PORTABLE CERTIFICATION CONTAINER

Deterministic JSON Certification for Distributed Systems

Technical Whitepaper • v4.0

February 2026

Chris Sharp

Executive Summary

JSON is the dominant data interchange format for web services, APIs, and distributed systems. Yet JSON has a fundamental problem: the same logical data can be serialized into different byte sequences. Whitespace, key ordering, number formatting, and Unicode escape handling all vary between implementations. This makes cryptographic operations—hashing, signing, integrity verification—unreliable without an additional canonicalization step.

Existing solutions address fragments of this problem. RFC 8785 (JSON Canonicalization Scheme) standardizes serialization but relies on IEEE-754 floating-point semantics, creating platform-dependent edge cases. JSON Schema validates structure but does not guarantee deterministic outcomes across implementations. Protocol Buffers offer deterministic serialization but abandon JSON entirely.

The Portable Certification Container (PCC) takes a different approach. Rather than solving canonicalization alone, PCC defines a complete deterministic pipeline: from raw input bytes through parsing, validation, normalization, and hashing to a final certification report. The guarantee is absolute within its scope: **given identical inputs and identical normative artifacts, any conforming implementation produces byte-identical reports.**

This whitepaper explains why that guarantee is hard to achieve, how PCC achieves it, what tradeoffs the design accepts, and where PCC fits relative to existing approaches.

1. The Problem: JSON Determinism Is Hard

Most developers treat JSON as a solved problem. Libraries parse and serialize it reliably enough for application logic. But cryptographic operations require a stronger property: the same data must always produce the same bytes. JSON, as specified in RFC 8259, does not provide this.

1.1 Sources of Non-Determinism

Key ordering. RFC 8259 states that JSON objects are unordered collections. Most serializers output keys in insertion order, hash-map order, or alphabetical order depending on language and library. Two serializers processing the same logical object routinely produce different byte sequences.

Number representation. The JSON number grammar allows multiple representations of the same value: 1.0, 1, 1e0, and 1.00 are all valid. RFC 8785 addresses this by mandating

ECMAScript's number-to-string algorithm, but that algorithm is defined in terms of IEEE-754 double-precision arithmetic. Languages without native double semantics must emulate the exact ECMAScript behavior—a non-trivial requirement that produces subtle divergence at the edges of the representable range.

Unicode handling. JSON strings can represent the same character as a literal UTF-8 byte or a \uXXXX escape. Surrogate pairs, Unicode normalization, and BOM handling add further variation.

Duplicate keys. RFC 8259 recommends against duplicate keys but does not forbid them. Parsers handle duplicates differently: some take the first value, some take the last, some reject the input. A document with duplicate keys can produce different parse trees in different implementations.

Whitespace and formatting. Insignificant whitespace is the most visible source of non-determinism, but also the easiest to solve. Canonicalization removes it. The harder problems are above.

1.2 Why Existing Solutions Are Incomplete

RFC 8785 (JCS) solves serialization determinism for the IEEE-754 numeric domain. It does not address: duplicate keys (assumes I-JSON compliance), schema validation, hash dependency ordering, or pipeline-level determinism. JCS is a canonicalization algorithm, not a certification framework.

JSON Schema validates structure and constraints but is not deterministic by design. Two compliant validators can disagree on edge cases in $dynamicRef resolution, pattern matching behavior, or annotation collection.

XML Digital Signatures (XMLDSig) solved this problem for XML using Canonical XML (C14N). The solution works but is complex, fragile, and tightly coupled to XML's namespace model. It provides a cautionary example: canonicalization alone is insufficient without constraints on the input model.

Protocol Buffers offer deterministic serialization within a single build of a single implementation, but the documentation explicitly states this is not canonical across implementations or versions.

> *The gap: no existing standard provides end-to-end deterministic certification of JSON documents—from raw bytes to a verifiable report—without requiring non-JSON intermediate formats or platform-specific behavior.*

2. Architecture Overview

PCC defines a pipeline of discrete phases. Each phase has explicit inputs, outputs, and failure modes. The pipeline is specified in a prose document (the PCC Specification) and a set of normative JSON artifacts that contain the exact algorithms, rules, and parameters.

2.1 The Pipeline

A PCC certification run processes inputs through these phases:

Phase	Operation	Purpose
1. Bootstrap	Manifest hash verification, parse, conflict scan	Establish trust anchor; verify artifact integrity
2. Parse Gate	UTF-8, syntax, int-only numbers, duplicate keys	Identical parse trees across implementations
3. Projection	Map raw inputs to JSON documents	Deterministic input normalization
4. Normalization	Apply field defaults, validate required fields	Eliminate optional-field variance
5. Schema	Validate against profiled JSON Schema subset	Structural correctness, deterministic evaluation
6. Ordering	Verify array sort order	Detect ordering violations without resorting
7. Hash Views	Extract subtrees, canonicalize, hash	Produce cryptographic bindings
8. Execution	Sandboxed computation on hash views	Deterministic derived outputs
9. Report	Generate certification report	Byte-identical output across implementations

The first terminal failure halts the pipeline. Events from all entered phases are collected into the report. The report is always emitted, even on failure.

2.2 The Trust Model

PCC's trust model is anchored in a single SHA-256 hash: the manifest hash. This value, provided out-of-band, binds to the manifest, which in turn binds (by hash) to every normative artifact. Change any artifact and the manifest hash changes. Change the manifest hash and verification fails.

This creates a Merkle-like integrity chain: one hash value verifies the entire normative surface. The verifier does not need network access, trusted third parties, or certificate authorities. It needs the manifest hash and the artifact bytes.

2.3 The Artifact Delegation Model

PCC splits authority between a prose specification and a set of JSON artifacts. The specification defines the pipeline structure, invariants, and failure semantics. The artifacts define the exact algorithms, parameters, and rules.

This is PCC's most unusual architectural decision. In most specifications, the prose document is the highest authority. In PCC, the artifacts prevail when they conflict with the prose. The specification is the interface between human understanding and machine verification; the artifacts are the machine-verifiable truth.

The rationale: prose is ambiguous; JSON is not. A natural-language description of a sorting algorithm can be interpreted differently by two implementers. A JSON object defining the sort key, comparison type, and failure code cannot. By making artifacts authoritative, PCC ensures that conformance testing is always against the exact same rules.

> *The specification tells you why each phase exists and what invariants it maintains. The artifacts tell you exactly what to implement. Both are necessary; neither is sufficient alone.*

3. Key Design Decisions

3.1 Integer-Only Numbers

PCC rejects all floating-point JSON: no fractions, no exponents, no scientific notation. Numbers must be integers within the signed 64-bit range. The literal -0 is forbidden.

This eliminates the largest source of cross-platform numeric divergence. IEEE-754 double-precision can represent integers exactly up to 2^{53}, but different languages handle the conversion between their native number types and JSON differently. By restricting to int64, PCC ensures that every conforming parser produces exactly one internal representation for every valid number.

PCC's canonicalization (PCC-JCS) modifies RFC 8785 accordingly: numbers are serialized as exact base-10 integer strings. No floating-point conversion is involved.

3.2 The Parse Gate

The parse gate is PCC's first line of defense against implementation divergence. It applies to every JSON document in PCC scope and enforces constraints stricter than RFC 8259:

- UTF-8 validation including surrogate rejection after escape decoding
- Duplicate key rejection (RFC 8259 recommends; PCC requires)
- Integer-only number validation with int64 range enforcement
- Deterministic error precedence when multiple errors coexist

The error precedence rule is critical. Without it, two parsers processing the same malformed input could report different first errors, producing different reports. PCC defines a five-tier priority (UTF-8 > syntax > number syntax > duplicate keys > number overflow) that ensures identical error reporting regardless of parser implementation strategy.

3.3 Extensions: The "Dark Space"

PCC reserves the /extensions path as a region where arbitrary JSON data can exist without affecting certification outcomes. Extension content must be valid JSON (parse gate applies), but it is excluded from schema validation, hash views, ordering, and execution.

The guarantee: two inputs differing only in /extensions content produce identical certification reports. This is enforced by conformance vectors using black-box mutation testing.

Extensions enable PCC-certified documents to carry metadata, provenance information, or application-specific data without breaking the certification guarantee.

3.4 Hash Dependency Management

When a JSON document contains multiple hash fields, the order of hash computation matters. If field A's hash view includes content that depends on field B's hash, then B must be computed first.

PCC models this as a directed graph: hash fields are nodes, and edges connect fields whose views overlap. Cycles are terminal failures. Acyclic graphs are resolved by topological sort with deterministic tie-breaking.

3.5 Profiled JSON Schema

PCC uses JSON Schema Draft 2020-12 but removes features that introduce non-determinism:

- $dynamicRef and $dynamicAnchor are forbidden (complex resolution semantics)
- pattern is forbidden (regex engine differences across platforms)
- Content-related keywords are forbidden (encoding-dependent behavior)
- $ref resolution is restricted to an explicit allowlist (no network, no relative URIs)

PCC adds one vocabulary: `pcc:hashField`, an annotation keyword that marks fields as containing hash values. This enables the hash dependency analysis described above.

4. Accepted Tradeoffs

Every design decision involves tradeoffs. This section documents the significant ones honestly.

4.1 Artifact Delegation vs. Human Auditability

Because artifacts prevail over prose, an auditor cannot fully evaluate PCC by reading only the specification document. Auditing PCC requires examining both the specification and the JSON artifacts.

The mitigation: every artifact is hash-bound in the manifest. Artifact changes produce a different manifest hash. An auditor can verify they are examining the exact artifact set that was certified. But they still need tooling to interpret the artifacts meaningfully.

4.2 Conformance by Test Vectors vs. Formal Semantics

PCC defines conformance partly through test vectors rather than a formal parser model. Any parsing strategy is acceptable if it produces identical results on all conformance vectors.

This is a pragmatic choice. A formal parser model would make the specification correct by construction but significantly harder to implement. The risk: edge cases not covered by the vector suite could allow two “conforming” implementations to diverge.

This is the single largest accepted risk in the PCC design. The conformance suite is effectively the most important artifact in the set.

4.3 Conflict Scan Limitations

PCC's conflict scan prevents the manifest hash from appearing as a literal string value in normative artifacts. However, the scan is intentionally limited: object keys are not scanned, extension subtrees are skipped, and the hash could be reconstructed from multiple values. The scan provides structural hygiene, not cryptographic containment.

4.4 Name-Based Extension Scoping

PCC uses two scoping models for extensions. Path-based scoping (/extensions/**) governs the core exclusion guarantees. Name-based scoping (any object key literally named "extensions") governs tie-break stripping and conflict scan exclusion. A domain data field named "extensions" outside the root path will be stripped during ordering tie-breaks. This is an accepted asymmetry.

4.5 The Fallback Report

If report generation itself fails (an implementation bug), a minimal fallback report is emitted. This fallback is explicitly nondeterministic: two implementations with different internal errors may produce different fallback reports. This violates PCC's core determinism guarantee. It is accepted because the alternative—no report on internal error—is worse.

5. Comparison to Existing Approaches

	PCC	JCS (RFC 8785)	XMLDSig	Protobuf	VC Data Integrity
Scope	Full pipeline	Serialization	Signature	Serialization	Proof generation
Format	JSON (int-only)	JSON (IEEE-754)	XML	Binary	JSON-LD / JSON
Det. parse	Yes (parse gate)	Assumes I-JSON	C14N	Schema-defined	Varies by suite
Schema	Profiled subset	None	XSD	Proto definition	JSON-LD context
Hash deps	Graph + topo sort	None	Reference model	None	Proof chains
Cross-platform	By design	Mostly	Yes (complex)	Same build only	Suite-dependent

PCC occupies a different niche than these alternatives. JCS solves canonicalization; PCC uses a JCS profile as one component of a larger pipeline. XMLDSig proves that end-to-end deterministic signatures are possible but shows the complexity cost. W3C Verifiable Credentials Data Integrity provides a proof framework for credentials but delegates canonicalization to cryptographic suites, each with their own determinism properties.

PCC is most directly comparable to VC Data Integrity when used with the eddsa-jcs-2022 cryptosuite. The difference is scope: VC Data Integrity defines how to attach proofs to credentials; PCC defines how to deterministically certify arbitrary JSON documents through a complete verification pipeline.

6. Adoption & Implementation

6.1 What an Implementer Builds

A PCC verifier implementation requires:

- A JSON parser that enforces PCC's parse gate constraints
- SHA-256 hashing
- PCC-JCS canonicalization (RFC 8785 with integer-only number rendering)
- A JSON Schema validator supporting the profiled Draft 2020-12 subset
- JSON Pointer (RFC 6901) resolution with PCC's restricted profile
- A sandboxed execution environment (if execution phases are used)

The normative artifacts define the specific rules, schemas, and parameters. The implementer writes the pipeline engine; the artifacts configure it.

6.2 Conformance Testing

PCC provides a conformance test suite containing baseline vectors that cover every failure code and pipeline behavior. An implementation passes conformance when it produces byte-identical reports for every vector.

This is a high bar. Most JSON-processing specs define conformance in terms of "correct behavior"; PCC defines it as byte-identical output. Conformance is fully automatable: run the vectors, compare the bytes, pass or fail.

6.3 Use Cases

Regulatory compliance. When regulatory frameworks require verifiable, tamper-evident JSON data—financial filings, healthcare records, audit trails—PCC provides cryptographic certification with deterministic verification.

Supply chain integrity. PCC-certified manifests can accompany software bills of materials (SBOMs), configuration files, or deployment descriptors. Verification requires no network access and no trusted intermediary.

Distributed systems. When multiple nodes must agree on the validity and content of a JSON document, PCC eliminates “it parses differently on my machine” disagreements.

Archival and provenance. PCC’s hash-bound artifact model means a certification report from 2026 can be verified in 2036 by anyone with the original artifacts and manifest hash. No infrastructure dependencies, no certificate expiration, no protocol versioning.

7. Future Work

- Developer Guide: a companion document with worked examples and implementation guidance
- Reference implementation: an open-source PCC verifier as the byte-identical baseline
- Artifact authoring tools: utilities for creating and validating normative artifact sets
- Extended conformance vectors: expanding the baseline suite through implementation experience
- Formal analysis: investigating whether key PCC invariants can be proven formally rather than tested empirically

---

PCC Specification v4.0 and this whitepaper are available at the project repository. The specification is normative; this whitepaper is explanatory. Where they conflict, the specification prevails.