OMATrust Proof Specification

Proof taxonomy of the OMATrust architecture

1. Executive Summary

This OMATrust Proof Specification (“Specification”) defines the canonical proof framework used across the OMATrust ecosystem. It mainly standardizes how issuers of an attestation can convey trust in the attestation (for example that a user was a client of a service they are reviewing). However, it is also used in other parts of OMATrust such as proving ownership in the Identity Registry. It is referenced by the OMATrust Identity Specification (“Identity Specification”) as well as the OMATrust Reputation Specification (“Reputation Specification”).

OMATrust Proofs are designed to be modular and composable. A Proof is a structured object that either (a) carries cryptographic evidence directly (a “Proof Object”), or (b) references evidence embedded in an external location. The framework separates what kind of evidence is being presented from where it is stored and from who is required to verify it.

The Specification defines a taxonomy of evidence that is dictated by identifier capability. Some identifiers are signer-capable (e.g., crypto wallets) and can produce arbitrary signature proofs; for these, evidence security derives from the signature (such as JWS) and may be stored anywhere. Other identifiers are non-signers (e.g., social media account handles); for these, evidence must be placed in locations that prove control of the identity (such as a verified profile field or a domain-controlled .well-known path). The Proof Object in this case is a pointer to that trusted location.

The Specification also defines Proof Purposes and their relationship to risk tiers and anti-spam economics. A Proof Purpose declares why the Proof exists (e.g., shared control of an identity vs. a commercial interaction). Certain mechanisms create Proofs differently depending on the Proof Purpose.

2. Scope

2.1 In-Scope

This document defines the requirements for Proofs within OMATrust. It is intended for implementers building reputation systems, identity registries, and clients in the broader OMATrust ecosystem.

2.2 Out-of-Scope

This specification does not define:

- How the Proofs are integrated into OMATrust systems such as reputation schemas and identity registries.
- Attestation schemas.
- Identity Registry object models and state transitions.
- Trust scoring / ranking / interpretation.

These are handled by other OMATrust specification documents or are out of scope of OMATrust.

3. References

OMATrust Whitepaper: https://github.com/oma3dao/omatrust-docs/blob/main/whitepaper.md
OMATrust Identity Registry Specification:
OMATrust Reputation Specification:
DID Specification: https://www.w3.org/TR/did-core
DID Spec Registries: https://www.w3.org/TR/did-spec-registries/
JCS Canonicalization Scheme: https://datatracker.ietf.org/doc/rfc8785/
CAIP-2 Standard: https://chainagnostic.org/CAIPs/caip-2

4. Definitions

4.1 Abbreviations

In the present document, the following abbreviations apply:

DID	Decentralized Identifier
DNS	Domain Name System
HTTP	Hypertext Transfer Protocol
HTTPS	Hypertext Transfer Protocol Secure
ID	Identifier
URL	Uniform Resource Locator
URI	Uniform Resource Identifier

Relevant External Specification Terminology Sections

DID Spec	https://www.w3.org/TR/did-core/#terminology

The following applies to the specification:

- “string” means a UTF-8 string
- “object” means a string in JSON format
- [] is meant to signify an array of whatever is inside the brackets.

4.2 Definitions

The following definitions are used within the present document.

Term	Definition
Service	Internet services such as APIs, applications, websites, and agents.
Client	Software that queries OMATrust to obtain information on a Service.
Verifier	Same as Client, but more commonly used in W3C DID.
Attester	An entity that issues credentials as attestations in OMATrust.
Issuer	Same as Attester, but more commonly used in W3C DID.
Attestation	A structured, signed statement conforming to an OMATrust schema, used to record claims about identity, usage, or trust.
Proof	Data in the form of a JSON object, that proves a particular relationship.
Proof Object	The native proof that is embedded in a Proof.
Proof Purpose	What a Proof is used for.
Proof Type	A label given to a specific Proof with a defined schema.
PoP	Proof of Possession.
x402	A bidirectional, payment-linked protocol used to exchange signed messages between client and server, capable of generating mutually verifiable Proof Objects.
Service Signing Key	A cryptographic key authorized to sign for a specific service DID.
Signing Algorithm	The cryptographic algorithm used to produce signatures (e.g., eip712, ed25519, secp256k1).
Authorization Set	The set of public keys that a service has published or attested as authorized to sign proofs.

Term	Definition
JCS	IETF RFC-8785 for canonicalization of JSON objects.
EAS	Ethereum Attestation Service (attest.org).

In addition to the above definitions all OMA3 specifications use requirements language as described in the OMA3 working group process.

5. Specification

Proofs are objects with defined structure that give verifiers the evidence they need to trust a claim. For example, if a verifier is reading a User Review attestation, how does the verifier know the attester was actually a user of the service being reviewed?

OMATrust requires Proofs because every major component of the ecosystem—Identity Registry, Reputation, and future service-verification systems—depends on reliable evidence of control, consent, or interaction. The Identity Registry must confirm that a controller truly owns or operates the identifier they are binding. The Reputation Framework must ensure that a reviewer actually interacted with a service or that an attestation corresponds to a real action.

To support these needs across heterogeneous identifiers, chains, and transports, OMATrust defines a unified, cross-chain proof framework. This framework standardizes how a Proof is structured, how it expresses its purpose, where its evidence is located, and how verifiers determine its validity. Proofs may be generated by cryptographic signers, by non-signing identifiers that rely on trusted locations, or by deterministic onchain actions. Each Proof therefore declares its **`proofType`**, its **`proofPurpose`**, and optionally its evidence location.

5.1 Proof Taxonomy

Proofs are categorized against several different dimensions, including:

- Identifier Capability: Is the issuer of the Proof capable of cryptographic signing?
- Evidence Location: Where the Proof Object is stored.
- Proof Purpose: What the Proof is proving
- Proof Type: How the Proof is constructed and verified.

5.1.1 Identifier Capability

Identifiers used in OMATrust Proofs fall into one of two capability classes. The capability of an identifier determines what kinds of proof mechanisms it is allowed to use, and what constitutes valid evidence of control for that identifier.

5.1.1.1 Signer-Capable Identifiers

An identifier is signer-capable if it has, or is bound to, at least one cryptographic key that can produce verifiable digital signatures. Examples include:

- EVM wallet addresses (secp256k1 EOAs)
- Contract wallets that expose onchain signature validation (e.g., EIP-1271)
- DID methods with associated key material (did:key, did:web with JWKs, etc.)
- Any identifier whose controller can produce a valid pop-jws or pop-eip712 Proof

5.1.1.2 Non-Signer Identifiers

An identifier is non-signer if it has no associated cryptographic key capable of producing a signature. These identifiers rely on externally controlled or platform-controlled locations to provide evidence of control. Examples non-signer identifiers include:

- Social handles (@username on social platforms)
- DNS names not bound to a DID Document with keys
- Platform-assigned identifiers with no signing capability
- Application IDs whose control is defined by platform metadata rather than keys

Requirements:

- Non-signer identifiers MUST use proof mechanisms that rely on trusted evidence locations, as defined by the binding for that identifier type (e.g., profile bio fields, **`.well-known`** paths, registry-controlled metadata).
- Non-signer identifiers MUST NOT use signature-based **`proofTypes`** (e.g., **`pop-jws`**, **`pop-eip712`**).
- Verifiers MUST treat the trust anchor (location control) as the basis for control, not a signature.

5.1.2 Evidence Location

Evidence for a Proof may appear directly inside the Proof Object or be retrieved from an external location. OMATrust defines evidence location categories to describe where a verifier must look when validating a Proof. These categories are conceptual and do not correspond to any serialized field. Instead, the **`proofType`** implicitly determines which evidence location category applies, based on the structure and fields of the Proof.

The following evidence locations are defined:

5.1.2.1 Inline Evidence

The complete evidence is embedded directly inside the Proof Object. Examples include:

- a JWS contained directly in the Proof
- an EIP-712 signature embedded as bytes
- an x402 receipt object included inline

For non-signer identifiers, inline evidence is only acceptable if there is a binding attestation that delegates an authorized signing authority for that identifier (e.g.- a **`did:pkh`** wallet identifier).

5.1.2.2 URL Evidence

URL evidence refers to proof mechanisms where the verifier retrieves evidence from a URL referenced by the Proof. This category encompasses both arbitrary URLs and standardized .well-known paths. Examples include:

- a JSON artifact at https://example.com/proof.json
- a signed receipt hosted at a service endpoint
- a DID Document at a .well-known path such as https://example.com/.well-known/did.json

Normative notes:

- Signature Proof Types MAY use any URL since authenticity derives from the cryptographic signature contained within the referenced content.
- For non-signature Proof Types, URL evidence is only authoritative if the binding explicitly defines the URL pattern as a trusted location for that identifier type. This includes .well-known paths as well as social media post URLs.
- Verifiers MUST use HTTPS when retrieving URL evidence unless a binding explicitly authorizes alternative retrieval mechanisms.

5.1.2.3 Transaction Evidence

Evidence is obtained from a blockchain transaction referenced by the Proof. Examples:

- deterministic tx-encoded-value Proofs
- memo-field or data-field embedded artifacts

Normative notes:

- Used primarily by Proof Types that rely on onchain side-effects.
- For non-signer identifiers, transaction-based Proofs MAY be allowed if there is a binding that defines the transaction sender as authoritative for control.

5.1.3 Proof Purpose

Every OMATrust Proof MUST declare a **`proofPurpose`**, which specifies why the Proof exists and what level of assurance it is intended to provide. The **`proofPurpose`** determines the required strength of the evidence, the acceptable Proof Types, and any economic or replay-prevention parameters associated with the Proof.

A **`proofPurpose`** is therefore a semantic contract between the Proof producer and verifier. Verifiers MUST apply the validation rules associated with the declared purpose and MUST reject Proofs whose purpose is missing, unsupported, or inconsistent with the expected verification context.

The following purposes are defined in this specification:

5.1.3.1 **`shared-control`**

Used for identity binding, controller verification, registry operations, and other cases where misbinding would undermine trust.

5.1.3.2 **`commercial-tx`**

Indicates a lower-friction Proof suitable for commercial interactions, usage confirmations, or lightweight reputation events. The emphasis is on usability and low cost, not on demonstrating control of high-value identifiers.

5.1.4 Proof Type

Every OMATrust Proof MUST declare a **`proofType`**, which identifies the verification algorithm and evidence interpretation rules that the verifier MUST apply. Unlike **`proofPurpose`**, which defines why a Proof exists, the **`proofType`** defines how the Proof is validated.

A **`proofType`** determines:

- how evidence is encoded,
- where evidence is expected to be located (inline, URL, transaction, etc.),
- which fields are required,
- which cryptographic operations the verifier performs,
- how the canonical seed is constructed (if applicable),
- and how replay protection is enforced for that mechanism.

A **`proofType`** does not define:

- the semantic meaning of the Proof (handled by **`proofPurpose`**),
- the assurance tier (also handled by **`proofPurpose`**),
- or the trust anchor for non-signer identifiers (defined by binding attestations).

Relationship Between **`proofType`** and **`proofPurpose`**:

- A **`proofType`** MAY be used with multiple purposes, depending on verifier policy.
- A **`proofType`** MUST declare which identifier capabilities it supports (signer-capable, non-signer, or both).
- A **`proofType`** MAY impose minimum or maximum purpose tiers (e.g., some types might be unsuitable for shared-control).
- A verifier MUST validate the **`proofType`** according to the rules of the declared **`proofPurpose`**.

5.2 Proof Object Parameters

The top-level OMATrust Proof is a standard JSON object with common fields to convey information about the embedded naive Proof Object such as **`proofType`**. OMATrust does not dictate a universal serialized JSON structure for the Proof Object itself. Different proof mechanisms—such as JWS, EIP-712 typed data, x402 receipts, or onchain transaction references—each use their own native container with their own field names and signing rules.

To ensure Proof Objects have the required functionality, OMATrust defines a set of semantic parameters that every Proof Object MUST convey, regardless of how those parameters are encoded or derived. These parameters form the conceptual Proof Object envelope: a verifier-level abstraction that defines what information must be obtained from the native Proof Object. Each **`proofType`** specification defines precisely how the verifier extracts or derives these parameters from the native **`proofObject`**. These are defined in Section 5.3.

The parameters of an OMATrust Proof Object are:

Parameter	Description
Subject	An identifier that represents the provider of the Proof and is the **`subject`** in the higher-level claim being made (e.g.- an attestation).
Controller	The identifier demonstrating shared control or an authoritative relationship with the subject, as required for the **`proofPurpose.`**
Counterparty	Subject or Controller.
Nonce	A challenge binding used for replay resistance. Required only by Proof Types that incorporate challenge/response semantics.

Timestamp	A creation timestamp (e.g., JWS iat, x402 receipt time, or block time). Required only for Proof Types that rely on temporal validation.
Signer	The DID that signs the Proof or delivers the credential. Does not map to an object field, but defines how the Proof is constructed.
Attester	The DID/wallet signing the attestation which includes the Proof. Not included in the Proof, but MAY use the same private key as either the Subject or Controller.
proofPurpose	See Section 4.2 and Section 5.3.

IMPORTANT: The interpretation of the Subject and Controller parameters depends on the higher-level attestation in which the Proof is embedded. A Proof does not stand alone: it is always consumed by an attestation whose schema determines which identifier is the "Subject" of the overall claim and which identifier plays the "Controller" role. The Proof must therefore express Subject and Controller values in a way that matches the attestation's semantics. For example, in a **`LinkedID`** attestation, the Subject of the Proof must correspond to the attestation **`subject`**, and the Controller must correspond to the **`linkedId`** field. In contrast, in independent proofs not associated with an attestation, the controller is typically the signer proving control over the Subject. This contextual dependency is normative: verifiers MUST interpret the Proof parameters according to the attestation schema that references the Proof.

The following subsections give instructive examples for the Subject and Controller parameters.

5.2.1 Delegated smart-contract authorization

Use case: A smart-contract controlling wallet delegates authority to a different wallet.

- Subject: DID of the smart-contract's controlling wallet
- Controller: DID or the wallet being granted delegation
- Signer: subject

"Controller wallet is authorized to act on behalf of Subject wallet for contract X."

5.2.2 DID ↔ handle linking

Use case: A platform handle account declares shared control with a wallet.

- Subject: DID of the handle (did:handle:x.com:alice)
- Controller: DID or the wallet being granted delegation
- Signer: Subject

"Controller wallet is authorized to act on behalf of the handle represented by the Subject DID."

5.2.3 Commercial transaction receipt

Use case: An x402 server sends the client a receipt of a completed transaction.

- Subject: the DID of the x402 server (e.g.- **`did:web`**)
- Controller: the DID of the client's wallet that paid the server
- Timestamp: Unix timestamp when the receipt was issued
- Signer: Subject

"Controller wallet paid for a service using x402 and received the service."

5.2.4 Service usage receipt

Use case: A service issues a proof-of-usage receipt (e.g., via x402) that can later be included in a reputation attestation.

- Subject: DID of the service (e.g., did:web:service.com)
- Controller: DID of the client wallet consuming the service
- Timestamp: Unix timestamp when the receipt was issued
- Signer: Subject (the service), using the signing key that controls the wallet that the client sent funds to.

"Controller wallet is a confirmed client of the Subject DID."

5.2.5 Handle ↔ handle linking

Use case: A centralized platform confirms that one of its handle accounts is associated with another handle.

- Subject: DID of the source handle (e.g., did:handle:x.com:alice)
- Controller: DID of the sink handle being linked to the source handle.
- Signer: None. Trust is given by placing the Proof Object at a URL that can only be controlled by the Subject (and/or the platform controlling the handle).

"Source handle confirms that it is under common control with the sink handle."

Note: a LinkedID attestation will need two Proofs (see OMATrust Reputation Specification).

5.3 Proof Types

Proofs in OMATrust use a two-tier structure. At the top level, a small, protocol-defined wrapper provides a consistent entry point for tooling and verifiers. The wrapper is intentionally minimal: it

exists to identify the proof and carry any OMATrust-level context that may be needed for routing, indexing, or display. The actual cryptographic or evidentiary artifact is carried in the wrapper's **`proofObject`** field, which holds the native Proof Object for the indicated **`proofType`**.

The wrapper does not impose a shared native schema across Proof Types. Instead, **`proofObject`** is interpreted according to **`proofType`** and may be either (a) a native serialized blob (for example, a compact JWS string for **`pop-jws`**) or (b) a structured JSON object whose fields are defined in the relevant **`proofType`** section. This keeps the wire format truthful to each proof ecosystem while still giving developers a uniform way to detect and parse Proofs.

Most wrapper fields are optional and are only required when a given **`proofType`** needs them to be expressed out-of-band. The only universally required field is **`proofType`**, which determines how **`proofObject`** is parsed and verified. Each **`proofType`** section below specifies: (1) the native **`proofObject`** format, (2) which wrapper fields are required or allowed for that type, and (3) how any included wrapper fields are cryptographically bound to the **`proofObject`**, aligning with the Common Proof Parameters defined in §5.2.

Some Proof Types defined in this section support EIP-712–based signatures (**`proofFormat`** = **`eip712`**). EIP-712 signatures differ from JWS signatures in that they cryptographically bind not only the message values, but also the typed schema under which those values are interpreted.

For EIP-712–based Proof Types:

- The canonical **`types`** and **`primaryType`** definitions are specified normatively in this document for each Proof Type.
- These definitions MUST NOT be included in transmitted Proof Objects.
- Signers MUST construct the EIP-712 signing digest using **`domain`**, **`message`**, and the canonical **`types`** and **`primaryType`** as defined by this specification.
- Verifiers MUST obtain the corresponding **`types`** and **`primaryType`** definitions from this specification in order to reconstruct the signing digest.

Conceptually, EIP-712 Proof Objects parallel JWS Proof Objects as follows:

- The EIP-712 **`domain`** provides signing context and replay protection, analogous to a JWS **`header`**.
- The EIP-712 **`message`** contains the signed payload data, analogous to a JWS **`payload`**.
- The EIP-712 **`signature`** provides cryptographic integrity, analogous to a JWS **`signature`**.

In x402 protocol messages, EIP-712–signed artifacts are transmitted as an object containing **`{ format, payload, signature }`**, where **`payload`** is the EIP-712 **`message`** values and **`signature`** is the hex-encoded signature. When constructing an OMATrust Proof Object for **`proofFormat = eip712`**, implementations MUST map **`payload`** to the EIP-712 **`message`**

field, copy the hex **`signature`** verbatim, and supply the canonical EIP-712 **`domain`** defined for this Proof Type in this specification.

While the EIP-712 schema (**`types`** and **`primaryType`**) does not appear on the wire, it is implicitly included in the **`signature`** through the EIP-712 hashing rules. As a result, any change to a canonical EIP-712 schema constitutes a breaking change and MUST be accompanied by explicit versioning.

Section numbers are non-normative and may change. Implementations and referencing specifications MUST rely on the proofType identifier rather than section numbering.

5.3.1 Proof Wrapper

All Proofs in OMATrust are expressed as a top-level JSON wrapper plus a **`proofType`**-specific native **`proofObject`**. The wrapper provides a consistent container for routing and interoperability, while allowing each **`proofType`** to retain its native wire format. Only a small number of wrapper fields are defined at this layer; most are optional and only become required for certain **`proofType`** that need them. The wrapper fields are as follows:

Field	Req	Format	Description
proofType	Y	string	Enum: **`pop-eip712`**, **`pop-jws`**, **`tx-encoded-value`**, **`x402-receipt`**, **`evidence-pointer`**, **`x402-offer`**.
proofObject	Y	string or object	Native proof object for **`proofType`**. MAY be embedded (string/object) or a pointer object that identifies an external artifact.
proofPurpose	N	string	The context of the Proof to prevent replay attacks. Enum: **`shared-control`** (for Binding/Linking), **`commercial-tx`** (for Reviews).
version	N	int	Default is 1
issuedAt	N	int	Unix timestamp
expiresAt	N	int	

Only **`proofType`** and **`proofObject`** are universally required. Each Proof Type section below specifies which wrapper fields are REQUIRED, OPTIONAL, or DISALLOWED for that Proof Type. Any wrapper field that contributes to semantic interpretation for a given Proof Type (at minimum **`proofPurpose`** and **`subject`**, and any time or issuer fields the proof relies on)

MUST be cryptographically bound to the native **`proofObject`** as described in that Proof Type's binding rules. If such a field is absent, verifiers SHOULD discount trust in the Proof.

All Proof Objects MUST conform to the **`proof`** definition in the Common Schema (Appendix B).

5.3.2 Standard Key Proof (**`pop-jws`**)

For private keys (e.g., PGP, SSH, Ed25519), a Proof MAY be provided as a compact JSON Web Signature (JWS) conforming to RFC 7800 and RFC 9449 (DPoP). The native Proof Object for this Proof Type is the compact JWS itself; when pop-jws is used, the wrapper's proofObject is a string containing the JWS.

5.3.2.1 Mechanism

The Subject (see Section 5.2) parameter's key signs a JWS. The JWS header includes the Subject key's public key as a JSON Web Key (JWK). The JWS payload includes the claims defined below. The JWS expresses that the Subject key approves or authorizes the **aud** entity for the declared purpose.

5.3.2.2 Claims

The JWS payload consists of the following fields:

Field	Req	Format	Parameter	Note
iss	Y	string	Subject	
aud	Y	string	Controller	
proofPurpose	Y	string	proofPurpose	
iat	N	int	issuedAt	
exp	N	int	expiresAt	
jti	N	string	Nonce	

The JWS header consists of the following fields:

Field	Req	Format	Meaning
alg	Y	string	JWS algorithm

jwk	Y	object	Public key of the subject signer in JWK form.

5.3.2.2 Verification and Use

The verifier MUST validate the JWS signature against the JWK embedded in the JWS header and MUST confirm that the required claims match the proofPurpose and proofType, including subject-as-signer.

For Web2 platforms (e.g., X, Facebook, etc.), a **`pop-jws`** proof MAY be pasted into a social platform URL controlled by the authorized entity (**`aud`**) to show that the Subject signer approves that Web2 identifier. In this use case, the **`pop-jws`** **`proofObject`** MUST include **`aud`** as **`did:handle`**.

5.3.3 Ethereum Wallet Proof (**`pop-eip712`**)

For Ethereum-compatible keys, a Proof MAY be provided as an EIP-712 typed-data signature. This Proof Type is used when the Subject controls an onchain account and needs to produce a portable, replay-resistant proof bound to a specific purpose and audience. When **`pop-eip712`** is used, the wrapper's **`proofObject`** is a JSON object whose fields are defined below.

5.3.3.1 Mechanism

The **`subject`** signs an EIP-712 typed-data message with the private key corresponding to the subject's onchain address. The signed typed-data message binds the Subject signer, an authorized/audience identifier, and a declared OMATrust Proof Purpose into a single verifiable statement.

5.3.3.2 Claims

The wallet signs a typed data structure conforming to the OMATrust Proof format. The native proof object for **`pop-eip712`** is a structured EIP-712 bundle:

Field	Req	Format	Note
domain	Y	object	See below
message	Y	object	See below
signature	Y	string	

The EIP-712 **`domain`** object contains the following fields:

Field	Req	Format	Value
name	Y	string	“OMATrustProof”
version	Y	string	“1”
chainId	Y	int	EIP-155 chain ID of the signer’s chain

The EIP-712 **`message`** object contains the following fields, which have names that are self-explanatory so that users know exactly what they are signing with their wallet. The **`message`** object is an instance of the canonical OMATrustProof EIP-712 type defined below, populated with the values being asserted by the signer.

Field	Req	Format	Parameter or Description
signer	Y	address	Subject
authorizedEntity	Y	string	Controller
signingPurpose	Y	string	proofPurpose
creationTimestamp	N	uint64	issuedAt
expirationTimestamp	N	uint64	expiresAt
randomValue	N	bytes32	jti/nonce
statement	Y	string	Human-readable safety statement shown by wallets. MUST state this is not a transaction or asset approval.

`message` example:

```json
{
  "signer": "0xabc...",
  "authorizedEntity": "did:web:example.com",
  "signingPurpose": "commercial-tx",
  "creationTimestamp": 1730000042,
  "expirationTimestamp": 0,
  "randomValue": "0x0000...",
  "statement": "This is not a transaction or asset approval."
```

```
}
```

For Proof Type **`pop-eip712`**, OMATrust defines a single canonical EIP-712 schema. Implementations MUST use this schema when requesting signatures from wallets and when verifying proofs. To avoid redundant encoding and reduce transport size, stored or transmitted **`pop-eip712`** proofs MUST NOT include the **`types`** or **`primaryType`** fields; verifiers MUST obtain the canonical schema from this specification.

When a client requests a **`pop-eip712`** signature from a wallet (e.g., via **`eth_signTypedData_v4`**), the client MUST supply the canonical **`types`** and **`primaryType`** shown below in the signing request. Any signature produced over a non-canonical schema is non-compliant and MUST be rejected by verifiers.

`creationTimestamp`, **`expirationTimestamp`**, and **`randomValue`** are OPTIONAL fields in message. If unused, clients SHOULD set:

- **`creationTimestamp`** = 0
- **`expirationTimestamp`** = 0
- **`randomValue`** = 0x00…00 (32 bytes)

Canonical EIP-712 Types (**`pop-eip712`**)

JSON

```
{
  "primaryType": "OMATrustProof",
  "types": {
    "EIP712Domain": [
      { "name": "name", "type": "string" },
      { "name": "version", "type": "string" },
      { "name": "chainId", "type": "uint256" }
    ],
    "OMATrustProof": [
      { "name": "signer", "type": "address" },
      { "name": "authorizedEntity", "type": "string" },
      { "name": "signingPurpose", "type": "string" },
```

```
      { "name": "creationTimestamp", "type": "uint64" },
      { "name": "expirationTimestamp", "type": "uint64" },
      { "name": "randomValue", "type": "bytes32" },
      { "name": "statement", "type": "string" }
    ]
  }
}
```

5.3.4 x402 Receipt Proof (**`x402-receipt`**)

The **`x402-receipt`** Proof Type provides cryptographic evidence that a service acknowledged payment and attested to successful delivery to a client. It is the canonical high-confidence commercial interaction proof for User Reviews and other commercial interaction attestations when the reviewed service implements the x402 Signed Offer and Service Receipt Extension.

5.3.4.1 **`x402-receipt`** Format

This Proof Type directly embeds the Service Receipt defined in the x402 Signed Offer and Service Receipt Extension. The receipt supports two signature formats: JWS (for web-native signing) and EIP-712 (for EVM wallet signing). This Proof wrapper MUST contain the following fields:

Field	Req	Format/Value
proofType	Y	String that MUST be **`x402-receipt`**
proofPurpose	Y	String that MUST be **`commercial-tx`**
proofObject	Y	Native receipt object as defined by the selected proofFormat

5.3.4.2 Receipt **`proofObject`**

`proofObject` is the receipt object returned in the **`extensions`** field of an x402 Settlement Response message.

```JSON
extensions["offer-receipt"].info.receipt
```

5.3.4.3 Verification Logic

Verifiers MUST validate the contained receipt according to the x402 Signed Offer and Service Receipt Extension specification referenced above. This includes signature verification, issuer authorization, and validation that the receipt semantically asserts successful service delivery according to the x402 Signed Offer and Service Receipt Extension specification.

OMATrust does not redefine x402 receipt verification semantics.

5.3.5 Evidence Pointer (**`evidence-pointer`**)

The **`evidence-pointer`** Proof Type references a publicly accessible *evidence artifact* hosted at a URL. The evidence artifact supports verification of an attestation by providing off-chain public proof of control or linkage. The Proof Object is a pointer; the external resource it resolves to is the *evidence artifact*.

The evidence artifact MAY take one of the following forms:

A. Embedded Cryptographic Proof: an embedded cryptographic proof object (**`pop-jws`** or **`pop-eip712`**) that can be extracted and verified independently; or
B. Handle-Link Statement: a clear, human-readable statement that references another social identifier, enabling linkage without embedding cryptographic material.

The specific attestation types and workflows that use **`evidence-pointer`**, including the required hosting party, signing party, and semantic interpretation of evidence modes, are defined in the Reputation Specification.

5.3.5.1 **`evidence-pointer`** Format

For this Proof Type, the OMATrust proof wrapper MUST contain the following fields:

Field	Req	Value
proofType	Y	String that MUST be **`evidence-pointer`**
proofPurpose	Y	String that MUST be **`shared-control`**
proofObject	Y	Object (see below)

The **`proofObject`** for **`evidence-pointer`** MUST be a JSON object with the following fields:

Field	Req	Value

url	Y	String: Public URL containing the evidence artifact.

No cryptographic material is embedded directly in the pointer; the url resolves to the evidence artifact that supports verification.

5.3.5.2 Evidence String Format

For **`evidence-pointer`** Proofs where the evidence is hosted on a social platform (profile bio, status, pinned post, etc.), and a cryptographic proof is not possible or desired, the evidence string MUST follow this format:

`v=1;controller=<DID>`

Example:

`v=1;controller=did:pkh:eip155:66238:0x0cB8859f11CC52Ca73256D86C37843E5a84b256B`

Parsing Rules:

- Fields are separated by semicolons (;) or whitespace
- Field order is not significant
- Unknown fields SHOULD be ignored for forward compatibility

Verification: For Handle-Link Statement evidence (human-readable text), the verifier MUST:

1. Parse the evidence string and extract key-value pairs using semicolon (;) or whitespace as delimiters.
2. Validate version: Confirm **`v=1`** is present.
3. Extract **`controller`**: Parse the controller value, which MUST be a valid DID (e.g., did:pkh:eip155:66238:0x..., did:handle:twitter:alice).
4. Confirm controller binding: The extracted controller MUST match:
    a. The attestation's **`linkedId`** (for Linked Identifier attestations), or
    b. The attestation's **`keyId`** (for Key Binding attestations)
5. Confirm subject control: The URL where the evidence is hosted MUST be controlled by the attestation's **`subject`**. For example:
    a. A Twitter profile URL is controlled by the did:handle:twitter:* subject
    b. A GitHub profile URL is controlled by the did:handle:github:* subject

If any of these checks fail, the evidence MUST be rejected.

Note: This format is intentionally identical to the DNS TXT record format used for did:web verification, enabling consistent tooling across verification methods.

5.3.5.3 Verification Procedure

Verifiers MUST validate a **`evidence-pointer`** proof using the following procedure:

1. Fetch evidence artifact: Retrieve the resource at proofObject.url. The content MUST be publicly accessible without authentication.
2. Determine evidence mode: The fetched content MUST satisfy one of the evidence modes defined above (Embedded Cryptographic Proof or Handle-Link Statement).
3. Verify evidence:
    a. If the evidence contains an embedded cryptographic proof, the verifier MUST extract and validate it according to its Proof Type specification in this section (§5.3.2–§5.3.3).
    b. If the evidence is a handle-link statement, the verifier MUST follow the verification procedure described in 5.3.5.2.
4. Accept/reject: The proof is valid only if the evidence artifact is retrievable and satisfies the verification requirements for its evidence mode and attestation context.

5.3.6 Encoded Value Transactions (**`tx-encoded-value`**)

The **`tx-encoded-value`** Proof Type is a deterministic “micro-challenge” proof for signers that cannot produce arbitrary signatures but can send a native-value transaction on a supported ledger. Instead of signing a message, the subject proves intent by transferring a precisely computed **`Amount`** of the chain’s native asset from the subject-controlled address to the Controller address specified in the attestation context. The **`Amount`** is derived from the proof context so that verifiers can recompute it independently and match it to an onchain transfer.

This Proof Type is defined once here and is used by both the Identity Specification and the Reputation Specification. The Identity Specification defines when **`tx-encoded-value`** is an acceptable proof mechanism for identity-level attestations, while the Reputation Specification uses the same Proof Object for reputation-layer attestations. In all cases, the Proof is chain-agnostic in semantics: the same inputs yield the same expected **`Amount`** across transports, with only per-chain constants varying by binding.

Because native ledgers differ in fee tokens and denominations, the **`Amount`** algorithm relies on chain-specific **`BASE`** and **`RANGE`** parameters. Those parameters are not specified inline here; each transport binding MUST supply them, and the current recommended values are listed in Appendix A referenced below.

5.3.6.1 **`tx-encoded-value`** Format

For this Proof Type, the OMATrust proof wrapper MUST contain the following fields:

Field	Req	Value
proofType	Y	String that MUST be **`tx-encoded-value`**
proofPurpose	Y	String
proofObject	Y	Object (see below)

The proofObject for tx-encoded-value MUST be a JSON object with the following fields:

Field	Req	Format	
chainId	Y	string	
txHash	Y	string	Transaction identifier on the specified chain

Requirements:

- For **`tx-encoded-value proofs`**, the native-asset transaction MUST occur on the chain identified by **`proofObject.chainId`**, which is the chain of the Subject being proven. Issuers and verifiers MUST search for and validate the transaction on that chain only.
- The transaction MUST be addressed to the Controller address, which MUST be a native address on the VM of **`proofObject.chainId`**.
- If the Controller address is not natively addressable on the VM of **`proofObject.chainId`**, this Proof Type is NOT valid.

Binding note. Transport bindings MAY require additional locator fields (e.g., **`sender`**, **`recipient`**, block height, or signature index) if the chain cannot be reliably searched by **`txHash`** alone. If a binding requires such fields, they MUST be added to **`proofObject`** by that binding and verifiers MUST use them as specified there.

5.3.6.2 Calculating the Amount

The required transaction **`Amount`** is deterministically derived based on the Proof Purpose. This ensures that (1) a transaction intended for one purpose cannot be replayed for another, and (2) higher-risk purposes incur higher spam-prevention costs.

```
Amount = BASE(proofPurpose,chainId) +
         (U256(H(Seed) mod RANGE(proofPurpose,chainId)))
```

All terms and operations in the **`Amount`** formula—including hash width, integer interpretation, and modulo semantics—are defined in the tables and requirements of this section. **`chainId`** is used only to select chain-specific **BASE** and **RANGE** constants and MUST NOT be included in **`Seed`**.

Term	Definition
Native unit	The smallest indivisible unit of the ledger's native fee token (e.g., wei, lamport, uatom).
proofPurpose	The declared **`proofPurpose`** for which the transaction is offered as proof (e.g., **`shared-control`**, **`commercial-tx`**).
chainId	The CAIP-2 identifier of the target chain on which the transaction was executed.
BASE(proofPurpose, chainId)	A base cost tier for the given Proof Purpose on the specific ledger identified by chainId, expressed in that ledger's native smallest unit.
RANGE(proofPurpose, chainId)	An entropy window for the given purpose on the specific ledger identified by chainId, expressed in native smallest units. RANGE SHOULD be a fixed fraction of BASE unless overridden by the binding (default: 10% of BASE).
Seed	Opaque bytes derived from the proof context, encoded canonically.
H(x)	A cryptographic hash function producing at least 256 bits of output. The default is SHA-256. Transports MAY specify an alternative hash only if it is widely deployed on that ledger and yields equivalent security properties. EVM is keccak256.
U256(b)	Interpret the left-most 32 bytes of **b** as an unsigned 256-bit integer in big-endian order

- **BASE** sets the minimum cost for a purpose on a given chain (spam resistance).
- **RANGE** adds a small, deterministic offset so the exact **`Amount`** is unique to the proof context (replay resistance).
- **`Seed`** ensures that uniqueness is bound to the Subject, Controller, and Proof Purpose, while remaining chain-agnostic.

Requirements:

- **`Amount`** is denominated in the chain's native smallest unit and MUST be representable without rounding. If a ledger does not support exact native-unit transfers, that binding MUST NOT support **`tx-encoded-value`**.
- **BASE** is a per-chain, per-purpose constant expressed in the chain's native unit. Bindings MUST define these values for every supported purpose.
- **`RANGE`** MUST be > 0.
- Unless explicitly overridden by a transport binding, **`RANGE`** MUST be computed as **`floor(BASE(proofPurpose,chainId)/10`** where **`RANGE`** > 0. Any binding that overrides this default MUST publish an explicit RANGE value for the affected purpose/chain.
- **`purpose`** MUST be taken from the wrapper's **`proofPurpose`** for this Proof. A verifier MUST recompute **`Amount`** using that declared purpose.
- The entropy term (**`U256(H(Seed)) mod RANGE`**) MUST be computed exactly as written. Implementations MUST NOT truncate or reinterpret the hash width or integer endianness.
- Because **`Seed`** includes both **`proofPurpose`** and Controller (see §5.3.6.3), an **`Amount`** derived for one purpose or Controller MUST NOT validate a proof for another.

`Seed` is calculated as per Section 5.3.6.3 below

5.3.6.3 Canonical **`Seed`** Construction

`Seed` is an opaque byte string derived deterministically from the proof context. It provides domain separation and binds the **`Amount`** to the Subject, Controller, and Proof Purpose in a chain-agnostic way.

Verifiers MUST construct **`Seed`** as follows:

1. Construct the canonical JSON object: Verifiers MUST construct a JSON object with exactly the following string fields and no others:

JSON

```
{
  "domain": "OMATrust:Amount:v1",
  "subjectDidHash": "<didHash(subject)>",
  "counterpartyIdHash": "<didHash(counterpartyId)>",
  "proofPurpose": "<proofPurpose>"
}
```

2. Field semantics:
    a. **`domain`** is a fixed domain-separation string for **`Amount`** derivation.
    b. **`subjectDidHash`** is **`didHash(subject)`** as defined in the Identity Specification.
    c. **`counterpartyIdHash`** is the didHash of the Counterparty wallet DID.
    d. **`proofPurpose`** is the declared Proof Purpose for this Proof.
3. Canonicalize with JCS: The JSON object MUST be canonicalized using JCS (JSON Canonicalization Scheme). No other canonicalization or packing rule is permitted.
4. Encode to bytes: **`Seed`** bytes are the UTF-8 encoding of the JCS-canonicalized JSON string.

Requirements:

- Bindings MUST NOT substitute VM-specific packing/concatenation (e.g., ABI packing) for the JCS object above. Doing so would break cross-chain determinism and produce a non-compliant **`Amount`**.
- All four fields MUST be treated as strings at the JSON layer, even if their underlying values originate as addresses, enums, or hashes.
- Any mismatch in field names, field order (post-JCS), field types, or domain string MUST cause verification failure.

Implementations MUST NOT use ad-hoc concatenation such as abi.encodePacked(...) to construct **`Seed`**. Such constructions are not JCS-canonical and will diverge from the **`Amount`** expected by compliant verifiers.

Each transport binding MUST define **`BASE`** and **`RANGE`** per **`proofPurpose`** for its chain. Recommended values are listed in Appendix A.

5.3.6.4 Verification Rules

A verifier validating a **`tx-encoded-value`** proof MUST perform all of the following steps. Failure of any step MUST cause the proof to be rejected.

1. Resolve the transaction: Using **`proofObject.chainId`** and **`proofObject.txHash`** (plus any binding-required locator fields), the verifier MUST locate the referenced native-value transaction on the specified ledger.
2. Confirm sender: The verifier MUST confirm that the transaction sender corresponds to the subject-controlled address for this attestation context, as defined by the applicable transport binding.
3. Confirm recipient: The verifier MUST confirm that the transaction recipient equals the Controller address specified in the attestation context, unless the transport binding explicitly defines an alternate sink/recipient rule for this Proof Type.

4. Select applicable parameter schedule: Transport bindings MUST publish time- or height-indexed schedules for BASE(proofPurpose, chainId) (and any explicit RANGE overrides). The verifier MUST determine the applicable schedule based on the transaction's onchain block timestamp or block height, as specified by the binding, and MUST use the schedule that was in effect at the time the transaction was executed.
5. Recompute **`Seed`**: The verifier MUST reconstruct **`Seed`** exactly as specified in §5.3.6.3, using:
    a. **`subjectDidHash = didHash(subject)`**
    b. **`counterpartyIdHash = didHash(counterpartyId)`**
    c. **`proofPurpose`** from the Proof wrapper
    d. the fixed domain string.
6. Recompute **`Amount`**: The verifier MUST recompute the expected **`Amount`** using the formula in §5.3.6.2, with:
    a. the declared **`proofPurpose`**,
    b. the binding's **`BASE`** (and derived or explicit **`RANGE`**) selected in Step 4, and
    c. the **`Seed`** from Step 5.
7. Compare transferred value: The verifier MUST accept the proof only if the onchain transferred native value equals the recomputed **`Amount`** exactly in native smallest units, with no rounding or tolerance.
8. Final acceptance: If Steps 1–7 succeed, the **`tx-encoded-value`** Proof is valid for the attestation context. Otherwise it is invalid.

Additional notes:

- Verifiers MUST treat **`chainId`** as a selector for chain-specific constants only; it MUST NOT be hashed into **`Seed`**.
- If a binding uses block height rather than timestamp for schedule selection, verifiers MUST use that height rule consistently and MUST NOT substitute timestamp logic.
- Verifiers MUST reject Proofs that reference a chain or purpose for which no binding schedule exists at the transaction time.

5.3.7 Onchain Interaction Proof (**`tx-interaction`**)

The **`tx-interaction`** Proof Type provides verifiable evidence that a reviewer interacted with a smart-contract-based service by submitting an onchain transaction directly to the service's contract. This Proof Type functions as the onchain analogue of a Web2 "API call" confirmation and is suitable only for **`commercial-tx`** purposes.

This Proof Type does not guarantee service delivery—only that a transaction occurred from the reviewer to the service contract. For delivery assurance, systems MUST prefer **`x402-receipt`**.

5.3.7.1 **`tx-interaction`** Format

The OMATrust proof wrapper MUST contain:

Field	Req	Value
proofType	Y	String that MUST be **`tx-interaction`**
proofPurpose	Y	String MUST equal **`commercial-tx`**
proofObject	Y	Object (see below)

The **`proofObject`** for **`tx-interaction`** MUST be a JSON object with the following fields:

Field	Req	Format	
chainId	Y	string	CAIP-2 chain identifier
txHash	Y	string	Transaction identifier on the specified chain

5.3.7.2 Verification Logic

A verifier validating an **`tx-interaction`** proof MUST perform all of the following steps:

1. Fetch the transaction- Retrieve the transaction object (**`tx`**) indexed by **`txHash`** from the ledger identified by **`chainId`**. If the transaction cannot be retrieved or the chain is unsupported, reject the proof.
2. Confirm the transaction succeeded- The transaction MUST have a success status (no revert, no failure).
3. Confirm sender binding- **`tx.from`** MUST correspond to the reviewer's identity defined by the attestation: **`tx.from == attester`**
4. Confirm subject binding- **`tx.to`** MUST correspond to the service contract being reviewed: **`tx.to == subject`** (Contract Address)
5. Confirm temporal validity- The block timestamp MUST satisfy verifier policy (e.g., must be ≤ the attestation's **`effectiveAt`**). The spec does not prescribe an exact policy but verifiers MUST enforce one.
6. Accept or Reject- The proof is valid only if ALL checks above pass.

5.3.8 x402 Offer Proof (**`x402-offer`**)

The **`x402-offer`** Proof Type provides cryptographic evidence that an x402 service committed to a specific set of commercial terms for a resource prior to payment. This Proof Type is used to

demonstrate that a service made a binding offer (or quote) for access to a resource under stated conditions.

An **`x402-offer`** Proof does not assert that payment was made or that service was delivered. It asserts only that the service presented and cryptographically signed an offer. For proof of completed service delivery, see **`x402-receipt`** (§5.3.4).

User Review attestations and other commercial interaction attestations MAY include both an **`x402-offer`** Proof and an **`x402-receipt`** Proof.

5.3.8.1 **`x402-offer`** Format

This Proof Type directly embeds a signed offer as defined in the x402 Signed Offers and Service Receipts Extension.

The OMATrust proof wrapper MUST contain the following fields:

Field	Req	Format/Value
proofType	Y	String that MUST be **`x402-offer`**
proofPurpose	Y	String that MUST be **`commercial-tx`**
proofObject	Y	Native offer object as defined by the selected **`proofFormat`**

Note that proofFormat is not needed because it is contained in the proofObject in accordance with the x402 Signed Offer and Service Receipt Extension.

5.3.8.2 Offer **`proofObject`**

`proofObject` is a signed offer object returned in the extensions field of an x402 Payment Requirements (402) response message.

```JSON
extensions["offer-receipt"].info.offers[*]
```

5.3.8.3 Verification Logic

Verifiers MUST validate the contained offer according to the x402 Signed Offers and Service Receipts Extension specification referenced above. This includes signature verification, issuer

authorization, and confirmation that the signed offer payload matches the advertised x402 payment requirements.

OMATrust does not redefine x402 offer verification semantics.

Change History

Version	Date	Comments
0.1	2025-11-30	Initial draft - Alfred Tom
0.2	2025-12-22	New extension-offer-and-receipt.md spec
0.21	2026-01-22	x402 Signed Offer and Service Receipt Extension clarifications

Appendix A

`tx-encoded-value` BASE and RANGE

CAIP-2 ID	units	**`Shared-control`** BASE	**`Commercial-tx`** BASE
eip155:6623 (OMAChain)	wei	1e16 wei (.01 OMA)	1e14 wei (.0001 OMA)
eip155:66238 (OMAChain Testnet)	wei	1e16 wei (.01 OMA)	1e14 wei (.0001 OMA)
eip155:1 (ETH Mainnet)	wei	1e14 wei (0.0001 ETH)	1e12 wei (0.000001 ETH)
eip155:11155111 (Sepolia)	wei	1e14 wei	1e12 wei
eip155:8453 (Base)	wei	1e14 wei	1e12 wei
solana:5eykt4UsFv8P8NJdTREpY1vzqKqZKvdp	lamport	1e6 lamports (0.001 SOL)	1 × 10^4 lamports (0.00001 SOL)
eip155:10 (Optimism)	wei	1e14 wei	1e12 wei
eip155:42161 (Arbitrum)	wei	1e14 wei	1e12 wei
eip155:137 (Polygon)	wei	1e14 wei	1e12 wei

These default values do not have an expiration date yet.