Portable Certification Container (PCC) Specification v4.0
Status: Normative Date: 2026-02-11

1 Scope

PCC defines a deterministic pipeline for certifying JSON documents. Given identical inputs and identical normative artifacts, any conforming implementation MUST produce byte-identical reports.

PCC covers: JSON parsing constraints, input projection, normalization, schema validation, array ordering verification, cryptographic hash views, phased execution, and certification reporting.

PCC does not cover: domain semantics of certified data, floating-point arithmetic, network protocols, concurrency, performance, or content beyond the JSON subset defined herein.

2 Conformance

An implementation conforms to this specification if and only if:

(a) It satisfies every MUST, MUST NOT, SHALL, SHALL NOT, and REQUIRED in this document and in the normative artifacts listed in §4.1.

(b) It passes all baseline conformance vectors in conformance_runner_v1.json (§18).

(c) For any input accepted by both a conforming implementation and a reference implementation, the two produce byte-identical reports under PCC-JCS serialization (§A).

Keywords MUST, MUST NOT, SHALL, SHALL NOT, REQUIRED, and FORBIDDEN are as defined in RFC 2119 and RFC 8174.

3 Normative References

- RFC 2119 Key words for use in RFCs
- RFC 3629 UTF-8
- RFC 6901 JavaScript Object Notation (JSON) Pointer
- RFC 8174 Ambiguity of Uppercase vs Lowercase in RFC 2119
- RFC 8259 The JavaScript Object Notation (JSON) Data Interchange Format
- RFC 8785 JSON Canonicalization Scheme (JCS)
- JSON Schema Draft 2020-12

4 Normative Artifacts & Precedence

4.1 Artifact Set

The following artifacts, enumerated in pcc_manifest_v1.json, are normative. Each is identified by an Artifact Identifier (§5) and bound by SHA-256 hash.

pcc_manifest_v1.json parse_profile_v1.json normalization_rules_v1.json schema_runtime_v1.json ordering_rules_v1.json hash_view_rules_v1.json checkpoint_schedule_v1.json input_model_v1.json spec_projection_v4.0.json report_schema_v1.json error_registry_v1.json conformance_runner_v1.json meta_schema_container_v1.json

Every truth table, algorithm specification, or appendix referenced in this document MUST be embedded in a listed artifact. Content not embedded in a listed artifact is non-normative.

4.2 Precedence
When this document and a normative artifact conflict, the artifact prevails.
Precedence order (highest first):

1. Manifest bytes and out-of-band hash contract (§6.2).
2. Normative artifacts (§4.1).
3. This document (where artifacts are silent).

4.3 Error Registry
All event codes (fail:*, violation:*, info:*) referenced in this document or in any normative artifact MUST exist in error_registry_v1.json. Each entry MUST include: code (string), class (failure, violation, or info), and priority (integer).
Completeness is validated after all normative artifacts pass the parse gate (§7) and conflict scan (§6.2). Implementations emit event codes as specified in this document; registry presence is not a prerequisite for emitting a code at runtime.
Incomplete registry → terminal fail:error_registry_incomplete.
5 Artifact Identity
An Artifact Identifier is a JSON string consisting exclusively of ASCII bytes:
First byte: 0x41–0x5A | 0x61–0x7A | 0x30–0x39 Subsequent bytes: 0x41–0x5A | 0x61–0x7A | 0x30–0x39 | 0x2E | 0x5F | 0x2D Length: 1–128 bytes
0x2E (.) is not a valid first byte. The strings "." and ".." are forbidden. Any non-ASCII byte → fail:identifier_invalid.
Identifiers are case-sensitive. No Unicode normalization is applied. Identifiers are opaque tokens; implementations MUST NOT interpret them as filesystem paths.
Duplicate identifiers within array contexts (conformance vector lists, ordering rule declarations) → terminal fail:identifier_duplicate. For identifiers appearing as JSON object keys, §7.2 (duplicate key rejection) applies instead.
Invalid identifier → terminal fail:identifier_invalid.
6 Trust Anchor
6.1 Manifest Bootstrap
The manifest (pcc_manifest_v1.json) is parsed using only this document and the parse gate (§7). It MUST be a JSON object containing:
"artifacts" REQUIRED. Object. Keys are Artifact Identifiers; values are objects containing at minimum "sha256" (64 lowercase hex characters).
"spec_version" REQUIRED. String.
"inputs" OPTIONAL. Object. Keys are Artifact Identifiers; values are objects containing at minimum "sha256" (64 lowercase hex) and "media_type" (string).
"outputs" OPTIONAL. Object. Keys are Artifact Identifiers; values are objects containing at minimum "sha256" (64 lowercase hex).
Keys MUST be unique across all present sections. Collision → terminal fail:manifest_namespace_collision.
Manifest semantic checks execute in this order after the parse gate:

1. Required top-level fields ("artifacts", "spec_version"). Absent → fail:manifest_missing_field.
2. Identifier grammar (§5) on keys in all present sections. Invalid → fail:identifier_invalid.

3. Cross-section key uniqueness. Collision → fail:manifest_namespace_collision.
4. Per-entry required fields ("sha256", etc.).

First terminal halts manifest semantic checks.

The verifier receives: (a) manifest raw bytes, (b) manifest_sha256 out-of-band, and (c) raw bytes for every manifest-listed artifact and input. Missing artifact → terminal fail:artifact_missing.

6.2 Hash Verification & Conflict Scan

manifest_sha256 MUST be exactly 64 lowercase hexadecimal characters matching [0-9a-f]{64}. Other format → terminal fail:manifest_sha256_format.

The verifier computes SHA-256 of the supplied manifest bytes and compares to manifest_sha256. Mismatch → terminal fail:manifest_hash_mismatch. Parsing begins only after verification succeeds.

Conflict scan: manifest_sha256 MUST NOT appear as a decoded JSON string value in any normative artifact (including the manifest itself). Inputs and outputs are not scanned.

Scan rules:

- Only decoded string values are inspected. Keys, numbers, booleans, and nulls are not inspected.
- The scan traverses all values including array elements.
- Object members keyed "extensions" are skipped. This is name-based: any object member whose key is "extensions" at any depth causes that member's value subtree to be excluded. String values equal to "extensions" appearing as array elements are not special.
- The scan runs per normative artifact as a post-parse-gate substep. It short-circuits on first match.

Match → terminal fail:manifest_sha256_conflict.

NOTE — The conflict scan provides structural hygiene against literal self-reference, not cryptographic containment. The anchor may appear in object keys, under "extensions" subtrees, or be reconstructible from multiple values. These are accepted limitations of the design.

6.3 Manifest Canonical Binding

After §6.2 verification:

1. Parse gate (§7) with all failure codes prefixed manifest_ (e.g., fail:manifest_parse_utf8).
2. PCC-JCS canonicalization (§A).
3. SHA-256 of canonical bytes → manifest_canonical_sha256.

manifest_canonical_sha256 is a required report field (defined in report_schema_v1.json). Failure at any step is terminal.

6.4 Meta-Schema Container

meta_schema_container_v1.json MUST be valid JSON under the parse gate. It MUST contain a deterministic array of entries, each with:

"$id" String, unique across entries. "sha256" 64 lowercase hex characters.

"schema" The meta-schema object.
Duplicate $id → terminal fail:metaschema_duplicate_id.

7 Parse Gate

The parse gate (parse_profile_v1.json) applies to all JSON processed by PCC: manifest, normative artifacts, projected documents, reports, extensions, conformance vectors, and schemas. The parse gate runs before all semantic checks.

7.1 UTF-8

UTF-8 encoding per RFC 3629. Invalid bytes → terminal fail:parse_utf8.
Surrogate code points U+D800–U+DFFF MUST be rejected after JSON escape decoding. The escape sequence \uD800 consists of valid UTF-8 bytes but decodes to a surrogate; implementations MUST check after unescape. Surrogate-after-unescape reports fail:parse_utf8 and is classified as UTF-8 for priority purposes (§7.5).
No Unicode normalization is applied at any point in PCC processing.

7.2 Object Keys

Duplicate keys within a single JSON object → terminal fail:parse_duplicate_key.
Key equality is defined as identical Unicode scalar sequences after JSON escape decoding.

7.3 Numbers

PCC JSON is integer-only. Grammar: -?(0|[1-9][0-9]*)
Reject: fractions, exponents, leading +, leading zeros, NaN, Infinity. The literal -0 is explicitly forbidden → fail:parse_number_syntax.
Range: signed 64-bit integer [-9223372036854775808, 9223372036854775807]. Overflow → fail:parse_number_overflow. Other syntax errors → fail:parse_number_syntax.

7.4 Structural Syntax

Any JSON syntax error not covered by §7.1–§7.3 → terminal fail:parse_syntax. This includes: missing commas, unmatched braces/brackets, invalid tokens, malformed \uXXXX escape sequences, and unrecognized escape sequences. Surrogate code points after unescape are covered by §7.1, not this section.
For the manifest: fail:manifest_parse_syntax per §6.3.

7.5 Failure Precedence

The parse gate processes input in a single left-to-right pass. The first failure encountered is reported. "Encountered" means detected at the earliest point during streaming parse; for duplicate keys, this is when the second occurrence is fully parsed.
When multiple errors are detected at the same token boundary, the following priority applies:

1. UTF-8 (§7.1)
2. Structural syntax (§7.4)
3. Number syntax (§7.3)
4. Duplicate keys (§7.2)
5. Number overflow (§7.3)

The first terminal failure halts processing of that artifact.
NOTE — Conformance is defined by test vector outcomes (§18), not by a formal parser model. Implementations may use any parsing strategy that produces identical results on all conformance vectors.
8 Input Model
Inputs are tuples of (input_id, raw_bytes, media_type).
input_id MUST satisfy §5 grammar. Invalid → fail:input_id_invalid. Input hash is SHA-256 of raw_bytes with no transformation.
8.1 Media Types
The media type allowlist is defined in input_model_v1.json.
media_type values MUST consist of ASCII bytes 0x21–0x7E (printable, no space). Non-conforming → fail:input_media_type_encoding. Allowlist entries MUST also satisfy this constraint.
Matching is by exact byte equality against allowlisted lowercase strings. No trimming, no case folding, no MIME parameter parsing unless the parameter string is allowlisted verbatim.
No match → fail:input_unknown_media_type.
9 Projection
Deterministic mapping from inputs to JSON documents.
Traversal order: lexicographic by input_id (UTF-8 byte comparison). The complete derivation algorithm — encoding, escaping, length limits, collision domains, and cardinality per media type — is specified in spec_projection_v4.0.json.
Collision → fail:projection_id_collision. Forbidden dependencies: filenames, paths, locale, time, environment.
Invalid UTF-8 during media-type-driven decoding → fail:projection_utf8. If the input media type is JSON, UTF-8 errors are caught by the parse gate (§7.1) as fail:parse_utf8.
9.1 Pointer Outcome Truth Table
Embedded in spec_projection_v4.0.json. Defines four distinct pointer resolution outcomes: missing, null, wrong-type, out-of-bounds. Implementations MUST NOT conflate these outcomes.
10 Normalization
Rules are explicit and closed. All synthesis rules, placement constraints, and field requirements are specified in normalization_rules_v1.json. "Required fields" in this section refers to fields designated by normalization_rules_v1.json, not JSON Schema required.
Missing required fields → violation. Violation codes are declared in normalization_rules_v1.json and MUST exist in the error registry (§4.3). A violation is recorded even when a deterministic default is synthesized; synthesis does not suppress the violation event. Synthesized values are deterministic and are included in hash views.
missing_determinism_fields is ordered by UTF-8 byte comparison of pointer strings.
11 Schema Runtime

PCC uses a profiled subset of JSON Schema Draft 2020-12. Required vocabularies are listed in schema_runtime_v1.json.

11.1 PCC Hash-Field Vocabulary

The pcc:hashField annotation keyword is defined as a PCC vocabulary. The vocabulary MUST appear in the required vocabulary list with full semantics specified in schema_runtime_v1.json.

11.2 Restrictions

Unknown vocabularies → terminal fail:schema_unknown_vocabulary. The vocabulary detection algorithm MUST be fully defined in schema_runtime_v1.json.

Forbidden keywords: $dynamicRef, $dynamicAnchor, contentEncoding, contentMediaType, contentSchema, pattern. Presence → terminal fail:schema_forbidden_keyword.

$ref: no network access. Resolution is by exact byte equality against allowlisted $id values. Relative $ref URIs are forbidden; all $ref values MUST be absolute. No URI normalization, no base-URI resolution. Allowlist defined in schema_runtime_v1.json.

$id: this restriction applies to schema documents only. $id appearing in instance or projected documents is ordinary domain data. Unknown $id in a schema → fail:schema_unknown_id.

format: annotation only; not enforced.

unevaluatedProperties / unevaluatedItems: required. The evaluation algorithm is fully pinned in schema_runtime_v1.json.

11.3 Schema Compliance

All schemas and bundled meta-schemas MUST satisfy parse_profile_v1. Bundled meta-schemas MUST omit forbidden keywords.

12 Ordering

Ordering checks apply only to arrays declared in ordering_rules_v1.json.

12.1 Comparator

Key pointers: declared PCC Pointers (§14.1). Ordering keys MUST NOT target / extensions/** → fail:ordering_extension_key. The empty-string pointer (document root) is not permitted as an ordering key.

String comparison: encode decoded scalars to UTF-8 per RFC 3629, compare bytes unsigned left-to-right. If one string is a prefix of the other, the shorter string is lesser.

Numeric comparison: signed int64. No type coercion; if the resolved value is not the expected type → fail:ordering_key.

Ordering uses PCC Pointer resolution (§14.2) but collapses all non-present outcomes (wrong-type step, missing leaf, out-of-bounds) to fail:ordering_key. This collapse is intentional; it simplifies the consumer-facing error model at the cost of requiring implementers to maintain awareness of which resolution mode is active. Pointer parse failures (fail:pointer_invalid, fail:pointer_empty_token) in ordering rule declarations are terminal at rule load time.

Ordering checks evaluate elements left-to-right. The first fail:ordering_key halts the check.

12.2 Tie-Breaker

When primary keys are equal:

1. Compute extension-stripped element: depth-first walk of the JSON value. At each object, delete members keyed "extensions" before visiting child values. Deleted subtrees are never entered. Continue into remaining object members and all array elements.
2. Compute PCC-JCS canonical bytes (§A) of the stripped element.
3. Compare canonical bytes lexicographically.

NOTE — Tie-break stripping is name-based (§13.6). Domain-significant fields keyed "extensions" outside the root /extensions path are stripped for tie-break purposes. When primary keys tie and stripped canonicalizations are identical, parse-order stability applies (§12.3).

12.3 Stability

Equal keys and equal tie-break → preserve parse-order sequence. Earlier phases MUST NOT reorder arrays. Normalization MUST NOT add or remove array elements unless normalization_rules_v1.json specifies a deterministic position.

12.4 Presorted Arrays

ordering_rules_v1.json MUST declare per array whether an unsorted array produces a failure or a violation, with the corresponding event code. The validator MUST NOT reorder arrays.

13 Extensions

Extensions are allowed only under /extensions/** (the pointer is /extensions or begins with /extensions/).

13.1 Parsing & Size Ceilings

The parse gate (§7) applies to extension content. Size ceilings are defined in normalization_rules_v1.json; each ceiling specifies its measurement unit, measurement point, and outcome.

13.2 Parse Failures

A parse-gate failure in /extensions/** is terminal. All extension content MUST be valid JSON under the full parse gate, including integer-only number constraints.

13.3 Exclusion

Extensions are excluded from: schema validation, ordering key extraction (stripped in §12.2), hash views, and execution.

Execution pointers targeting /extensions/** → terminal fail:execution_extension_leak.

13.4 Mutation Invariance

Inputs differing only in /extensions/** content (both passing parse gate and size ceilings) MUST produce identical hashes, outputs, and report fields. This property is verified by conformance vectors using black-box testing.

13.5 Traversal Constraint

No phase other than the parse gate and size ceiling checks may traverse /extensions/** content for semantic purposes. The §6.2 conflict scan skips subtrees keyed "extensions" (name-based) and therefore does not inspect extension content.

This constraint is enforced through the mutation invariance property (§13.4): any implementation whose semantic phases inspect extension content will fail the corresponding conformance vectors.

13.6 Name-Based vs Path-Based Scoping

Two scoping models coexist in PCC:

Path-based (/extensions/**): §13.1–§13.5. Applies only to the root extensions path.

Name-based (object key "extensions" at any depth): §12.2 tie-break stripping, §6.2 conflict-scan exclusion.

Name-based behaviors extend beyond the root /extensions path. A domain field whose key is "extensions" but which is not under the root /extensions path is stripped during tie-break and excluded from conflict scans, but it is NOT an extension. Such fields are ordinary domain data for schema validation, hash views, and execution.

14 Hash Views

14.1 PCC Pointer

PCC Pointer is a restricted profile of RFC 6901. It is the sole pointer language in this specification.

A PCC Pointer either starts with / or is the empty string. The empty string represents zero tokens (the document root); it is distinct from an empty token. The root pointer is permitted in hash view definitions and projection pointers but NOT in ordering key declarations.

Escaping: only ~0 (for ~) and ~1 (for /) within tokens. Any other use of ~ → terminal fail:pointer_invalid. Empty tokens are forbidden → fail:pointer_empty_token. The bare pointer / (a single empty token) → fail:pointer_empty_token.

Pointers appearing in normative artifacts, events, reports, and hash field identifiers MUST use only ~0 and ~1 escaping. Non-conforming → fail:pointer_invalid.

No Unicode normalization. Pointer ordering: unsigned UTF-8 byte comparison, left-to-right; shorter is lesser if prefix-equal.

14.1.1 Array Index Tokens

Grammar: 0 | [1-9][0-9]* No leading zeros, no negatives, no -. Range: [0, 9223372036854775807].

Overflow produces a non-index token (wrong-type step, not out-of-bounds). Non-index token against array → wrong-type step (§14.2). Valid index ≥ array length → out-of-bounds (§14.2).

14.2 View Construction

Hash views are evaluated after normalization. Pointers into /extensions/** are forbidden.

Each view is scoped to a single document. Pointer resolution is rooted at that document's root object.

A view is a JSON object mapping pointer strings to deep-copied subtrees. No merging. Duplicate pointers (by decoded token sequence equality) → terminal fail:hash_view_duplicate_pointer.

Pointer resolution outcomes:

Present Resolves to a value (including null). Included in view. Missing leaf Final token absent. Triggers per-view policy. Wrong-type Token targets incompatible type → fail:hash_view_pointer_type. Out-of-bounds Valid index ≥ length → fail:hash_view_pointer_bounds.
Per-view missing-pointer policy is defined in hash_view_rules_v1.json: "omit" (exclude entry) or "fail" (fail:hash_view_missing_pointer).
14.3 Serialization
PCC-JCS canonicalization (§A) of the view object produces the hash input bytes.
14.4 Hash Dependency Rule
Hash fields are marked by the pcc:hashField annotation (§11.1). Hash field discovery is performed by schema evaluation: the set of instance locations where pcc:hashField evaluates to true, producing PCC Pointers in the normalized instance. Discovery occurs after schema validation completes.
In schemas with conditional applicators (if/then/else, oneOf, anyOf, allOf), annotations are collected from all subschemas that successfully validate the instance, per Draft 2020-12 annotation collection semantics. schema_runtime_v1.json MUST pin the exact collection algorithm.
Containment: pointer P contains hash field Q if the decoded tokens of Q equal those of P, or if P's tokens are a leading subsequence of Q's tokens.
Exclusion: a view MUST NOT include a pointer whose subtree contains a hash field → fail:hash_view_contains_hash_field.
Dependency graph:

- Nodes: hash fields, identified by PCC Pointer.
- Edges: A → B if any pointer in view(A) overlaps any pointer in view(B). Overlap: decoded token sequences where P = Q, P is a prefix of Q, or Q is a prefix of P. This is syntactic; runtime disjointness MUST NOT relax edges.
- Scope: all hash fields in the certification run. In multi-document runs, identity is compound: (document_id, pointer). Two pointers in different documents never overlap.
- Cycle → fail:hash_cycle.
- Topological sort order; ties broken by (document_id, pointer) with document_id compared first (UTF-8 byte comparison).

15 Pipeline & Halt
Phases execute in the order defined by checkpoint_schedule_v1.json. The first terminal failure halts the pipeline. Events are recorded only for entered phases. Parse-gate and conflict-scan events are attributed to the phase that triggered processing of the artifact, not to a separate phase.
Intra-phase check ordering is defined in checkpoint_schedule_v1.json. The parse gate (§7) and conflict scan (§6.2) always precede schema, ordering, and hash checks for each artifact. These are post-parse substeps and are not reorderable. For the manifest, §6.1 defines its own semantic check ordering during the bootstrap phase.

15.1 Report Emission

Report emission is mandatory post-pipeline finalization, not a phase. It executes unconditionally after the pipeline completes or halts.

If report generation itself fails (implementation error): emit a minimal report with certification_result = "failed" and a single event fail:report_generation with phase_index = 0. This overrides §17.1 minimums.

NOTE — The fallback report is an explicitly nondeterministic escape hatch. Two implementations with different internal errors may produce different fallback reports from the same input. This is accepted; the alternative (no report on internal error) is worse.

16 Event Model

16.1 Event Classes

Three classes, exhaustive: failure Terminal. Halts pipeline. violation Non-terminal. Recorded, processing continues. info Informational.

16.2 Certification Result

Any failure → "failed". Else any violation → "violated". Else → "certified".

16.3 Event Ordering

Applied at report generation. Each event MUST carry a phase index (integer). Events are ordered by:

1. Phase index (ascending).
2. Class (failure < violation < info).
3. Priority (lower wins).
4. Code (UTF-8 byte comparison).
5. Pointer or identifier (UTF-8 byte comparison; absent = empty string; pointer takes precedence over identifier if both present).

17 Report

The report is always emitted (§15.1) and MUST satisfy parse_profile_v1.

Optional fields are omitted, never null. If a value is not deterministically available, the field MUST be omitted unless report_schema_v1.json defines an explicit sentinel for that condition.

Hashes are lowercase hexadecimal SHA-256. Reports MUST NOT contain environment paths or host identifiers.

17.1 Minimal Report on Early Halt

A report emitted after early halt MUST include: certification_result ("failed"), phases_entered, and events. Exception: the §15.1 fallback report may omit phases_entered.

17.2 Serialization

PCC-JCS canonical bytes (§A). Report equality is byte equality.

18 Conformance

18.1 Vector Execution

Conformance vectors are manifest-listed artifacts. Execution order: sort by UTF-8 byte comparison of identifier strings, regardless of JSON source order. conformance_runner_v1.json MUST list required vector identifiers. Missing vector → fail:conformance_vector_missing.

18.2 Baseline Coverage

conformance_runner_v1.json MUST enumerate the required vectors. Baseline vectors MUST cover at minimum:

- Parse gate: all §7 failure codes, -0, surrogates, escape failures, precedence with co-present errors, int64 boundary values.
- Identifiers: grammar boundaries, non-ASCII rejection, "." and ".." rejection, input_id validation.
- Projection: truth table outcomes, UTF-8 errors, collision.
- Schema: forbidden keywords, vocabulary detection, $ref/$id resolution, pcc:hashField with conditional applicators.
- Normalization: missing-field violations, synthesis behavior.
- Ordering: tie-break with name-based stripping, presorted arrays, key extraction failures, extension key rejection, type errors.
- Extensions: mutation invariance, parse failures, size ceilings, name-based vs path-based scoping interaction.
- Hash views: cycle detection, containment, missing-pointer policies, wrong-type, out-of-bounds, dependency graph overlap, duplicate pointers, overflow index token handling.
- Trust anchor: hash mismatch, format validation, conflict scan, extension exclusion behavior.
- Manifest: canonical binding, required fields, namespace collision, semantic check ordering.
- Pipeline: report after halt, generation failure fallback.
- Execution: sandbox violations, undeclared output, allowlist validation, counter overflow.
- PCC-JCS: integer canonicalization at int64 boundaries.
- Pointers: invalid escapes, empty tokens, bare /, root pointer context restrictions.
- Error registry: completeness, entry structure.
- Meta-schemas: container compliance, duplicate $id.
- Multi-document: compound pointer identity.

19 Execution Contract

Execution receives only:

- Declared hash views (§14.3).
- Allowlisted non-extension subtrees, keyed by PCC Pointer and serialized via PCC-JCS. The allowlist is declared per-phase in checkpoint_schedule_v1.json. Allowlist pointers MUST satisfy §14.1, MUST NOT target /extensions/**, MUST NOT contain hash fields (§14.4 containment), and MUST NOT have decoded-token duplicates. Violations → fail:execution_allowlist_invalid.

Bundle format is defined in checkpoint_schedule_v1.json.

Forbidden resources: clocks, random number generators, process identifiers,

filesystem metadata, environment variables, network access, concurrency primitives. Access to any forbidden resource → terminal fail:execution_sandbox. Outputs MUST be declared in the manifest "outputs" section with hash bindings. Identifiers MUST satisfy §5. Undeclared output → terminal fail:output_undeclared.

19.1 Resource Limits

Deterministic step counters are defined per-phase in checkpoint_schedule_v1.json. All counters are int64. Overflow → fail:counter_overflow. Wall-clock timeouts MUST NOT affect deterministic outcomes.

Annex A PCC-JCS Canonicalization (normative)

PCC-JCS is a profile of RFC 8785 (JSON Canonicalization Scheme). All canonicalization operations in PCC (§6.3, §12.2, §14.3, §17.2, §19) MUST use PCC-JCS.

PCC-JCS modifies RFC 8785 as follows: numbers are serialized as exact base-10 signed integer strings matching the grammar -?(0|[1-9][0-9]*). Implementations MUST NOT use IEEE-754 floating-point conversion for number serialization.

All other RFC 8785 rules (object key ordering, string escaping, Unicode handling) apply without modification.

Implementations using generic RFC 8785 libraries MUST verify that the library enforces PCC's integer rendering rule. Libraries that apply floating-point number formatting do not conform.