consent, the two MUST correspond: the entry's `facetRef` MUST name the representation facet the verified grant covers, and the entry's `scope` and `purpose` MUST be within the uses and purpose the verified grant states — an evaluator MUST NOT act at Tier 1 on a consent entry whose scope exceeds the verified grant (a verified `use.display` grant never Tier-1-authorizes a `use.train` entry). The consent entry SHOULD carry a candidate **`grantClaimRef`** member — the `@id` of its consent-grant claim — making the correspondence mechanically checkable (the same reference discipline as `actorState.delegationRef`, O9.3; the name is deliberately distinct from the receipt-side `claimRef` status field, Base §3.3.1, which is already a published context term). On any mismatch, or where the referenced claim fails verification, the consent entry is **not** verified represented-person consent: it is a Tier-0 self-assertion by the manifest signer and MUST be treated as such; a deployment requiring verified represented-person consent then fails closed, exactly as for an absent record (O12.4).

O12.4. Carrier Binding

How the record travels with the asset is transport, not trust. Candidate carriers — an embedded UM record inside the asset container, an embedded UM locator, a sidecar file, a C2PA assertion [C2PA], a 3D-container metadata extension, a watermark- or fingerprint-derived locator, or a URI-bound reference — are all acceptable; the trust model is always the UM record itself: its signature, its consent entries, its claim proofs, and its current-state resolution. To bind the record to specific asset bytes, the asset-bound manifest SHOULD carry an **`assetBinding`** object (candidate member): `digest` (a multihash over the asset bytes or a defined container scope; the multihash function MUST be SHA-256 for v0.5, the same discipline as O1.7), `bindingScope` (what the digest covers — candidate value pattern: `whole-asset` for the complete asset artifact, or `container:<path>` naming a container-relative scope; further forms register via O6), and optionally the carrier kind. The manifest signature then covers the binding, so the consent record cannot be silently re-attached to a different asset. A carrier can be stripped; a stripped asset simply arrives with no manifest, and a deployment requiring represented-person consent MUST treat *absent record* as *absent consent* (fail closed), exactly as Stage 4 treats an ungoverned facet. The same fail-closed rule covers derivatives: an authorized transformation (e.g. `use.transform`, `use.regenerate`) produces new asset bytes outside the original `assetBinding` digest, so the original record neither travels to nor authorizes the onward use of the derivative — a deployment requiring represented-

person consent needs a record bound to the derivative's own bytes. Automatic consent carry-forward across derivations is out of scope for this profile.

O12.5. Current Consent-State Resolution

Represented-person consent changes over time — renewal, withdrawal, expiry, supersession. The record resolves currency through the existing machinery composed in Base §3.4.1 and, where implemented, the current policy-state profile (O2.6): manifest TTL bounds staleness; `signature.statusRef` (and a `claimProof` entry's own `statusRef`) resolves revocation/supersession; a `policyWindow` with `deny`/`escalate` dispositions governs unresolved or stale state before a governed use — display, porting, transformation, regeneration, training, or an agentic action — proceeds; and the receipt records the outcome with the O2.6 reason codes. The two status channels are not interchangeable: the manifest-level `signature.statusRef` is operated by whoever signed the asset-bound manifest — typically the operator, owner, or rights holder — and cannot be assumed to reflect the represented person's withdrawal; the person's (or their delegate's) independent withdrawal channel is the consent-grant claim's own `statusRef` (O12.3). A deployment requiring *current* represented-person consent MUST therefore resolve the claim-level status where the grant exposes one, not only the manifest-level status. Withdrawal is forward-only: it governs future processing and does not un-render, un-copy, or un-train what a then-valid consent already authorized (Base §7.2).

O12.6. Honest Limits

This profile makes represented-person consent *portable, verifiable, and currency-checkable*. It does not prove the asset actually depicts the named person (that is an attestation/provenance question for issuers and attesters, e.g. C2PA-style capture provenance carried as `claimProof` material); it does not make carriers strip-proof (carrier robustness — watermark survival, container integrity — is deployment- and carrier-specific and outside UM); it does not adjudicate likeness rights (this is standards/runtime machinery, not legal advice); and enforcement against a runtime that ignores the record is a platform/governance matter — UM makes honoring and violating the record *auditable*, via receipts. Finally, the profile does not by default hide *who is inside the asset*: a resolvable `representedParty` identifier — or a verified consent-grant claim issued under the person's own DID — discloses the person↔asset link to every recipient of the asset and its manifest. Deployments for which that link is itself sensitive use the pairwise-identifier and sealed-entry options, with the verifiability trade-off stated in O12.3, and SHOULD treat the representation facet as high-sensitivity content under the ordinary projection and consent machinery (Base §3.1.3, §3.1.4).

O13. Signed Referral-Route Attribution and Route-Scoped Disclosure Profile PREVIEW

Candidate registered profile (`um:profile:domain:referral-route-attribution`, registration via O6). **Design-stage:** the members and value tokens below are candidates whose wire shape may change; they are **not yet** in the published `schema.json`/`schema.jsonld` or the conformance fixture suite, and per the Base conformance discipline (Base §4.4) they carry no conformance tests until the design is finalized.

O13.1. Scope Boundary

The route by which two parties reach each other — a directory listing, a bulletin, a matching agent, a referral — is today either an unverifiable browser artifact (`Referer`) or a captive record inside one platform. This profile makes route provenance a signed, portable, minimal evidence layer. The boundary is strict: Universal Manifest is **not** an advertising network, a marketplace, a payment rail, a pricing engine, a platform-fee enforcement system, or a tax/royalty-distribution authority, and this profile MUST NOT be implemented as one. UM's role ends at *evidence, disclosure policy, and receipts*: proving a route occurred, letting consent key off the route, and issuing a minimal attribution record that external payment, royalty, or commerce systems MAY honor — without UM moving funds or ranking anything. The browser `Referer` header MUST NOT be treated as an authoritative route signal.

O13.2. Route Actors and Role Classes

A **route actor** is the party that listed, matched, introduced, or forwarded the encounter. Candidate role-class registry values: `directory`, `bulletin`, `utility`, `matching-agent`, `delegated-agent`, `affiliate`, `platform` (extensible via O6). A route actor's *authority to act in a role* is itself an ordinary UM question: it can carry a claim with `claimProof`, be scoped by an agent delegation (Base §6.5, O7), and resolve currency via `statusRef` (O2) — so route authority is revocable and currency-checkable like any other governed state.

O13.3. Signed Route Evidence

Route evidence rides the existing `pointers[]` member as a candidate pointer type **`um:routeAttribution`** (which, like `um:agentDelegation`, replaces the base `target` requirement with its own fields — Base §1.4.5). Candidate members:

- `routeActor` (REQUIRED) — DID (or service identifier) of the party that listed, matched, introduced, or forwarded the encounter.
- `roleClass` (REQUIRED) — one registry value of O13.2.
- `routedAt` (REQUIRED) — RFC 3339 timestamp of the route step.
- `audience` (REQUIRED) — the destination party of the route step (DID or origin), the same audience-binding discipline as `presentationProof` (EXT-T1 §T1.2).
- `challenge` (OPTIONAL; REQUIRED for interactive presentations) — the verifier-issued nonce of the presentation the route evidence accompanies (the same anti-replay model as `presentationProof`).
- `prevRouteRef` (OPTIONAL) — the `@id` of the immediately preceding route-attribution entry in a multi-hop route.
- `routeProof` (REQUIRED) — the route actor's proof object: `proofType` (candidate baseline `"did-auth"` — an Ed25519 signature by a route-actor key), OPTIONAL `verificationMethod` (a DID URL naming the signing key; when absent, resolution starts from `routeActor`), and `proofValue` (base64url, unpadded).

Signing input (candidate layout). `routeProof.proofValue` is computed over the UTF-8 bytes of the JCS canonicalization (the JSON Canonicalization Scheme already mandatory for Signature Profile A — Base §1.6.1) of the five-element array `[subject, roleClass, routedAt, audience, challenge]` — the routed manifest's `subject` string, then the entry's `roleClass`, `routedAt`, `audience`, and `challenge` values in that order, with `challenge` the empty string when absent. The **routed subject** is the manifest `subject`, not the manifest `@id` and not a signing-input hash: the route step binds the routed *party*, is typically minted before the presentation manifest exists, and cannot include that manifest's signing-input hash without circularity — `pointers[]` lies *inside* the signing input (Base §1.6.3), unlike the excluded `presentationProof`. Signing the role and timestamp alongside subject and audience prevents a proof issued in one role from being replayed as evidence of another role, destination, or time.

Key resolution. The proof verifies against a verification method of the **route actor**: resolve the `routeActor` DID document and confirm the signing key appears under the `assertionMethod` verification relationship — the same key-purpose discipline as the EXT-T1 §T1.5.3 table (route attestation is an assertion about the encounter, not an authentication of the actor to the evaluator). Where `routeActor` is not a DID, verification requires a deployment-pinned key for that service identifier; absent one, the step is unverifiable and is recorded as unverified.

Multi-hop routes. A multi-hop route is a chain of route-attribution entries, each step independently signed; the chain is evidentiary, not hash-linked — order is established by `routedAt`, with `prevRouteRef` as the explicit link where timestamps do not suffice (equal or unordered

timestamps). An evaluator verifies the steps it needs and records unverifiable steps as unverified rather than inventing route trust; **any one verified route entry that meets the route-actor trust requirement of O13.4 and whose `roleClass` matches a `route-class:<value>` condition satisfies that condition** — additional unverified entries never defeat such a match. An unrecognized `um:routeAttribution` pointer degrades safely under the existing rule: recorded as present, not acted on (Base §1.4.5).

O13.4. Route-Scoped Disclosure

The route someone arrives by can change what they are allowed to see — a public-directory arrival may justify less disclosure than a verified matching-agent introduction. This profile expresses that as **consent conditions keyed by route class** (candidate condition convention: `route-class:<value>`, e.g. `route-class:verified-directory`) on ordinary consent entries (Base §1.4.4). The evaluator matches *verified* route evidence (O13.3) against the condition — and verification alone is origin, not authority: a verified `routeProof` proves *which* route actor signed the step, and nothing about whether that actor legitimately holds the asserted role; the bag-of-claims limitation (Base §6.4.1) applies to route evidence exactly as to claims, and a bare `roleClass` value is otherwise self-asserted by construction. A `route-class:<value>` condition is therefore satisfied only by a verified route entry whose `routeActor` the evaluator trusts *for that role class* under an explicit trust basis — a deployment-pinned route-actor set, a verified role-authority claim with `claimProof` (O13.2), a delegation whose scope covers the role (O7), or an equivalent locally configured anchor — with the actor's authority currency-checkable via `statusRef` (O13.2). A consent conditioned on a route the evaluator cannot verify — including the case where no trust basis exists for the asserted role class — fails closed under the existing unrecognized/unenforceable-condition rule. Route-scoped disclosure is an instance of the bounded selective-minimum-disclosure profile (Base §3.1.3, §7.1): the route is part of what the target *is allowed* to see, never a reason to disclose more than the interaction needs.

O13.5. Attribution Receipts and the Privacy Boundary

When a verified route contributes to a governed action, the evaluator MAY record it with the candidate receipt event class `route-attributed` and reason codes `um:reason:route:credit-recorded`, `um:reason:route:unverified`, `um:reason:route:ineligible`, `um:reason:route:overbroad`, and `um:reason:route:disclosure-denied` (O3.2, O3.3). The **proof-to-referrer receipt** is the minimal attribution record a route actor receives: its own identity and role class, the session-scoped correlator (`exchangeId` — inside a bilateral session, the session's own; outside one, a fresh single-use correlator generated for the encounter under the

same rules, at least 128 bits of entropy and no party identifiers (O4.1), so attribution receipts pair without requiring a session), the attribution outcome, and a timestamp. It MUST NOT carry the counterparty's identity, sealed fields, transaction contents, prices, or any private buyer data unless that data is independently consented under its own consent entry. The *attribution outcome* it conveys is likewise minimal — whether credit was recorded for the encounter, nothing more: the `um:reason:route:*` detail above is evaluator-receipt evidence owed to the parties of the evaluation, and the counterparty's per-facet consent and disclosure decisions MUST NOT be conveyed to the route actor through the outcome field or otherwise. A downstream payment, royalty, or commerce system MAY treat the receipt as evidence that a route contributed to an encounter — settlement, splitting, and pricing happen entirely outside UM. Receipt minimization and session-scoped receipt-signing guidance (Base §7, O3.1) apply so attribution records do not become long-lived cross-encounter correlators of the referred party.

O13.6. Honest Limits

The profile proves *that a signed route step occurred and was accepted*; it does not prove the route actor's marketing caused the encounter, does not prevent route actors from colluding off-manifest, does not rank or price routes, and does not give the referred party unlinkability across encounters (the bounded selective-minimum-disclosure non-claims of Base §7.1 apply; the PREVIEW BBS+ selective-disclosure track is in EXT-T2). Nor does it hide from a route actor what that actor itself brokered: a route actor knows the routed party's `subject` identifier and the destination `audience` at mint time by construction, because it signs them (O13.3) — the profile bounds what a route actor *additionally* learns through attribution; it does not un-know the introduction.

O14. GeoPose Primary Pose Value Profile PREVIEW

Candidate registered profile (`um:profile:domain:geopose-primary-pose`, registration via O6). **Design-stage:** the class, members, and value tokens below are candidates whose wire shape may change; they are **not yet** in the published `schema.json`/`schema.jsonld` or the conformance fixture suite, and per the Base conformance discipline (Base §4.4) they carry no conformance tests until the design is finalized.

O14.1. Scope Boundary

When a Universal Manifest subject is a real-world object, a digital twin, a 3D asset, a scan, a dataset record, a scene element, a vehicle, a sensor observation, or another object whose primary

spatial fact is *position plus orientation*, the primary pose value SHOULD be expressed as an OGC GeoPose 1.0 payload [OGC-GEOPOSE] or a pointer to one. The GeoPose value answers "where is this object, and how is it oriented, in a geographically anchored frame?" Universal Manifest answers "who asserted it, what evidence and policy govern it, what may be disclosed, and what did the evaluator do with it?"

This profile therefore does not redefine GeoPose. OGC GeoPose remains authoritative for the pose data object, its Basic-YPR, Basic-Quaternion, Advanced, Chain, Graph, Regular Time Series, Irregular Time Series, and Stream targets, and their JSON-schema conformance. Universal Manifest carries or references that value inside the UM envelope so signatures, claim proofs, consent, current-state resolution, selective minimum disclosure, asset binding, and receipts can govern use of the pose.

O14.2. Primary Pose Attachment

A manifest implementing this profile identifies the GeoPose value through one of two candidate attachment forms:

- **Inline primary-pose claim.** A standard `claims[]` entry with candidate `@type` `geopose.primary`, whose `subject` is the manifest subject or a dataset-record URI, whose candidate **`poseValue`** member (REQUIRED for this claim type) carries exactly one OGC GeoPose JSON data object, and whose `claimProof` verifies the issuer or measuring system when the pose is relied upon above Tier 0. Because the claim rides the signed payload, the inline form is restricted to **static** GeoPose targets — Basic-YPR, Basic-Quaternion, Advanced, Chain, Graph, and *recorded* (complete, bounded) time-series targets: a Stream target has no fixed byte representation to sign, is incoherent inside a signed claim, and MUST NOT be carried inline; streams use the pointer form.
- **Primary-pose pointer.** A standard `pointers[]` entry with candidate pointer type `um:geoPosePrimary` whose `target` URI resolves to an OGC GeoPose JSON data object or a signed container that includes one (any target form, including Stream). The pointer SHOULD carry a candidate **`digest`** member — a multihash over the exact GeoPose bytes the target resolves to (the multihash function MUST be SHA-256 for v0.5, the same discipline as O1.7 and `assetBinding.digest`, O12.4) — so the UM signature binds the manifest to the exact pose value, not just to a mutable location. A Stream target, having no fixed bytes, cannot be digest-bound; it remains a live reference whose currency is governed by O14.3.

For dataset use, each record that has its own pose SHOULD have its own record URI and its own primary-pose claim or pointer. A dataset-level manifest MAY also carry a collection-level GeoPose

value only when it describes the dataset as a whole; it MUST NOT be treated as the pose of every member record unless the dataset profile states that rule explicitly.

O14.3. Provenance, Time, and Currency

A GeoPose value can be observed, computed, imported, or asserted. The manifest SHOULD state the issuer or measuring system in the ordinary claim issuer/proof path, SHOULD bind the value to capture/measurement provenance when available, and SHOULD record the timestamp or validity interval supplied by the GeoPose payload or the surrounding claim. When current pose is required for a governed action, the evaluator MUST compose the manifest TTL, `statusRef`, consent windows, the current policy-state profile (O2.6) where implemented, and any domain-specific freshness policy before relying on the pose.

If the GeoPose value governs a represented-person asset (O12), an agentic action, a route-scoped disclosure (O13), or another consent-gated use, the pose is just another governed datum: it is projected only when the viewer/verifier/system/use case is allowed or needs to see it, and the receipt records whether the pose was accepted, withheld, stale, unverifiable, or unprocessed.

O14.4. Honest Limits

This profile makes GeoPose a first-class primary pose value inside the Universal Manifest lifecycle. It does not certify that an OGC GeoPose payload is itself valid unless an OGC GeoPose validator or attester has done so; it does not convert between reference frames; it does not resolve ambiguity in external frame definitions; it does not make a stale pose current; it does not prove the pose corresponds to the depicted asset without separate binding/provenance evidence; and it does not authorize location disclosure without an applicable consent or policy basis. Missing, unverifiable, stale, or withheld GeoPose data MUST be recorded honestly and MUST NOT be inferred.

✧ ✧ ✧

O15. Signature Profile B: JCS + ES256

Registry identifier: `um:profile:signature:jcs-es256`, in the `signature` category (O6).
Status: normative in v0.5 and recognized by this candidate's `schema.json`; OPTIONAL for evaluators to implement. Signature Profile A (Base §1.6) remains the default and the mandatory-to-

implement baseline for every evaluator. Profile B is additive: it never replaces Profile A, and it changes nothing about Profile A.

Signature Profile B is the second signature suite Universal Manifest recognizes, and it is here for one reason: **hardware key custody**. The safest place for a signing key is inside a device security chip, and the chips that ship on the platforms UM targets hold NIST P-256 keys and only P-256 keys — Apple's Secure Enclave, Android Keystore with StrongBox, and the TPM 2.0 PC Client platform profile [TPM2]. An Ed25519 key can be *wrapped* by those custodians but never *resident* inside them. Under Profile A alone, then, an implementer on those platforms must choose between following the specification and keeping the signing key in the chip, and the `hardware-bound` rung of the assurance ladder (EXT-T1 §T1.3.3) is unreachable on that hardware by construction. Profile B removes that choice: the key that signs is the key that cannot leave the chip.

Profile B also rehearses the machinery the named post-quantum migration (EXT-T2 §T2.3) will need. It is the first suite added under the governed-and-open suite mechanism (Base §1.6), and it is added by the ordinary registration step, not by a special case.

O15.1. Cryptographic Suite

Profile identity is the pair (`"ES256"`, `"JCS-RFC8785"`), read exactly as Base §1.6.5 defines profile identity.

- **Curve** — NIST P-256, also named secp256r1 and prime256v1 (three names, one curve) [FIPS186]. Its security level is approximately 128 bits, the same class as Profile A's Ed25519. This profile says "P-256 (secp256r1)" once and P-256 thereafter.
- **Signature algorithm** — ECDSA over P-256 with SHA-256. `signature.algorithm` MUST be `"ES256"`, the IANA JOSE name for this suite [RFC7518]. SHA-256 is intrinsic to the suite; there is no separate hash member.
- **Canonicalization** — `signature.canonicalization` MUST be `"JCS-RFC8785"`, identical to Profile A. One canonicalization pipeline serves both profiles; the signature primitive is the only variable.
- **Signature encoding** — `signature.value` MUST be the base64url encoding, unpadded (Base §1.6.1), of the **64-byte raw `R || S` concatenation** (the JOSE and WebCrypto convention). A DER/X9.62-encoded signature value MUST fail verification. This has to be normative because platforms disagree at the API boundary: Apple's Security framework emits DER and an implementation MUST convert it to raw before setting `signature.value` (CryptoKit's `P256.Signing.ECDSASignature` exposes `rawRepresentation` directly), while

WebCrypto and Android Keystore emit raw already. Left unpinned, the same hardware key would produce values two conformant verifiers disagree about.

O15.2. ECDSA Discipline: what this profile deliberately does not require

Two rules that ECDSA deployments elsewhere commonly adopt are **excluded** from this profile, on purpose and normatively. Both exclusions exist because a hardware custodian cannot honor them, and a rule no secure chip can satisfy would make every secure chip non-conformant — defeating the profile's entire purpose.

- **No deterministic-nonce requirement.** Deterministic ECDSA [RFC6979] MUST NOT be required of a signer. Secure Enclave, StrongBox, and TPM signers generate the per-signature nonce internally and expose no control over it. A signer using a software P-256 key SHOULD use RFC 6979 or a cryptographically secure random source; a hardware signer is conformant as it is. **Consequence, stated so no implementer is surprised by it:** re-signing identical bytes yields a different `signature.value` each time. Signature bytes are not a stable identifier. A manifest is identified by its `@id` (Base §1.2.2), never by its signature value, and no evaluator behavior in this specification keys off signature bytes.
- **No low-S rule.** A verifier MUST accept any mathematically valid `(r, s)` and MUST NOT reject a signature for carrying a high `s`. JOSE requires no low-S normalization and the hardware custodians do not guarantee it. The malleability corollary — where `(r, s)` verifies, `(r, n − s)` verifies too, so a signature has two valid encodings — is named here so it is not mistaken for a defect to be patched: because nothing in Universal Manifest keys off signature bytes, a second valid encoding of the same signature confers no capability, correlates nothing, and replays nothing that the first encoding did not already permit. An implementer MUST NOT add a Bitcoin-style low-S rejection; it rejects valid hardware signatures.

Neither exclusion is a relaxation of Profile A, which is unaffected: Ed25519 signatures are deterministic by construction, and the question does not arise there.

O15.3. Signature Shape and Signing Input

The `signature` object carries the same members as Profile A (Base §1.6.2) at the same requirement levels: `algorithm`, `canonicalization`, `keyRef`, and `value` REQUIRED; `publicKeySpkiB64`, `created`, `statusRef`, and `revocationCursor` OPTIONAL, with unchanged semantics. Profile B adds no member, removes none, and adds no term to the versioned context document.

The signing input is identical to Profile A's (Base §1.6.3). To compute a Profile B signature: (1) start with the complete JSON-LD production of the manifest; (2) remove the `signature` property entirely, and remove `presentationProof` and `postQuantumSignature` when present; (3) canonicalize the remainder with JCS (RFC 8785), producing a UTF-8 byte sequence; (4) compute the ECDSA-P256-SHA256 signature over those bytes; (5) set the `signature` property. The I-JSON constraint and the duplicate-member-name rejection of Base §1.6.3 apply unchanged, as does the per-production signing rule (O1.5): a Profile B signature is bound to the bytes of one production, so a re-encoded manifest MUST be re-signed.

Because the input is byte-identical, a holder can produce a Profile A and a Profile B signature over the same manifest content without a second canonicalization path, and an evaluator supporting both profiles shares one implementation of everything except the verification primitive.

O15.4. Key Representation

- **`signature.keyRef`** — semantics unchanged from Profile A (Base §1.6.2): a URI reference to the verification key material, a DID URL recommended. For a P-256 `did:key`, the multicodec is `p256-pub` (`0x1200`), which yields identifiers with the `zDn` prefix rather than Ed25519's `z6Mk` (for example, `did:key:zDn…#keys-1`). `did:web`, `did:jwk`, and HTTPS references work unchanged.
- **`signature.publicKeySpkiB64`** — the same member, a different payload: base64 of the SubjectPublicKeyInfo DER for `id-ecPublicKey` (OID 1.2.840.10045.2.1) with namedCurve prime256v1 (OID 1.2.840.10045.3.1.7), carrying the uncompressed EC point [RFC5480]. That is a 65-byte point inside a 91-byte SPKI, against Ed25519's 44-byte SPKI.
- **SPKI-algorithm consistency (REQUIRED).** SubjectPublicKeyInfo is algorithm-tagged, which is what lets one member carry both profiles without ambiguity. An evaluator MUST check that the SPKI's algorithm identifier matches `signature.algorithm`. A manifest declaring `"ES256"` while carrying an Ed25519 SPKI, or the reverse, is a profile/key mismatch: the evaluator MUST reject it, recording `signatureCheck: "invalid"`. This check is what keeps the shared member honest.
- **`keyRef` ↔ `publicKeySpkiB64` consistency and the key-resolution ladder** — carried over from Profile A verbatim (Base §1.6.4), including the key-substitution security note and the offline path that caps `keyRef`-derived identity assurance at Tier 0 with `keyRefResolution: "unresolved"`. The byte-identity comparison is performed after converting the resolved key to its SPKI DER encoding, equivalently over the raw 65-byte uncompressed P-256 point, exactly as Profile A compares raw 32-byte Ed25519 key values.

O15.5. Evaluator Checklist

An evaluator implementing Profile B MUST, in order:

1. Confirm the document is a v0.x Universal Manifest (required members present, `@type` includes `um:Manifest`), preserving unknown members before verification (Base §3.1.1).
2. Enforce TTL and clock skew exactly as under Profile A (Base §3.1.2).
3. Confirm profile support from the `algorithm` + `canonicalization` pair (Base §1.6.5): require `"ES256"` **and** `"JCS-RFC8785"` together.
4. Obtain the P-256 public key through the unchanged resolution ladder, and enforce the SPKI-algorithm check and the `keyRef` consistency check (O15.4).
5. Recompute the signing input (Base §1.6.3).
6. Verify the ECDSA-P256-SHA256 signature over the canonical bytes against the 64-byte raw `R || S` value: accept any mathematically valid `(r, s)` (O15.2); reject a DER-encoded value (O15.1). On failure the manifest MUST be rejected for use, and MAY be retained for debugging.
7. Apply the revocation-aware verification extension unchanged, if claimed (Base §1.6.6).

An evaluator that does **not** implement Profile B MUST reject a Profile-B-signed manifest and record `signatureCheck: "unsupported-profile"`, and MUST NOT reinterpret the pair under Profile A (Base §1.6.5). That rejection is correct behavior, not a defect: an unrecognized suite fails closed in both directions.

O15.6. Custody and Assurance Class

Profile B is the suite a hardware-resident key can use. It is not, by itself, evidence that the key *is* hardware-resident. **A signature proves possession, not residency.** Residency is a separate evidence question, answered by platform key attestation material (an Android Keystore attestation chain, Apple App Attest material, or a TPM attestation [WEBAUTHN], [TPM2]) carried alongside the manifest as claim material and verified through the ordinary claim-proof path (EXT-T1 §T1.5). A signature profile must not hard-depend on vendor attestation infrastructure, so this profile does not; a dedicated device-key-attestation claim type is candidate work for a future version, and its absence blocks nothing here.

The honest-labeling rule follows from that separation, and it governs the assurance ladder `software` < `hardware-uv` < `hardware-bound` (EXT-T1 §T1.3.3):

- Only a hardware-resident, non-extractable signing key may satisfy `requiredAssuranceClass: "hardware-bound"`. An evaluator requiring evidence for that rung takes it from attestation material, never from the signature suite alone. A Profile-B signature by itself does not establish `hardware-bound`.
- A hardware key gated by biometric or PIN user verification satisfies `hardware-uv` semantics for key *use*.
- A software P-256 key is **software-class** — including a software key **wrapped** at rest under a hardware key and unwrapped into process memory for signing. Hardware protection of that storage does not establish hardware custody of the signing operation, and such a software key MUST NOT be presented as satisfying `hardware-uv` or `hardware-bound`. A serialized hardware-wrapped handle to a non-exportable key that signs inside the security component is a different custody mechanism; its assurance class still depends on the evidence validated under EXT-T1 §T1.3.3. The word "wrapped" alone establishes neither mechanism nor assurance class. Wrapping remains permitted at-rest hygiene, and a receipt must state the custody actually established.

The remote-evidence ceiling and correlating-metadata caution in O9.2 apply equally to Profile B: device-local observations, holder-signed platform state and hardware-wrapped storage alone do not establish remote authenticator provenance. Apply EXT-T1 §T1.3.3 only to evidence actually validated. Verified hardware-backed user verification and verified device-bound provenance are separate findings; neither a holder assertion nor a signature-suite choice supplies the other.

`requiredTrustTier` is untouched by any of this. Trust tiers are orthogonal to the signature suite, and their raise-only semantics (Base §6.4.5) are unchanged. The interplay is with the assurance-class ladder alone, where Profile B is what makes `hardware-bound` satisfiable on the hardware people actually carry.

O15.7. Adoption Posture and Honest Limits

Profile A is the universal floor. Every evaluator MUST implement it; an evaluator that implements only Profile A is fully conformant and always will be. Profile B is an optional additive claim (O11), exactly like every other optional module in this document, and a deployment names it in its conformance claim (O11.1).

A holder MUST NOT assume Profile B acceptance. Where a relying party's capability is unknown, the holder issues that manifest under Profile A from a key the identity controls, or accepts the fail-closed rejection. Manifests are short-lived, per-issuance capsules that already mint a fresh `@id` each time, so **re-issuance per audience is the negotiation mechanism this version**

defines: no signature array, no new wire member, and no negotiation round trip is introduced. A declaration by which each party advertises the suites it can verify would remove that guesswork in bilateral exchange; it is candidate convenience work for a future version, deliberately not part of this profile.

One identity spans both suites. A subject's Apple, Android, or TPM device signs under Profile B while another device or a software key signs under Profile A, and the identity is the subject DID plus its enrolled device keys — not any one key. Where those keys are of different schemes, **reciprocal cross-signatures are the binding mechanism**: each key signs a statement naming the other, and the pair of signatures is verified under each key's own suite (EXT-T2 §T2.1 records the boundary — the Tier-2 zero-knowledge binding profiles assume same-scheme key material and do not cover cross-suite binding).

Honest limits. This profile does not make P-256 preferable to Ed25519 where both are available; Ed25519 remains the default, and a deployment with no hardware-custody driver has no reason to adopt Profile B. It does not certify hardware residency (above). It does not change the signing input, the canonicalization, the receipt surface, the tier semantics, or any term in the versioned context document. It does not make ECDSA misuse-resistant: a software implementation with a weak nonce source leaks its private key, which is exactly why the profile's home is hardware and why software signers are told to use RFC 6979 or a cryptographically secure random source. And it does not oblige any existing evaluator to do anything: a Profile-A-only deployment rejects Profile B manifests fail-closed and remains conformant.



Appendix OA. Manifest Class Registry Snapshot PREVIEW

This appendix is informative. It snapshots the manifest classes (Base §1.0.1) currently surfaced by integration profiles, to illustrate the polymorphic-envelope model. It does **not** bind conformance; the authoritative, versioned registry is maintained as a standalone document under the profile registration mechanism (O6). Discriminating members are the members whose presence identifies the class.

Manifest class	Discriminating members	Purpose
Identity capsule	`claims[]` with identity claim types; `holderBinding`	Portable identity and credential presentation.
Consent record	`consents[]` with `um:Consent` entries	Records permission grants governing facet use.

Manifest class	Discriminating members	Purpose
Device-capability descriptor	`devices[]` with `deviceCapability`	Advertises session-scoped device capabilities.
Device-attestation record	`devices[]` with `deviceAttestation`	Conveys manufacturer-signed hardware provenance.
Receipt	`@type` includes `um:Receipt`; `seq`/`prevHash`	Signed, chainable evaluation/audit record.
Cross-DID binding	`claims[]` with `identity.crossDidBinding`	Asserts common control of multiple DIDs.
Agent-delegation capsule	`pointers[]` with `um:agentDelegation`; `actorState`	Declares delegated session authority and the operating actor.
Encrypted-facet carrier	`facets[]` with `encryptionProfile`	Carries sealed, recipient-scoped payloads.
Category/service descriptor	`claims[]` with `operatorIdentity` / `categoryAttestation` types	Describes a service with attested category trust.
Bilateral-session record	`@type` `um:BilateralSession`; `exchangeId`	Correlates a two-party manifest exchange.
Asset-bound consent capsule (candidate, O12)	`subject` = asset URI; representation facet + represented-person consent entry	Carries the consent state of the person represented inside a portable asset.
GeoPose-bearing asset or dataset record (candidate, O14)	`claims[]` with `geopose.primary` or `pointers[]` with `um:geoPosePrimary`	Carries or references an OGC GeoPose primary pose value for the subject or record.

The snapshot is illustrative and non-exhaustive; integration profiles (e.g., the RP1 integration profile) surface additional classes. A manifest may belong to more than one class at once; the normative rule is in Base §1.0.1. The bilateral-session-record row anticipates a manifest that *records* a session: the session object of O4.1 is a protocol-layer object, not itself a manifest, so a session-record manifest would carry the common envelope members with `@type` including both `um:Manifest` and `um:BilateralSession`; its class profile is later work.

References

Cited in this profile (full descriptors in the published artifact): **[DID-CORE]**, **[INFRA]**, **[JSON-LD]**, **[CBOR-LD]**, **[RFC8949]** (CBOR), **[RFC3339]**, **[RFC9162]** (Certificate Transparency 2.0),

[MULTIFORMATS], **[VC-STATUS]** (Bitstring Status List), **[C2PA]** (Content Provenance and Authenticity), **[OGC-GEOPOSE]**, **[eIDAS2]**, **[OPENXR]**, **[WEBAUTHN]**, **[TPM2]**, **[RFC9591]** (FROST), **[RFC7518]** (JSON Web Algorithms — the `ES256` suite name), **[RFC6979]** (Deterministic ECDSA — cited to exclude, O15.2), **[RFC5480]** (Elliptic Curve SubjectPublicKeyInfo), **[FIPS186]** (Digital Signature Standard — the P-256 curve).



Companion to the Base specification. For worked examples, see the Cookbook.