

# RFC-033: Schema Evolution

**Status:** Draft **Date:** January 2026 **Author:** Derrell Piper ddp@eludom.net  
**Implementation:** Proposed

---

## Abstract

This RFC specifies schema evolution for the Library of Cyberspace: how soup metadata schemas change over time while maintaining backward compatibility, migration paths, and query consistency. Schemas are themselves content-addressed objects.

---

## Motivation

Data outlives code:

- **Long-term preservation** - Data must remain readable for centuries
- **Software updates** - New versions need new fields
- **Federation** - Different vaults may have different schema versions
- **Rollback** - Must handle downgrades gracefully

Schema evolution must be:

- **Non-breaking** - Old data remains accessible
  - **Bidirectional** - Upgrade and downgrade paths
  - **Explicit** - Changes documented and versioned
  - **Verifiable** - Schema validation at boundaries
- 

## Schema Model

### Schema Definition

```
(define-schema 'document
  (version 1)
  (fields
    (hash (type string) (required #t) (indexed #t))
    (name (type string) (required #t))
    (content-type (type string) (required #t))
    (size (type integer) (required #t))
    (created (type timestamp) (required #t))
    (modified (type timestamp) (required #f))
    (author (type principal) (required #f))
    (tags (type (list string)) (required #f) (default '()))))
```

## Schema as Content-Addressed Object

```
;; Schemas are soup objects
(soup-object
  (hash "sha256:schema-hash...")  

  (type schema)
  (name document)
  (version 1)
  (fields ...)
  (created "2026-01-01"))

;; Reference schema by hash for reproducibility
(define (validate-against-schema obj schema-hash)
  (let ((schema (soup-get schema-hash)))
    (validate-object obj schema)))
```

## Schema Registry

```
(define schema-registry (make-hash-table))

(define (register-schema! schema)
  (let* ((name (schema-name schema))
         (version (schema-version schema))
         (hash (soup-put schema type: 'schema)))
    (hash-table-set! schema-registry (cons name version) hash)
    (audit-append action: 'schema-registered
                  name: name
                  version: version
                  hash: hash)
    hash))

(define (get-schema name #!optional version)
  "Get schema by name and optional version"
  (if version
      (soup-get (hash-table-ref schema-registry (cons name version)))
      (get-latest-schema name)))
```

---

## Evolution Rules

### Compatible Changes

```
;; These changes are backward compatible
(define compatible-changes
  '(add-optional-field      ; New field with default
    add-index              ; New index on existing field
```

```

widen-type           ; integer -> number
relax-constraint     ; required -> optional
add-enum-value))    ; Extend enumeration

(define (change-compatible? old-schema new-schema)
  "Check if schema change is backward compatible"
  (let ((changes (diff-schemas old-schema new-schema)))
    (every compatible-change? changes)))

```

## Breaking Changes

```

;; These changes require migration
(define breaking-changes
  '(remove-field           ; Field no longer exists
    add-required-field     ; New required field without default
    narrow-type            ; number -> integer
    tighten-constraint     ; optional -> required
    remove-enum-value      ; Shrink enumeration
    rename-field))         ; Field name changed

(define (requires-migration? old-schema new-schema)
  "Check if migration needed"
  (let ((changes (diff-schemas old-schema new-schema)))
    (any breaking-change? changes)))

```

---

## Version Management

### Semantic Versioning

```

;; Schema versions follow semver
(define (schema-version-compare v1 v2)
  (let ((major1 (version-major v1)) (major2 (version-major v2))
        (minor1 (version-minor v1)) (minor2 (version-minor v2))
        (patch1 (version-patch v1)) (patch2 (version-patch v2)))
    (cond
      ((not (= major1 major2)) (- major1 major2))
      ((not (= minor1 minor2)) (- minor1 minor2))
      (else (- patch1 patch2)))))

;; Major: breaking changes
;; Minor: compatible additions
;; Patch: documentation/fixes

```

## Version History

```
(define (schema-history name)
  "Get all versions of a schema"
  (soup-query type: 'schema
    name: name
    order-by: '((version desc))))  
  
(define (schema-lineage name version)
  "Get evolution path to current version"
  (let ((schema (get-schema name version)))
    (if (schema-parent schema)
        (cons schema (schema-lineage name (schema-parent-version schema)))
        (list schema)))  
  
-----
```

## Migration

### Migration Definition

```
(define-migration 'document-v1-to-v2
  (from-schema 'document (version 1))
  (to-schema 'document (version 2))
  (up (lambda (obj)
        ;; Add new field with default
        (assoc-set obj 'visibility 'private)))
  (down (lambda (obj)
        ;; Remove new field
        (assoc-remove obj 'visibility))))
```

### Migration Execution

```
(define (migrate-object obj from-version to-version)
  "Migrate object between schema versions"
  (let ((migrations (find-migration-path from-version to-version)))
    (fold (lambda (migration obj)
            ((migration-up migration) obj))
          obj
          migrations)))  
  
(define (find-migration-path from to)
  "Find sequence of migrations between versions"
  (let ((direction (if (> to from) 'up 'down)))
    (filter (lambda (m)
              (and (eq? (migration-direction m) direction)
                   (version-in-range? (migration-from m) from to))))
```

```
(all-migrations))))
```

## Lazy Migration

```
; Migrate on read, not on upgrade
(define (soup-get-migrated hash)
  "Get object, migrating if necessary"
  (let* ((obj (soup-get-raw hash))
         (obj-version (object-schema-version obj))
         (current-version (current-schema-version (object-type obj))))
    (if (= obj-version current-version)
        obj
        (let ((migrated (migrate-object obj obj-version current-version)))
          ;; Optionally persist migrated version
          (when *persist-migrations*
            (soup-put-raw migrated))
          migrated))))
```

## Batch Migration

```
(define (batch-migrate schema-name from-version to-version)
  "Migrate all objects of a schema"
  (let ((objects (soup-query type: schema-name
                             schema-version: from-version)))
    (for-each (lambda (obj)
                (let ((migrated (migrate-object obj from-version to-version)))
                  (soup-update (object-hash obj) migrated)
                  (audit-append action: 'object-migrated
                               hash: (object-hash obj)
                               from: from-version
                               to: to-version)))
              objects)))
```

---

## Validation

### Schema Validation

```
(define (validate-object obj schema)
  "Validate object against schema"
  (let ((errors '()))
    ;; Check required fields
    (for-each (lambda (field)
                (when (and (field-required? field)
                           (not (assoc-ref obj (field-name field)))))
                  (set! errors (cons `(missing-required ,(field-name field))
```

```

                errors)))
(schema-fields schema))

;; Check field types
(for-each (lambda (pair)
  (let* ((name (car pair))
         (value (cdr pair))
         (field (schema-field schema name)))
    (when (and field (not (type-matches? value (field-type field))))
          (set! errors (cons `(type-mismatch ,name) errors))))
  obj)

(if (null? errors)
  (result-ok obj)
  (result-err errors)))

```

## Validation Modes

```

(define validation-mode (make-parameter 'strict))

(define (validate-with-mode obj schema)
  (case (validation-mode)
    ((strict)
     ;; All fields must match schema
     (validate-object obj schema))
    ((permissive)
     ;; Extra fields allowed
     (validate-required-fields obj schema))
    ((warn)
     ;; Log warnings but don't fail
     (let ((result (validate-object obj schema)))
       (when (result-err? result)
         (log-warn "Validation errors" errors: (result-error result)))
       (result-ok obj)))))



---



```

## Federation Compatibility

### Schema Negotiation

```

(define (negotiate-schema peer schema-name)
  "Negotiate compatible schema version with peer"
  (let* ((local-versions (local-schema-versions schema-name))
         (peer-versions (request-schema-versions peer schema-name))
         (compatible (intersection local-versions peer-versions)))
    (if (null? compatible)

```

```

(error 'no-compatible-schema)
(max compatible)))))

(define (sync-with-schema-version peer schema-name version)
  "Sync objects using negotiated schema version"
  (for-each (lambda (obj)
    (let ((converted (convert-to-version obj version)))
      (send-object peer converted)))
    (objects-to-sync schema-name)))

```

## Cross-Version Queries

```

(define (query-with-version-tolerance query)
  "Query that handles mixed schema versions"
  (let* ((results (soup-query-raw query))
         (current-version (current-schema-version (query-type query))))
    (map (lambda (obj)
      (if (= (object-schema-version obj) current-version)
          obj
          (migrate-object obj (object-schema-version obj) current-version)))
      results)))

```

---

## Schema Introspection

### Schema Queries

*;; What schemas exist?*  
 (soup-query type: 'schema)

*;; What versions of a schema?*  
 (soup-query type: 'schema name: 'document)

*;; What fields in a schema?*  
 (schema-fields (get-schema 'document 2))

*;; Schema dependency graph*  
 (define (schema-dependencies schema)
 (filter-map (lambda (field)
 (when (schema-reference? (field-type field))
 (field-type-target field)))
 (schema-fields schema)))

## Schema Documentation

```
(define-schema 'document
  (version 2)
  (description "A document stored in the vault")
  (fields
    (hash
      (type string)
      (required #t)
      (indexed #t)
      (description "Content-addressed hash of document"))
    (name
      (type string)
      (required #t)
      (description "Human-readable document name")
      (example "RFC-033-schema-evolution.md"))))
```

---

## Default Values

### Static Defaults

```
(define-schema 'document
  (version 2)
  (fields
    (visibility
      (type enum)
      (values (public private restricted))
      (default 'private))
    (tags
      (type (list string)))
      (default '()))))
```

### Computed Defaults

```
(define-schema 'document
  (version 2)
  (fields
    (created
      (type timestamp)
      (default (current-time)))
    (id
      (type string)
      (default (generate-uuid)))))
```

## Migration Defaults

```
; Default for objects migrated from v1 to v2
(define-migration 'document-v1-to-v2
  (defaults
    (visibility (lambda (obj)
      ; Infer from existing data
      (if (assoc-ref obj 'public)
        'public
        'private)))))
```

---

## Security Considerations

### Schema Tampering

```
; Schemas are content-addressed - tampering detectable
(define (verify-schema-integrity schema-hash)
  (let* ((schema (soup-get schema-hash))
         (computed-hash (content-hash (serialize schema))))
    (equal? schema-hash computed-hash)))
```

### Migration Authorization

```
; Migrations require capability
(spki-cert
  (issuer vault-admin)
  (subject schema-admin)
  (capability
    (action migrate-schema)
    (object (schema 'document)))
  (validity (not-after "2027-01-01")))
```

---

## References

1. Protocol Buffers - Updating A Message Type
  2. Avro Schema Evolution
  3. RFC-020: Content-Addressed Storage
  4. RFC-025: Query Language
- 

## Changelog

- 2026-01-07 - Initial draft

---

**Implementation Status:** Draft **Dependencies:** soup, cas **Integration:**  
Soup queries, federation, long-term preservation