Workgroup:	Network Working Group
Internet-Draft:	draft-jurkovikj-collab-tunnel-01
Published:	14 November 2025
Intended Status:	Experimental
Expires:	18 May 2026
Author:	A. Jurkovikj

The Collaboration Tunnel Protocol (TCT)

Abstract

This document specifies the Collaboration Tunnel Protocol, an HTTP-based method for efficient, verifiable delivery of web content to automated agents. The protocol uses bidirectional URL discovery, template-invariant content fingerprinting with strong ETags, sitemap-first verification, and strict conditional request discipline to reduce bandwidth while preserving canonical URLs.

Status of This Memo

Copyright Notice

Table of Contents

1. Introduction

Automated agents (search engines, AI crawlers, archives, monitoring tools, aggregators) increasingly consume web content at scale. Fetching and parsing full HTML pages for machine consumption is often inefficient:

- Page weight is dominated by templates, navigation, ads, and scripts.
- Machine consumers typically need a stable textual representation of the core content.
- Many pages do not change frequently, but are re-fetched in full.

Various sites already expose JSON APIs or feeds, but:

- formats differ across sites, and
- use of HTTP validators is inconsistent.

The Collaboration Tunnel Protocol (TCT) defines a simple, interoperable profile on top of HTTP that:

- exposes a canonical JSON representation (M-URL) for selected resources;
- advertises C-URL/M-URL mappings and validators in a JSON sitemap (M-Sitemap);
- uses a single, well-defined strong ETag method for M-URLs; and
- enables "zero-fetch" behavior when content is unchanged.

TCT is intentionally conservative: it uses only existing HTTP mechanisms, is backward compatible with the Web, and does not define policy or licensing semantics.

1.1. Terminology

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "NOT RECOMMENDED", "MAY", and "OPTIONAL" in this document are to be interpreted as described in BCP 14 [RFC2119] [RFC8174] when, and only when, they appear in all capitals.

- C-URL:
    - The canonical, human-facing URL of a resource, typically an HTML document.
- M-URL:
    - The machine-facing URL providing the TCT JSON representation of that resource.
- M-Sitemap:
    - The JSON sitemap enumerating C-URL/M-URL pairs and associated validators.
- Representation:
    - As defined in [RFC9110]: the information in a payload, including representation metadata, that is subject to content negotiation.

Unless stated otherwise, "client" refers to an automated agent that is aware of TCT.

1.1.1. Problem Statement

Key inefficiencies in current automated consumption of web content include:

- repeated transfer of large HTML documents whose core content has not changed;
- lack of a standard, compact, semantics-focused representation for page-like resources;
- ad hoc usage or absence of validators (ETag, Last-Modified), hindering efficient revalidation;
- difficulty for agents to reason about change detection at scale using only HTML and XML sitemaps.

1.1.2. Goals and Non-Goals

TCT is designed to:

- reuse HTTP semantics ([RFC9110], [RFC9111]) rather than introduce new ones;
- provide a simple, deterministic JSON representation appropriate for:
    - search indexing,
    - content analysis,
    - retrieval-augmented generation (RAG),
    - archival and monitoring;
- define discovery and validation clearly enough for interoperable clients and servers.

TCT explicitly does NOT:

- define how content may or may not be used (policy, licensing, AI usage);
- change the semantics of resources at C-URLs;
- require new HTTP methods or status codes.

1.2. Architecture

TCT introduces three main elements per participating origin:

- C-URL:
    - The canonical resource URL for humans, often serving HTML.
- M-URL:
    - A URL providing a canonical JSON representation of the same logical resource (TCT JSON).
- M-Sitemap:
    - A JSON document listing C-URLs, M-URLs, and strong validators (`etag` values).

High-level flow (informative):

1. Client performs `GET /` at `https://example.com/`.
2. The origin root response includes:
    - `Link: </llm-pages.json>; rel="index"; type="application/json"`.
3. Client fetches `/llm-pages.json` (M-Sitemap).
4. For each item:
    - learns `(cUrl, mUrl, etag)`.
5. Client fetches `mUrl` as needed:
    - `GET mUrl`, with `If-None-Match` on subsequent checks.

6. Server responds with:
    - `200 OK` + JSON when changed;
    - `304 Not Modified` when unchanged.

TCT is additive and optional:

- Non-TCT clients ignore TCT artifacts.
- Servers can deploy TCT gradually alongside existing content and sitemaps.

1.3. Design Rationale and Relation to Existing Mechanisms

TCT is grounded in existing mechanisms and complements several related efforts:

1.3.1. Related Work

XML Sitemaps: - Widely deployed for URL discovery (search engines, crawlers). - Provide `<lastmod>` timestamps but do not define normative bindings between sitemap entries and HTTP validators (ETags) for endpoint representations. - TCT builds on this model by adding structured JSON representations and strong validator integration.

ResourceSync: - Developed by the Open Archives Initiative and collaborators for resource synchronization in digital libraries. - Provides resource lists, change lists, and capability documents. - Focus is synchronization and preservation; does not define a single, tightly integrated pattern combining: - per-resource JSON representation, - sitemap listing with validators, and - ETag-based zero-fetch semantics. - TCT addresses the specific use case of efficient web content delivery to automated agents.

AMP (Accelerated Mobile Pages): - Defines an HTML subset optimized for fast rendering on mobile devices. - Provides alternate representations via `<link rel="amphtml">`. - TCT provides JSON (not HTML) for machine consumption, targeting crawlers and content analysis rather than human browsing.

Custom JSON APIs: - Many sites expose custom JSON endpoints for content access. - Structures, field names, and validator usage vary widely across implementations. - TCT aims to standardize a minimal, interoperable profile suitable for broad adoption.

robots.txt: - Defines crawl directives and access policies. - TCT does not replace robots.txt; publishers MAY use both mechanisms: - robots.txt for crawl permissions and rate guidance, and - TCT for efficient content delivery.

RSS/Atom Feeds: - Provide syndication of updates and content excerpts. - Typically lack HTTP caching integration (per-item ETags, conditional requests). - TCT can be viewed as "RSS with disciplined HTTP caching semantics."

1.3.2. TCT Design Choices

Key design choices in TCT:

- Use only existing HTTP semantics:
  - GET, HEAD, 200, 304, ETag, Cache-Control, Link.
- Use JSON as the machine representation:
  - Simple, widely supported, easy to parse.
  - TCT is complementary to existing HTTP content-coding and dictionary-based compression mechanisms: it reduces representation size and refetch frequency at the application layer while remaining fully compatible with compression applied to M-URL responses.
- Define one strong ETag method:
  - Based on canonical JSON bytes of the M-URL payload.
  - Avoid ambiguous or dual hash methods.
- Keep policy and energy considerations out of the core:
  - Those can be specified separately as informational work.

The intent is to offer an interoperable profile that is:

- easy to implement,
- friendly to existing caches/CDNs, and
- precise enough for standardization.

1.4. Discovery

1.4.1. M-Sitemap Discovery

A publisher implementing TCT MUST expose an M-Sitemap and advertise it from the origin root resource.

When a client performs:

- `GET /` with `Host: example.com`

and receives a successful (2xx) response (either directly or after following redirects per [RFC9110]), that response:

- MUST include a `Link` header with:
  - `rel="index"`
  - `type="application/json"`
  - a target that is the M-Sitemap URL for this origin.

Example:

- `Link: </llm-sitemap.json>; rel="index"; type="application/json"`

In this specification, a Link header field with `rel="index"`, `type="application/json"`, and a target whose content matches Section 7 identifies the TCT M-Sitemap for that origin. Other uses of `rel="index"` remain valid and are outside the scope of this document.

Notes:

- `/llm-sitemap.json` is an example; any stable path MAY be used.
- If `/` redirects (e.g., `301` or `302` to `/en/` or `/index.html`), the `Link` header MUST appear on the final redirect target.
- Clients:
  - SHOULD follow redirects per [RFC9110] before checking for TCT support.
  - MUST discover the M-Sitemap via the `Link` header on the final response.
  - If no such `Link` is present, SHOULD assume TCT is not deployed and MUST NOT guess paths.
- If multiple `Link` headers with `rel="index"` and `type="application/json"` are present, clients MAY choose any or load all. This document does not define multi-sitemap selection semantics; large-site sharding is left to operational conventions (see Section 9.2).

This specification does not define or require any `/.well-known/` URI.

1.4.2. M-URL Discovery and Canonical Links

For each resource where an M-URL is provided:

- The C-URL response (typically HTML) SHOULD advertise the M-URL as an alternate JSON representation:

  Either via HTML:

  - `<link rel="alternate" type="application/json" href="https://example.com/post/llm.json">`

  Or HTTP:

  - `Link: <https://example.com/post/llm.json>; rel="alternate"; type="application/json"`

- The M-URL response for that resource MUST include a corresponding canonical link ([RFC6596]):
  - `Link: <https://example.com/post/>; rel="canonical"`

Note on M-URL paths: The choice of M-URL path (e.g., `/post/llm.json`, `/post/llm/`, `/post.json`, etc.) is not specified by this document. Publishers MAY choose any stable URL scheme that suits their architecture. Clients MUST discover M-URLs via advertised links (as shown above) and MUST NOT assume a fixed path pattern.

Examples of valid M-URL patterns: - `https://example.com/post/llm.json` (used in this document) - `https://example.com/post/llm/` (directory-style) - `https://example.com/post.json` (extension-based)

This bidirectional linkage allows clients to:

- verify that an M-URL is an alternate for the expected C-URL;
- detect misconfigurations when links are inconsistent; and
- ensure the `canonical_url` field in the M-URL JSON matches (or is consistent with) the URL advertised via `rel="canonical"` for the corresponding C-URL.

1.4.3. Template-Invariance

TCT's template-invariance property:

- Changes to HTML templates, CSS, or JavaScript at the C-URL SHOULD NOT require changes to the M-URL JSON, so long as the underlying resource content has not changed.

This is achieved by:

- treating M-URLs as distinct, canonical JSON representations of content; and
- computing strong ETags over the M-URL JSON only.

Template-invariance is NOT achieved by relaxing strong ETag semantics; for any given M-URL, identical strong ETags MUST imply byte-identical JSON bodies.

1.5. M-URL Representation

An M-URL is an HTTP resource that serves a JSON representation of a content resource suitable for machine consumption.

This specification defines the observable JSON representation at M-URLs; it does not constrain how servers derive these representations from their internal data models, templates, or storage.

1.5.1. Content-Type and Encoding

M-URL responses:

- MUST use:
  - `Content-Type: application/json; charset=utf-8`
- MUST be valid JSON as per [RFC8259].
- MUST use UTF-8 without BOM.

1.5.2. Required JSON Fields

The minimal M-URL JSON object for a resource MUST contain:

- `canonical_url` (string, REQUIRED)
  - The canonical human-facing URL (C-URL).
- `title` (string, REQUIRED)
  - A human-readable title for the resource.
- `content` (string, REQUIRED)
  - Core textual content:
    - plain text (no raw HTML markup);
    - SHOULD exclude purely template/boilerplate text.
- `hash` (string, REQUIRED)
  - A validator string of the form:
    - `sha256-` followed by 64 lowercase hex characters,
    - computed as specified in Section 6.2.
  - The `hash` field of an M-URL representation MUST equal the strong `ETag` value for that representation, excluding the HTTP quoting.

Additionally:

- `profile` (string, RECOMMENDED)
  - TCT profile identifier, e.g., `tct-1`.

Example (non-normative):

```
json { "profile": "tct-1", "canonical_url": "https://example.com/post/", "title":
"Article Title", "content": "Core article content...", "hash":
"sha256-2c26b46b68ffc68ff99b453c1d30413413422d706483bfa0f98a5e886266e7" }
```

1.5.3. Representation Stability

For an M-URL implementing this specification:

- For a given resource state, the JSON body MUST be deterministic.
- Any change to the JSON body bytes (including required or optional fields) MUST result in a different strong ETag (Section 6.2).
- Servers MUST NOT include per-request randomness (e.g., varying timestamps) in the TCT JSON representation.

1.6. Deterministic JSON and Strong ETags

1.6.1. Deterministic JSON Serialization

M-URL responses MUST use deterministic JSON serialization sufficient to support strong ETag semantics:

Implementations SHOULD use the JSON Canonicalization Scheme (JCS) defined in [RFC8785]. Implementations that do not use RFC8785 MUST define and implement a canonicalization algorithm that:

- produces a single, unique octet sequence for each abstract JSON value;
- applies stable ordering of object members at all levels (for example, lexicographic by code point);
- uses deterministic formatting of numbers; and
- does not introduce insignificant whitespace beyond what is necessary to delimit JSON tokens.

The same canonicalization algorithm MUST be used both when computing the ETag and when generating the response body.

1.6.2. Strong ETag Generation (Single Method)

TCT defines one mandatory method for computing ETags for M-URLs.

For an M-URL representation:

1. Construct the full JSON object representing the resource, including all fields except `hash`.
2. Canonicalize this JSON object to a UTF-8 byte sequence as per Section 6.1.
3. Compute the SHA-256 digest of these bytes.
4. Let F be the 64-character lowercase hex encoding of this digest.
5. Set:
   - HTTP `ETag` header:
     - `"sha256-$F"`
   - JSON `hash` field:
     - `sha256-$F$`
6. Canonicalize the final JSON object (now including `hash`) using the same algorithm defined in Section 6.1, and send that canonical form as the response body.

Implementations MUST NOT compute the ETag over one serialization of the JSON object and send a different, non-canonical serialization on the wire.

Requirements:

- ETags for M-URLs:
  - MUST be strong (no `W/` prefix).
  - MUST be quoted-strings as in [RFC9110].
- Two successful M-URL responses with the same strong ETag MUST have identical response body bytes.
- Any change in the response body bytes MUST cause the strong ETag to change.

1.6.3. Relationship to Template-Invariance

Strong ETags in TCT are validators for the M-URL JSON representation only.

Template-invariance is achieved structurally:

- M-URLs do not include HTML templates or layout.
- Changes to C-URL HTML that do not affect the M-URL JSON do not affect the strong ETag.
- For TCT-conformant M-URLs, strong ETags MUST be representation-based and MUST NOT vary per request in the absence of a change to the underlying JSON representation.

1.6.4. Canonical Text Normalization (Optional for Producers)

TCT does not require clients to normalize text or recompute hashes.

Use of this normalization algorithm is OPTIONAL and is not required for TCT conformance by either servers or clients. It defines an additional 'TCT text normalization' profile for implementations that choose to adopt it.

However, producers or validators that derive text fields (such as `content`) from upstream formats MAY implement the normalization algorithm defined here to ensure stable input before JSON canonicalization.

If an implementation claims conformance to this normalization, it:

- MUST follow the algorithm in this section; and
- MUST pass the test vectors in Appendix A.

Algorithm (summary):

Given input string S:

1. (Optional) HTML entity decoding:
   - If starting from HTML source, deterministically decode character references.
2. Unicode normalization:
   - Apply NFKC as defined in Unicode Standard Annex #15 (UAX15).

3. Case folding:
    - Apply Unicode case folding as defined in Unicode-CaseFolding.
    - Locale-dependent mappings MUST NOT be used.
4. Control characters:
    - Remove all Cc characters except:
        - U+0009 (TAB), U+000A (LF), U+000D (CR).
5. Whitespace collapsing:
    - Define set W:
        - U+0020, U+0009, U+000A, U+000D.
    - Optionally treat U+00A0 as whitespace, if done consistently.
    - Replace each maximal run of W (and optional NBSP) with a single U+0020 SPACE.
6. Trim:
    - Remove leading and trailing SPACE (U+0020).

The result is the normalized string N(S).

Details, examples, and conformance language are in Appendix A.

1.7. M-Sitemap Format and Semantics

1.7.1. M-Sitemap JSON Structure

The M-Sitemap is a JSON object that lists TCT-enabled resources.

Fields:

- `version` (integer, REQUIRED)
    - M-Sitemap format version. Initial value: 1.
- `profile` (string, RECOMMENDED)
    - Profile identifier, e.g., `tct-1`.
- `items` (array of objects, REQUIRED)
    - Each item:
        - `cUrl` (string, REQUIRED)
            - Canonical URL.
        - `mUrl` (string, REQUIRED)
            - M-URL of the JSON representation.

  - `etag` (string, REQUIRED)
    - Strong validator identifier, equal to the M-URL ETag value without quotes.
    - MUST be `sha256-` followed by the 64-hex digest used in Section 6.2.

Example:

`json { "version": 1, "profile": "tct-1", "items": [ { "cUrl": "https://example.com/post/", "mUrl": "https://example.com/post/llm.json", "etag": "sha256-2c26b46b68ffc68ff99b453c1d30413413422d706483bfa0f98a5e886266e7" } ] }`

1.7.2. HTTP Response for M-Sitemap

The M-Sitemap:

- MUST use:
  - `Content-Type: application/json; charset=utf-8`
- SHOULD use cache directives that encourage timely revalidation, for example:
  - `Cache-Control: max-age=0, must-revalidate`, or
  - a short max-age value appropriate to the site's update frequency.
- SHOULD include:
  - `Vary: Accept-Encoding`

1.7.3. Parity Semantics

Design intent:

- M-Sitemap `etag` values SHOULD match the current strong ETag values of their corresponding M-URLs.

Requirements:

- Publishers MUST compute `etag` values using the same algorithm as Section 6.2.
- Publishers SHOULD keep M-Sitemap `etag` values in sync with M-URL ETags.
- Transient mismatches (due to non-atomic updates, caches, or propagation delays) MAY occur.

Client behavior:

- Clients SHOULD treat M-Sitemap `etag` as a hint for change detection.
- Clients MAY compare `etag` with the M-URL ETag.
- Clients MUST NOT treat mismatches alone as protocol errors.
- In case of mismatch, clients SHOULD fall back to standard conditional requests on the M-URL and treat the M-Sitemap etag as an advisory signal only.

Strong ETag values for M-URLs and etag values in the M-Sitemap are scoped to the origin that serves them. Clients MUST NOT assume that identical validator values observed on different origins imply identical content.

1.8. Client Behavior

1.8.1. Zero-Fetch Optimization

A typical TCT-aware client:

1. Fetches `/` and discovers the M-Sitemap via `Link`.
2. Fetches the M-Sitemap.
3. For each item `(cUrl, mUrl, etag)`:
    - If it has a cached ETag for `mUrl`:
        - If cached ETag equals sitemap `etag`:
            - MAY skip fetching `mUrl` (zero-fetch).
        - Otherwise:
            - SHOULD issue conditional GET:
                - `GET mUrl`
                - `If-None-Match: "sha256-..."`.
    - If no cached ETag:
        - SHOULD fetch `mUrl` with GET.
        - MAY include `If-None-Match` using sitemap `etag` as hint.

This enables large reductions in redundant fetches.

This optimization is OPTIONAL. Clients that prefer strict HTTP cache validation MAY always perform a conditional GET with If-None-Match on M-URLs instead of relying solely on M-Sitemap etag hints.

1.8.2. Conditional Requests to M-URLs

M-URLs implementing TCT:

- MUST support `If-None-Match` as defined in [RFC9110].

Specifically:

- If `If-None-Match` matches the current strong ETag:
    - Respond with `304 Not Modified`, no body.
- Otherwise:
    - Respond with `200 OK` and the JSON representation.

If both `If-None-Match` and `If-Modified-Since` are present:

- `If-None-Match` MUST take precedence (per [RFC9110]).

Servers SHOULD send appropriate `Cache-Control` directives to encourage revalidation and safe caching.

1.8.3. Use of HEAD (Optional)

Clients MAY use `HEAD` on M-URLs as an optimization:

- If a HEAD response includes an `ETag` equal to the cached one, a GET may be skipped.
- If HEAD is unreliable, clients SHOULD fall back to conditional GET.

Conditional GET with `If-None-Match` SHOULD be considered the primary mechanism.

1.8.4. Client Validation (Optional)

For protocol correctness:

- Clients are not required to implement the text normalization algorithm or to recompute validators.
- Clients MAY treat ETags and sitemap `etag` values as opaque.

Clients MAY perform additional checks as desired, such as:

- verifying that an M-URL includes `rel="canonical"` pointing at the expected C-URL;
- checking M-Sitemap `etag` vs M-URL ETag parity;
- recomputing hashes using the published algorithms.

Such additional checks are implementation choices and are out of scope for TCT compliance.

1.9. Operational Considerations (Informative)

1.9.1. Deployment with CDNs and Proxies

- M-URLs and M-Sitemaps are ordinary cacheable JSON resources.
- CDNs and reverse proxies:
  - MAY cache them;
  - SHOULD honor strong ETags and 304 responses.
- Since ETags are based on uncompressed canonical JSON:
  - Downstream compression does not invalidate validators.

1.9.2. Large Sites and Sharding

For very large sites:

- Operators MAY create multiple M-Sitemaps (e.g., per section or date).

- A primary M-Sitemap can list additional ones by convention.
- This document does not standardize a JSON sitemap index; such work may be done separately.

A future revision of this specification or a separate document MAY define a JSON M-Sitemap index format, analogous to the XML Sitemap Index, to provide interoperable conventions for sharding and large-site discovery.

1.9.3. Error Handling

Recommended behavior:

- If the M-Sitemap is unavailable or invalid:
  - Treat TCT as temporarily unavailable.
- If an M-URL returns 4xx/5xx:
  - Follow normal HTTP semantics for retries/backoff.
- If the M-URL is persistently broken:
  - Clients MAY ignore it and rely on the C-URL HTML, if appropriate.

1.9.4. Backwards Compatibility

- TCT is purely additive:
  - It does not alter C-URL semantics.
  - Non-participating clients see no change.
- Operators can deploy TCT incrementally on selected resources.

1.10. Security Considerations

TCT builds directly on HTTP; most security considerations are inherited from [RFC9110] and [RFC9111].

Key points:

- HTTPS:
  - Publishers SHOULD serve M-URLs and M-Sitemaps over HTTPS.
  - Clients SHOULD validate TLS as usual.
- Integrity:
  - Strong ETags identify specific representations but do not authenticate servers.
  - For stronger guarantees, publishers MAY use:
    - `Content-Digest` headers ([RFC9530]); and/or
    - HTTP Message Signatures ([RFC9421]).

- Access control:
  - If a C-URL requires authentication, its corresponding M-URL SHOULD be similarly protected.
  - M-Sitemaps MUST NOT leak sensitive M-URLs or validators.
- Cache poisoning:
  - Correct use of ETag and Cache-Control mitigates stale or mixed content.
  - Clients MUST treat JSON as untrusted data and validate/sanitize as appropriate.
- Privacy:
  - As with any sitemap, listing URLs in an M-Sitemap can reveal site structure.
  - Publishers concerned about this SHOULD limit entries or restrict access.
- Origin trust:
  - Clients that consume M-Sitemaps and M-URLs inherently trust the origin in the same way they trust HTML pages or XML Sitemaps from that origin. A compromised or misconfigured origin can advertise incorrect mappings; this is not a new class of attack introduced by TCT.
- Intermediaries:
  - Deployments SHOULD ensure that CDNs and other intermediaries do not strip or rewrite strong ETags on M-URLs or M-Sitemaps, as doing so can interfere with correct validation and zero-fetch behavior.

1.11. IANA Considerations

This document has no IANA actions.

A future revision or separate document MAY define and register:

- a well-known URI for TCT M-Sitemaps; and/or
- a dedicated media type or `profile` parameter for TCT JSON.

Such registrations are intentionally out of scope for this version.

2. References

2.1. Normative References

[RFC2119] Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", BCP 14, RFC 2119, DOI 10.17487/RFC2119, March 1997, <https://www.rfc-editor.org/info/rfc2119>.

[RFC8174] Leiba, B., "Ambiguity of Uppercase vs Lowercase in RFC 2119 Key Words", BCP 14, RFC 8174, DOI 10.17487/RFC8174, May 2017, <https://www.rfc-editor.org/info/rfc8174>.

[RFC8259] Bray, T., Ed., "The JavaScript Object Notation (JSON) Data Interchange Format", STD 90, RFC 8259, DOI 10.17487/RFC8259, December 2017, <https://www.rfc-editor.org/info/rfc8259>.

[RFC8785] Rundgren, A., Jordan, B., and S. Erdtman, "JSON Canonicalization Scheme (JCS)", RFC 8785, DOI 10.17487/RFC8785, June 2020, <https://www.rfc-editor.org/info/rfc8785>.

[RFC9110] Fielding, R., Ed., Nottingham, M., Ed., and J. Reschke, Ed., "HTTP Semantics", STD 97, RFC 9110, DOI 10.17487/RFC9110, June 2022, <https://www.rfc-editor.org/info/rfc9110>.

[RFC9111] Fielding, R., Ed., Nottingham, M., Ed., and J. Reschke, Ed., "HTTP Caching", STD 98, RFC 9111, DOI 10.17487/RFC9111, June 2022, <https://www.rfc-editor.org/info/rfc9111>.

2.2. Informative References

[RFC6596] Ohye, M. and J. Kupke, "The Canonical Link Relation", RFC 6596, DOI 10.17487/RFC6596, April 2012, <https://www.rfc-editor.org/info/rfc6596>.

[RFC9421] Backman, A., Ed., Richer, J., Ed., and M. Sporny, "HTTP Message Signatures", RFC 9421, DOI 10.17487/RFC9421, February 2024, <https://www.rfc-editor.org/info/rfc9421>.

[RFC9530] Polli, R. and L. Pardue, "Digest Fields", RFC 9530, DOI 10.17487/RFC9530, February 2024, <https://www.rfc-editor.org/info/rfc9530>.

Appendix A. Appendix A. Normalization Test Vectors

This appendix provides test vectors for implementations that claim conformance to the normalization algorithm (Section 6.4).

Conformance Requirement: Implementations claiming normalization support MUST produce the outputs specified below for all test inputs.

Test Format: Each test shows: - Input string - Output after each normalization step - Final SHA-256 hash (computed over UTF-8 bytes of final output)

A.1. A.1. Basic ASCII

Test 1: Simple ASCII text - Input: `"Hello World"` - After step 1 (HTML decode): `"Hello World"` - After step 2 (NFKC): `"Hello World"` - After step 3 (casefold): `"hello world"` - After step 4 (control chars): `"hello world"` - After step 5 (whitespace): `"hello world"` - After step 6 (trim): `"hello world"` - SHA-256: `b94d27b9934d3e08a52e52d7da7dabfac484efe37a5380ee9088f7ace2efcde9`

Test 2: Leading/trailing whitespace - Input: `" Hello World "` - After step 1: `" Hello World "` - After step 2: `" Hello World "` - After step 3: `" hello world "` - After step 4: `" hello world "` - After step 5: `" hello world "` - After step 6: `"hello world"` - SHA-256: `b94d27b9934d3e08a52e52d7da7dabfac484efe37a5380ee9088f7ace2efcde9`

Test 3: Multiple spaces - Input: `"Hello World"` - After step 1: `"Hello World"` - After step 2: `"Hello World"` - After step 3: `"hello world"` - After step 4: `"hello world"` - After step 5: `"hello world"` - After step 6: `"hello world"` - SHA-256: `b94d27b9934d3e08a52e52d7da7dabfac484efe37a5380ee9088f7ace2efcde9`

A.1.1. A.2. HTML Entities

Test 4: Common HTML entities - Input: `"Hello &amp; goodbye"` - After step 1: `"Hello & goodbye"` - After step 2: `"Hello & goodbye"` - After step 3: `"hello & goodbye"` - After step 4: `"hello & goodbye"` - After step 5: `"hello & goodbye"` - After step 6: `"hello & goodbye"` - SHA-256: `da73536eaa9c427f3189de5b6371d798193e98f3c31df8bef710bba835e8c621`

Test 5: Angle brackets - Input: `"&lt;tag&gt;"` - After step 1: `"<tag>"` - After step 2: `"<tag>"` - After step 3: `"<tag>"` - After step 4: `"<tag>"` - After step 5: `"<tag>"` - After step 6: `"<tag>"` - SHA-256: `c81ef880af0fcfef49e1b45c3690a1666c47d9e064b7eaead2af09bb78884dcd`

Test 6: Quotes - Input: `"&quot;quoted&quot;"` - After step 1: `"\"quoted\""` - After step 2: `"\"quoted\""` - After step 3: `"\"quoted\""` - After step 4: `"\"quoted\""` - After step 5: `"\"quoted\""` - After step 6: `"\"quoted\""` - SHA-256: `272fca25899893eeb27b89583d5c81b8a4ac5af4d1e37e3909d879947303c1c5`

A.1.2. A.3. Unicode Normalization (NFKC)

Test 7: Composed vs decomposed é - Input (composed): `"Café"` (U+00E9) - After step 1: `"Café"` - After step 2 (NFKC): `"Café"` (normalized to composed form) - After step 3: `"café"` - After step 4: `"café"` - After step 5: `"café"` - After step 6: `"café"` - SHA-256: `850f7dc43910ff890f8879c0ed26fe697c93a067ad93a7d50f466a7028a9bf4e`

Test 7b: Decomposed form (should produce same result) - Input (decomposed): `"Cafe\u0301"` (e + combining acute) - After step 2 (NFKC): `"Café"` (normalized to composed) - Final result: Same as Test 7 - SHA-256: `850f7dc43910ff890f8879c0ed26fe697c93a067ad93a7d50f466a7028a9bf4e` (same as Test 7)

Test 8: Full-width characters - Input: `"ＨＥＬＬＯ"` (full-width Latin) - After step 1: `"ＨＥＬＬＯ"` - After step 2 (NFKC): `"HELLO"` (converted to half-width) - After step 3: `"hello"` - After step 4: `"hello"` - After step 5: `"hello"` - After step 6: `"hello"` - SHA-256: `2cf24dba5fb0a30e26e83b2ac5b9e29e1b161e5c1fa7425e73043362938b9824`

A.1.3. A.4. Case Folding Edge Cases

Test 9: German sharp S - Input: `"Straße"` - After step 1: `"Straße"` - After step 2: `"Straße"` - After step 3 (casefold): `"strasse"` (ß -> ss) - After step 4: `"strasse"` - After step 5: `"strasse"` - After step 6: `"strasse"` - SHA-256: `16d96952087774fee069b7585d3991b24d90c181c09b2129b4908c35baa7f0c0`

Test 10: Turkish İ (dotted capital I) - Input: `"İstanbul"` - After step 1: `"İstanbul"` - After step 2: `"İstanbul"` - After step 3 (casefold): `"i\u0307stanbul"` (locale-independent) - After step 4: `"i\u0307stanbul"` - After step 5: `"i\u0307stanbul"` - After step 6: `"i\u0307stanbul"` - SHA-256: `4a4df120f7d1f3c286f58651abfcec2aade892ace635f96f02b946c96e6e1f86`

A.1.4. A.5. Control Characters

Test 11: Embedded tab - Input: `"Hello\tWorld"` - After step 1: `"Hello\tWorld"` - After step 2: `"Hello\tWorld"` - After step 3: `"hello\tworld"` - After step 4: `"hello\tworld"` (tab preserved) - After step 5: `"hello world"` (tab -> space) - After step 6: `"hello world"` - SHA-256: `b94d27b9934d3e08a52e52d7da7dabfac484efe37a5380ee9088f7ace2efcde9`

Test 12: Embedded newline - Input: `"Hello\nWorld"` - After step 1: `"Hello\nWorld"` - After step 2: `"Hello\nWorld"` - After step 3: `"hello\nworld"` - After step 4: `"hello\nworld"` (newline preserved) - After step 5: `"hello world"` (newline -> space) - After step 6: `"hello world"` - SHA-256: `b94d27b9934d3e08a52e52d7da7dabfac484efe37a5380ee9088f7ace2efcde9`

Test 13: Control character (BEL) - Input: `"Hello\u0007World"` (U+0007 = BEL) - After step 1: `"Hello\u0007World"` - After step 2: `"Hello\u0007World"` - After step 3: `"hello\u0007world"` - After step 4: `"helloworld"` (control char removed) - After step 5: `"helloworld"` - After step 6: `"helloworld"` - SHA-256: `936a185caaa266bb9cbe981e9e05cb78cd732b0b3280eb944412bb6f8f8f07af`

A.1.5. A.6. Whitespace Edge Cases

Test 14: Non-breaking space (NBSP) - Input: `"Hello\u00A0World"` (U+00A0 = NBSP) - After step 1: `"Hello\u00A0World"` - After step 2: `"Hello\u00A0World"` - After step 3: `"hello\u00A0world"` - After step 4: `"hello\u00A0world"` - After step 5: `"hello world"` (NBSP -> space, if treating NBSP as whitespace) - After step 6: `"hello world"` - SHA-256: `b94d27b9934d3e08a52e52d7da7dabfac484efe37a5380ee9088f7ace2efcde9`

Note: Step 5 behavior for NBSP is implementation-defined per Section 6.4 ("Optionally treat U+00A0 as whitespace, if done consistently"). This test assumes NBSP is treated as whitespace.

Test 15: Mixed whitespace - Input: `"Hello \t\n World"` - After step 1: `"Hello \t\n World"` - After step 2: `"Hello \t\n World"` - After step 3: `"hello \t\n world"` - After step 4: `"hello \t\n world"` - After step 5: `"hello world"` - After step 6: `"hello world"` - SHA-256: `b94d27b9934d3e08a52e52d7da7dabfac484efe37a5380ee9088f7ace2efcde9`

A.1.6. A.7. Empty and Edge Cases

Test 16: Empty string - Input: `""` - After all steps: `""` - SHA-256: `e3b0c44298fc1c149afbf4c8996fb92427ae41e4649b934ca495991b7852b855`

Test 17: Whitespace only - Input: `"  "` - After step 1: `"  "` - After step 2: `"  "` - After step 3: `"  "` - After step 4: `"  "` - After step 5: `"  "` - After step 6: `""` (trimmed) - SHA-256: `e3b0c44298fc1c149afbf4c8996fb92427ae41e4649b934ca495991b7852b855`

Test 18: Single character - Input: `"A"` - After step 1: `"A"` - After step 2: `"A"` - After step 3: `"a"` - After step 4: `"a"` - After step 5: `"a"` - After step 6: `"a"` - SHA-256: `ca978112ca1bbdcafac231b39a23dc4da786eff8147c4e72b9807785afee48bb`

A.1.7. A.8. Complex Real-World Examples

Test 19: Article excerpt - Input: `"The café's "special" offer: 50% off!"` - Expected output after normalization: `the café's "special" offer: 50% off!` - SHA-256: `25cdbe2315674d38ddaf1df6fe7ccd494ce89efebe8a3b5285742e57e7367545`

Test 20: Unicode mixed with entities - Input: `"Cliché &amp; résumé"` - Expected output after normalization: `cliché & résumé` - SHA-256: `7d56f360edd22f7be0bc0f126d45481df83e8afc68b83788cf37544c4ee6ce21`

A.1.8. A.9. Implementation Notes

Computing SHA-256: - Encode the final normalized string as UTF-8 bytes - Compute SHA-256 over those bytes - Express result as 64 lowercase hex characters

Test Vector Validation: - Implementations claiming normalization support MUST produce the SHA-256 hashes specified above - The seven "hello world" variants (Tests 1, 2, 3, 11, 12, 14, 15) all normalize to identical output (`"hello world"`), demonstrating whitespace normalization equivalence - Tests 7 and 7b demonstrate NFKC combining character handling (both produce identical hashes)

A.2. Appendix B. Example Flows (Informative)

This appendix illustrates TCT discovery and fetch patterns.

A.2.1. B.1. Initial Discovery and First Fetch

Scenario: Client visits origin for the first time.

``` 1. Client -> Server: GET / HTTP/1.1 Host: example.com

1. Server -> Client: HTTP/1.1 200 OK Link: </llm-pages.json>; rel="index"; type="application/json" Content-Type: text/html
```

```
                [HTML body...]
```

2. Client -> Server: GET /llm-pages.json HTTP/1.1 Host: example.com
3. Server -> Client: HTTP/1.1 200 OK Content-Type: application/json; charset=utf-8 Cache-Control: max-age=0, must-revalidate

```
                {
                  "version": 1,
                  "profile": "tct-1",
                  "items": [
                    {
                      "cUrl": "https://example.com/article/",
                      "mUrl": "https://example.com/article/llm.json",
                      "etag": "sha256-abc123..."
                    }
                  ]
                }
```

4. Client -> Server: GET /article/llm.json HTTP/1.1 Host: example.com
5. Server -> Client: HTTP/1.1 200 OK Content-Type: application/json; charset=utf-8 ETag: "sha256-abc123..." Link: https://example.com/article/; rel="canonical"

```
                {
                  "profile": "tct-1",
                  "canonical_url": "https://example.com/article/",
                  "title": "Article Title",
                  "content": "Article content...",
                  "hash": "sha256-abc123..."
                } ```
```

Client actions after step 6: - Stores M-URL content locally - Caches ETag `"sha256-abc123..."` for future revalidation

A.2.2. B.2. Zero-Fetch Optimization (Content Unchanged)

Scenario: Client returns after some time; content hasn't changed.

``` 1. Client -> Server: GET /llm-pages.json HTTP/1.1 Host: example.com

1. Server -> Client: HTTP/1.1 200 OK Content-Type: application/json; charset=utf-8
```

```
{
  "version": 1,
  "items": [
    {
      "cUrl": "https://example.com/article/",
      "mUrl": "https://example.com/article/llm.json",
      "etag": "sha256-abc123..."
    }
  ]
}
```

2. Client local comparison:
   - Sitemap etag: "sha256-abc123..."
   - Cached ETag: "sha256-abc123..."
   - Match! -> Skip fetch entirely (zero-fetch)
3. Client uses locally cached content for /article/ ```

Result: Zero bytes transferred for M-URL; content known to be current.

A.2.3. B.3. Conditional Request (Content Unchanged)

Scenario: Sitemap etag differs from cache, but actual content hasn't changed.

``` 1. Client -> Server: GET /llm-pages.json HTTP/1.1

1. Server -> Client: [Sitemap shows etag: "sha256-def456..."]
2. Client comparison:
   - Sitemap etag: "sha256-def456..." (different!)
   - Cached ETag: "sha256-abc123..."
   - Mismatch -> Must fetch, but use conditional request
3. Client -> Server: GET /article/llm.json HTTP/1.1 If-None-Match: "sha256-abc123..."
4. Server -> Client: HTTP/1.1 304 Not Modified ETag: "sha256-abc123..."
5. Client actions:
   - Content unchanged; uses cached copy
   - Notes: sitemap was stale/inconsistent; no protocol violation ```

**Result:** Small 304 response instead of full payload.

### A.2.4. B.4. Content Changed

**Scenario:** Content has been updated.

``` 1. Client -> Server: GET /llm-pages.json HTTP/1.1

1. Server -> Client: [Sitemap shows etag: "sha256-xyz789..."]

2. Client comparison:
    - Sitemap etag: "sha256-xyz789..." (different)
    - Cached ETag: "sha256-abc123..."
    - Mismatch -> Fetch with If-None-Match
3. Client -> Server: GET /article/llm.json HTTP/1.1 If-None-Match: "sha256-abc123..."
4. Server -> Client: HTTP/1.1 200 OK Content-Type: application/json; charset=utf-8 ETag: "sha256-xyz789..."

```
                {
                  "canonical_url": "https://example.com/article/",
                  "title": "Updated Title",
                  "content": "Updated content...",
                  "hash": "sha256-xyz789..."
                }
```

5. Client actions:
    - Replaces cached content
    - Updates cached ETag to "sha256-xyz789..." ```

Result: Full new representation received.

A.2.5. B.5. Parity Mismatch Handling

Scenario: Sitemap etag doesn't match actual M-URL ETag (transient inconsistency).

``` 1. Client -> Server: GET /llm-pages.json HTTP/1.1

1. Server -> Client: [Sitemap shows etag: "sha256-old999..."]
2. Client -> Server: GET /article/llm.json HTTP/1.1 If-None-Match: "sha256-old999..."
3. Server -> Client: HTTP/1.1 200 OK ETag: "sha256-new000..." <- Different from sitemap!

```
                [Full M-URL JSON...]
```

4. Client actions:
    - Accepts M-URL response (valid per HTTP)
    - Uses ETag from M-URL response ("sha256-new000...") for future revalidation
    - Notes: sitemap inconsistency tolerated; no error ```

**Result:** Client falls back to standard HTTP caching; no protocol failure.

### A.2.6. B.6. Using HEAD for Efficient Freshness Check

**Scenario:** Client wants to check freshness before fetching.
```

``` 1. Client -> Server: HEAD /article/llm.json HTTP/1.1 If-None-Match: "sha256-abc123..."

1. Server -> Client: HTTP/1.1 304 Not Modified ETag: "sha256-abc123..."
2. Client actions:
   - Content unchanged; uses cached copy
   - Avoided transferring full body

OR (if content changed):

1. Server -> Client: HTTP/1.1 200 OK ETag: "sha256-xyz789..." Content-Length: 4567
2. Client -> Server: GET /article/llm.json HTTP/1.1 If-None-Match: "sha256-abc123..."
3. Server -> Client: HTTP/1.1 200 OK ETag: "sha256-xyz789..."

   ```
   [Full M-URL JSON...]
```
   ```

Note: HEAD support is optional; conditional GET is the primary mechanism.

A.2.7. B.7. Discovery via C-URL

Scenario: Client discovers M-URL directly from HTML page.

``` 1. Client -> Server: GET /article/ HTTP/1.1

1. Server -> Client: HTTP/1.1 200 OK Link: </article/llm.json>; rel="alternate"; type="application/json" Content-Type: text/html

   ```
   <!DOCTYPE html>
   <html>
   <head>
     <link rel="alternate" type="application/json"
           href="https://example.com/article/llm.json">
   </head>
   ...
   ```

2. Client -> Server: GET /article/llm.json HTTP/1.1
3. Server -> Client: HTTP/1.1 200 OK ETag: "sha256-abc123..." Link: https://example.com/article/; rel="canonical"

   ```
   [M-URL JSON...]
   ```

4. Client verifies:
   - Canonical link points back to /article/ -> Consistent OK ```

Result: M-URL discovered and validated without sitemap.
   ```

A.3. Appendix C. Implementation Notes (Informative)

This appendix provides guidance for implementers.

A.3.1. C.1. Reference Implementations

Note: The following implementations are provided as informative examples. Repository URLs, package names, and deployment details may change over time and are not normative.

The following implementations demonstrate TCT in production environments:

WordPress Plugin (PHP): - Repository: https://github.com/antunjurkovic-collab/tct-wp-plugin - Version: 1.0.0 - Deployment: 3 production sites (970 URLs total) - Features: - Automatic M-URL generation for posts/pages - M-Sitemap generation and caching - Strong ETag computation using canonical JSON - Normalization algorithm implementation - Dependencies: WordPress 6.0+, PHP 7.4+ - JSON serialization: `json_encode()` with `JSON_UNESCAPED_SLASHES | JSON_UNESCAPED_UNICODE`

Python Validator: - PyPI package: `collab-tunnel` (https://pypi.org/project/collab-tunnel/1.0.2/) - Version: 1.0.2 - Purpose: Protocol compliance testing - Features: - Validates M-URL and M-Sitemap structure - Tests ETag parity - Verifies canonical link bidirectionality - Runs normalization test vectors - Usage: `python from collab_tunnel import validate_origin results = validate_origin("https://example.com")`

Cloudflare Worker (Edge Proxy): - Repository: https://github.com/antunjurkovic-collab/tct-worker - Purpose: Demonstrates CDN integration - Features: - Proxies M-URLs with proper ETag handling - Implements 304 Not Modified caching - Handles conditional requests correctly - Deployment: Cloudflare Workers platform

A.3.2. C.2. Deterministic JSON Libraries

For RFC 8785 (JSON Canonicalization Scheme) compliance:

Python: `python import canonicaljson canonical_bytes = canonicaljson.encode_canonical_json(obj)` - Library: `pip install canonicaljson` - Docs: https://github.com/matrix-org/python-canonicaljson

JavaScript: `javascript const canonicalize = require('canonicalize'); const canonical_string = canonicalize(obj);` - Library: `npm install canonicalize` - Docs: https://github.com/cyberphone/json-canonicalization

Go: `go import "github.com/cyberphone/json-canonicalization/go/json" canonical, _ := json.CanonicalizeJSON(input)`

Alternative (Stable Ordering): If not using RFC 8785, ensure: - Object keys sorted lexicographically (at ALL nesting levels) - No insignificant whitespace - Consistent number formatting - UTF-8 encoding without BOM

Important: Simple key-sorting helpers (e.g., `Object.keys(obj).sort()` in JavaScript) are **insufficient** for nested objects. For production implementations claiming TCT conformance, use RFC 8785 libraries or implement the full specification. The code examples in C.3 are **illustrative only** and may not handle all edge cases correctly.

A.3.3. C.3. SHA-256 Computation

The examples in this section are illustrative only. By themselves they do not guarantee the deterministic JSON requirements of Section 6.1 unless combined with a complete canonicalization algorithm such as RFC8785.

Note: The examples below are simplified for illustration. For production use, ensure proper RFC 8785 canonicalization (see C.2) before hashing.

Python: ```python import hashlib import json

Appendix B. Canonical JSON

canonical_json = json.dumps(obj, ensure_ascii=False, sort_keys=True, separators=(',', ':')) canonical_bytes = canonical_json.encode('utf-8')

Appendix C. Hash

sha256_hash = hashlib.sha256(canonical_bytes).hexdigest() etag_value = f'"sha256-{sha256_hash}"' ```

JavaScript: ```javascript const crypto = require('crypto');

// Canonical JSON // NOTE: This is illustrative only; for full conformance, use RFC 8785 or a complete canonicalization implementation. const canonical_json = JSON.stringify(obj, Object.keys(obj).sort()); const canonical_bytes = Buffer.from(canonical_json, 'utf-8');

// Hash const sha256_hash = crypto.createHash('sha256').update(canonical_bytes).digest('hex'); const etag_value = `"sha256-${sha256_hash}"`; ```

PHP: ```php // Canonical JSON $canonical_json = json_encode($obj, JSON_UNESCAPED_SLASHES | JSON_UNESCAPED_UNICODE);

// Hash $sha256_hash = hash('sha256', $canonical_json); $etag_value = '"sha256-' . $sha256_hash . '"'; ```

C.1. C.4. Unicode Normalization

Python: ```python import unicodedata

Appendix D. NFKC normalization

normalized = unicodedata.normalize('NFKC', text)

Appendix E. Case folding

casefolded = normalized.casefold() ```

JavaScript: ```javascript // NFKC normalization const normalized = text.normalize('NFKC');

// Case folding (approximation: toLowerCase with locale-independent behavior) const casefolded = normalized.toLocaleLowerCase('en-US'); ```

PHP: ```php // NFKC normalization (requires intl extension) $normalized = Normalizer::normalize($text, Normalizer::NFKC);

// Case folding (mb_strtolower with UTF-8) $casefolded = mb_strtolower($normalized, 'UTF-8'); ```

E.1. C.5. HTTP Response Headers

Typical headers for M-URLs:

A conformant M-URL response will include at least: `Content-Type: application/json`, `ETag` (strong, quoted), and `Link: rel="canonical"` (see Sections 5-6). Example:

`HTTP/1.1 200 OK Content-Type: application/json; charset=utf-8 ETag: "sha256-abc123..." Link: <https://example.com/article/>; rel="canonical" Cache-Control: public, max-age=3600, must-revalidate Vary: Accept-Encoding`

Typical headers for M-Sitemap: `HTTP/1.1 200 OK Content-Type: application/json; charset=utf-8 Cache-Control: max-age=0, must-revalidate # Example; a short max-age consistent with Section 7.2 MAY also be used Vary: Accept-Encoding`

Conditional request handling: ``` # Request with If-None-Match GET /article/llm.json HTTP/1.1 If-None-Match: "sha256-abc123..."

Appendix F. Response if unchanged

HTTP/1.1 304 Not Modified ETag: "sha256-abc123..." Cache-Control: public, max-age=3600, must-revalidate

Appendix G. Response if changed

HTTP/1.1 200 OK ETag: "sha256-xyz789..." Content-Type: application/json; charset=utf-8 [full body...] ```

G.1. C.6. Production Deployment Data

Note: The following deployment data represents a snapshot as of November 2025. Specific sites, URLs, and metrics are provided as informative examples and may change.

As of November 2025, TCT is deployed on:

bestdemotivationalposters.com: - 500 URLs - 100K+ pageviews/month - WordPress 6.4 + TCT plugin v1.0.0 - Average HTML size: 103 KB (gzipped) - Average M-URL size: 17.7 KB (gzipped) - Bandwidth reduction: 83%

wellbeing-support.com: - 400 URLs - Health/wellness content - Average zero-fetch rate: 85%

omacedonii.com: - 70 URLs - Multilingual (Polish) - Demonstrates Unicode normalization in production

Aggregate Results: - Total URLs: 970 - Bandwidth reduction: 83% median - Zero-fetch rate: 70-90% (depends on update frequency) - Combined bandwidth elimination: ~98% in steady-state

G.1.1. C.7. Common Implementation Pitfalls

Pitfall 1: Non-deterministic JSON - Problem: Random key ordering, floating-point precision issues, timestamps - Solution: Use RFC 8785 or strict ordering; exclude per-request randomness

Pitfall 2: Weak ETags - Problem: Using `W/"sha256-..."` instead of `"sha256-..."` - Solution: Always use strong ETags for M-URLs (no W/ prefix)

Pitfall 3: Sitemap Staleness - Problem: Sitemap regenerated asynchronously; race conditions cause mismatches - Solution: Accept transient inconsistency; clients fall back to conditional GET

Pitfall 4: Incorrect Canonical Links - Problem: rel="canonical" points to wrong URL or is missing - Solution: Validate bidirectional linkage (C-URL <-> M-URL)

Pitfall 5: HTML in Content Field - Problem: Including raw HTML tags in `content` field - Solution: Extract plain text or use deterministic markup (e.g., Markdown)

Pitfall 6: Missing Vary: Accept-Encoding - Problem: CDN caches gzipped and uncompressed versions with same key - Solution: Always include `Vary: Accept-Encoding` header

Pitfall 7: Locale-Dependent Case Folding - Problem: Turkish İ -> i vs. I -> ı (locale-specific) - Solution: Use Unicode default case folding (locale-independent)

G.1.2. C.8. Testing and Validation

Automated Testing: 1. Protocol compliance: Use `collab-tunnel-validator` PyPI package 2. Normalization: Run Appendix A test vectors 3. ETag parity: Compare sitemap etag vs. actual M-URL ETag 4. Conditional requests: Test If-None-Match with matching/non-matching ETags

Manual Testing: 1. Check discovery: `curl -I https://example.com/` (look for Link header) 2. Fetch sitemap: `curl https://example.com/llm-pages.json` 3. Validate M-URL: `curl -H "Accept: application/json" https://example.com/article/llm.json` 4. Test 304: `curl -H 'If-None-Match: "sha256-..."' https://example.com/article/llm.json`

Integration Testing: - Deploy to staging environment - Monitor 304 response rates (should be >70% after initial crawl) - Check CDN cache hit rates - Verify canonical link bidirectionality

G.1.3. C.9. Performance Considerations

Server-Side: - Cache normalized content and ETags (don't recompute on every request) - Generate M-Sitemap asynchronously during content updates - Use HTTP/2 for parallel M-URL fetches - Implement early ETag validation (before full response generation)

Client-Side: - Use zero-fetch optimization when possible (sitemap comparison) - Batch M-Sitemap fetches (don't fetch per-URL) - Implement exponential backoff for 429/503 responses - Cache M-URLs and ETags persistently

CDN/Proxy: - Configure strong ETag preservation - Enable compression (gzip/brotli) with Vary: Accept-Encoding - Cache 304 responses appropriately - Don't strip ETag headers

G.1.4. C.10. Security Best Practices

HTTPS: - It is RECOMMENDED to serve M-URLs and M-Sitemap over HTTPS (see Section 10) - Validate TLS certificates properly - Use HSTS where appropriate

Access Control: - If C-URL requires authentication, protect M-URL similarly - Don't leak sensitive URLs in public sitemaps - Implement rate limiting to prevent abuse

Content Validation: - Treat M-URL JSON as untrusted input - Validate/sanitize before use - Don't execute code from content field

ETag Integrity: - Use Content-Digest (RFC 9530) for additional integrity - Consider HTTP Message Signatures (RFC 9421) for authentication - Monitor for ETag collision attempts (though SHA-256 makes this infeasible)

Appendix H. Acknowledgments

Thanks to reviewers and implementers who provided feedback on earlier revisions, including those who highlighted:

- the need for a single strong ETag method;
- the importance of deterministic JSON;
- the separation of protocol from policy; and
- simplification of discovery and parity semantics.

Author's Address

Antun Jurkovikj
North Macedonia
Email: antunjurkovic@gmail.com