Independent Submission
Internet-Draft
Intended status: Informational
Expires: 29 August 2026

M. Caldas
Independent

25 February 2026

CSV++ (CSV Plus Plus): Extension to RFC 4180 for Hierarchical Data
draft-mscaldas-csvpp-02

Abstract

This document specifies CSV++ (CSV Plus Plus), an extension to the Comma-Separated Values (CSV) format defined in RFC 4180. CSV++ adds support for repeating fields (one-to-many relationships) and hierarchicalcomponent structures while maintaining backward compatibility with standard CSV parsers. The extension uses declarative syntax in column headers to define array fields and nested structures, enabling representation of complex real-world data while preserving the simplicity and human-readability of CSV.

Table of Contents

1. Introduction

CSV++ extends the CSV format defined in [RFC4180] to support repeating fields (one-to-many relationships) and hierarchical component structures while maintaining backward compatibility with standard CSV parsers.

1.1. Motivation

Traditional CSV files represent flat, tabular data. However, real-world data often contains:

* Repeated values (e.g., multiple phone numbers for one person)

* Structured components (e.g., addresses with street, city, state, zip)

* Nested hierarchies (e.g., addresses with multiple address lines)

CSV++ addresses these limitations by introducing:

While formats like JSON, XML, and YAML excel at representing hierarchical data, they introduce complexity and redundancy that may not be warranted for moderately structured datasets. CSV++ occupies a middle ground by extending CSV's tabular simplicity with hierarchical capabilities, making it particularly suitable for:

* Data interchange where CSV is already established but structure is needed

* Spreadsheet applications where users expect tabular layouts

* Systems with existing CSV infrastructure that need enhanced capabilities

* Scenarios where human readability and editability in text editors is valued

* Applications requiring backward compatibility with legacy CSV parsers

CSV++ maintains CSV's core strengths - simple tooling, wide compatibility, and human-readable plain text - while addressing its limitations with hierarchical data through declarative header syntax.

1.2. When to Use CSV++

CSV++ is most appropriate for:

* Moderately structured data (1-3 levels of nesting)
* Environments where CSV is already the established interchange format
* Scenarios requiring backward compatibility with existing CSV infrastructure
* Applications that benefit from self-documenting tabular data with inline structure definitions
* Data that needs to be both machine-parseable and human-readable in plain text
* Large datasets where file size and bandwidth matter, as CSV's columnar format avoids repeating field names in every record (unlike JSON or XML)

For deeply nested hierarchical data (4+ levels), document-oriented formats like JSON or XML may provide better readability and tooling support. CSV++ aims to extend CSV's capabilities for moderately structured data while preserving its tabular nature, not to replace hierarchical data formats.

1.3. Design Principles

1. *Backward Compatibility:* Standard CSV parsers can read CSV++ files (though they won't interpret the enhanced structure)
2. *Self-Documenting:* Structure is defined in column headers
3. *Tabular Readability:* Data maintains a tabular layout suitable for spreadsheet viewing and editing, though deeply nested structures (3+ levels) may be more readable in hierarchical formats like JSON
4. *Explicit Over Implicit:* Delimiters are declared, not assumed

5. *Recursively Composable:* Structures can nest to any depth, though practical implementations SHOULD limit nesting to 3-4 levels for readability

1.4. Requirements Language

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, as shown here.

2. Conformance with RFC 4180

CSV++ files MUST conform to [RFC4180] with these specifications:

* Fields are separated by a delimiter (comma by default)

* Records are separated by line breaks (CRLF or LF)

* Fields containing special characters MUST be enclosed in double-quotes

* Double-quotes within quoted fields MUST be escaped by doubling: ""

* The first record MUST be a header row declaring all field names and their types. CSV++ does not support headerless files; a header row is REQUIRED in every CSV++ document.

* MIME type: text/csv

3. Field Separator Detection

The field separator character is detected using the same rules as [RFC4180]. Parsers SHOULD auto-detect the field separator by:

1. Scanning the first line (header row)

2. Tracking bracket depth: [] and ()

3. Identifying characters that appear outside brackets (depth = 0)

4. Selecting the most common such character as the field separator

5. Common candidates: , (comma), \t (tab), | (pipe), ; (semicolon)

The comma (,) is the conventional field separator for CSV++ files.

4. Array Fields (Repetitions)

4.1. Syntax

A field containing repeated values is declared in the header using square brackets:

```
column_name[delimiter]
column_name[]
```

Where:

* column_name - The name of the field
* [delimiter] - The character used to separate repeated values
* [] - Empty brackets use the default array delimiter (first-level arrays only; see below)

Delimiter Resolution:

1. If a delimiter is specified: phone[|] uses |
2. If empty brackets at the top (first) nesting level: phone[] uses the tilde (~) as the default delimiter
3. If empty brackets at any deeper nesting level: INVALID. Nested arrays MUST explicitly specify a delimiter.

The tilde (~) is the default array delimiter for first-level (top-level) arrays only. When an array appears nested inside a structure or another array, its delimiter MUST be explicitly declared and MUST differ from every delimiter already in use at enclosing levels. Omitting the delimiter (using []) is not permitted for nested arrays.

Rationale: because the default tilde (~) is already consumed as the outermost repetition separator, reusing it at an inner level would make the data ambiguous and unparseable. Requiring an explicit delimiter at each inner level keeps the format unambiguous and consistent with the general principle that every nesting level MUST use a distinct delimiter.

4.2. Examples

```
id,name,phone[|],email[;]
1,John,555-1234|555-5678|555-9012,john@work.com;john@home.com
2,Jane,555-4444,jane@company.com
```

Figure 1: Arrays with Explicit Delimiters

```
id,name,phone[],email[]
1,John,555-1234~555-5678~555-9012,john@work.com~john@home.com
2,Jane,555-4444,jane@company.com
```

Figure 2: Arrays with Default Delimiters

4.3. Empty Values

Empty values in repetitions are represented by consecutive delimiters:

```
id,tags[|]
1,urgent||priority
```

Figure 3

This represents three tags: "urgent", "" (empty), "priority"

4.4. Escaping

If a delimiter character appears within a leaf value, that leaf value MUST be quoted per [RFC4180]. Quoting rules and the definition of a leaf element are specified in Section 7.

5. Structured Fields (Components)

5.1. Syntax

A field containing structured components is declared using parentheses:

```
column_name[repetition_delim]component_delim(
    comp1 component_delim comp2 ...)
column_name[]component_delim(comp1 component_delim comp2 ...)
column_name[](comp1 component_delim comp2 ...)
column_name(comp1 component_delim comp2 ...)
```

Component Delimiter Resolution:

1. If specified before (: address^(...) uses ^

2. If omitted: address(...) uses the caret (^) as default delimiter

The caret (^) is recommended as the default component delimiter to avoid conflicts with common data characters.

5.2. Examples

```
id,name,geo^(lat^lon)
1,Location A,34.0522^-118.2437
2,Location B,40.7128^-74.0060
```

Figure 4: Simple Structure

```
id,name,address[~]^(street^city^state^zip)
1,John,123 Main St^Los Angeles^CA^90210~456 Oak Ave^New York^NY^10001
2,Jane,789 Pine St^Boston^MA^02101
```

Figure 5: Repeated Structures

6. Nested Structures

6.1. Recursive Composition

Structures can nest arbitrarily deep. Component names can themselves be arrays or structures. Within component names in (...), array and structure syntax applies recursively.

6.2. Examples

```
id,name,address[~]^(type^lines[;]^city^state^zip)
1,John,home^123 Main;Apt 4^LA^CA^90210~work^456 Oak^NY^NY^10001
```

Figure 6: Array Within Structure

```
id,location^(name^coords:(lat:lon))
1,Office^34.05:-118.24
2,Home^40.71:-74.00
```

Figure 7: Structure Within Structure

6.3. Delimiter Selection Guidelines

To maintain readability and parseability:

1. *REQUIRED:* Use different delimiters at each nesting level. Nested structures MUST use different component delimiters than their parent

2. Use visually distinct delimiters at each level

3. *Recommended progression:* ~ -> ^ -> ; -> :

4. Avoid using the field separator as a component delimiter

5. Document delimiter choices for complex schemas

6. *Recommendation:* Limit nesting to 3-4 levels maximum

7. Quoting and Escaping

7.1. Leaf Elements

A *leaf element* is a value that will not be further split by any array or component delimiter -- it is the innermost atomic unit at its position in the CSV++ hierarchy. Examples of leaf elements:

* The value of a simple field
* An individual item within an array (a single repetition after splitting on the array delimiter)
* An individual component value within a structure (a single component after splitting on the component delimiter)
* An individual item within a nested array or nested structure, once all enclosing levels have been split

RFC 4180 double-quote quoting MUST only be applied to leaf elements. Quoting a value that still contains unprocessed array or component delimiters causes those delimiters to be treated as literal characters, preventing the parser from splitting the value into its constituent parts. This MUST NOT be done.

7.2. Valid Quoting (Leaf-Level)

Quoting is valid when applied to an individual leaf value to escape a delimiter character that appears literally within that value:

```
id,notes[|]
1,First note|"Second note with | pipe"|Third note
```

Figure 8: Quoting an Individual Array Item (Leaf)

The second array item is a leaf; quoting it escapes the literal pipe. The outer pipe delimiters that separate the three items remain unquoted and function as separators.

```
id,address^(street^city^state^zip)
1,"123 Main St, Apt 4"^Springfield^IL^62701
```

Figure 9: Quoting an Individual Component Value (Leaf)

The street component is a leaf; quoting it escapes the comma within the street address.

7.3. Invalid Quoting (Non-Leaf)

Quoting MUST NOT be applied to a value that contains unprocessed array or component delimiters. The following examples are invalid CSV++ and MUST be rejected by parsers:

```
id,notes[|]
1,"First note|Second note|Third note"
```

Figure 10: Invalid: Quoting an Entire Array Field Value

The field value is quoted at the array level. The pipe characters are treated as literal data, so the parser sees a single note rather than three. This defeats the purpose of the array declaration and is invalid.

```
id,address^(street^city^state^zip)
1,"123 Main St^Springfield^IL^62701"
```

Figure 11: Invalid: Quoting a Structured Value

The entire structured value is quoted. The component delimiters are swallowed by the quote, so no components can be extracted. This is invalid.

```
id,address[~]^(street^city^state^zip)
1,"123 Main St^Springfield^IL^62701"~456 Oak Ave^New York^NY^10001
```

Figure 12: Invalid: Quoting an Array Item That Is Itself Structured

The first repetition is quoted at the structure level, preventing component splitting. To escape a literal tilde within a component leaf, quote only that leaf value.

8. Parsing

CSV++ parsers process files in two phases:

1. *Header Parsing:* Parse column headers to identify field types (simple, array, or structured) and extract delimiter information

2. *Data Parsing:* For each data row, split fields according to their declared type, respecting [RFC4180] quoting rules for nested delimiters

The ABNF grammar in Appendix A provides a formal specification. Implementations MUST handle arbitrary nesting depth up to their documented limits.

9. Implementation Considerations

9.1. Validation

Implementations SHOULD validate:

* Matching number of components across repeated structures
* Proper bracket nesting in headers
* Delimiter conflicts (same delimiter at multiple levels)
* MUST reject: Nested structures using the same component delimiter as their parent
* Reasonable nesting depth (recommend warning beyond 3-4 levels)

9.2. Limits

Implementations MAY impose reasonable limits on:

* Nesting depth (recommended minimum: 10 levels)
* Number of components per structure (recommended minimum: 100)
* Number of repetitions per array (recommended minimum: 1000)

10. MIME Type and File Extension

10.1. MIME Type

CSV++ files use the text/csv media type defined in [RFC4180].

10.2. File Extensions

* .csv - Standard extension (recommended for compatibility)
* .csvpp - MAY be used to explicitly indicate CSV++ format
* .csvplus - Alternative explicit extension

11. Security Considerations

CSV is a long-established and widely deployed format with well-known security considerations. As a result, most mature implementations already incorporate mitigations for common CSV-related risks. This specification builds on [RFC4180] and remains fully backward compatible, but introduces additional structural semantics that may increase parser complexity and therefore require corresponding care in implementations.

11.1. Injection and Interpretation Risks

Malicious data may attempt to inject delimiters or structural markers to influence parsing behavior. Implementations MUST respect [RFC4180] quoting rules. Delimiters and structural markers appearing within quoted fields MUST be treated as literal values.

The default delimiters defined by this specification are intentionally chosen to be neutral and to avoid characters commonly associated with executable or control semantics. In addition, the explicit declaration of any non-default delimiters in the header allows parsers to establish expectations up front, reducing the likelihood of delimiter injection or ambiguous interpretation.

As with traditional CSV, some spreadsheet applications interpret certain values (e.g. those beginning with "=", "+", "-", or "@") as formulas. This specification does not attempt to redefine or mitigate spreadsheet formula evaluation; producers and consumers SHOULD continue to apply established best practices when targeting such environments.

11.2. Complexity and Resource Exhaustion

Deeply nested, malformed, or highly repetitive structures may lead to excessive CPU or memory consumption during parsing.

Implementations SHOULD:

* Enforce configurable limits on nesting depth and repetition
* Enforce reasonable limits on field sizes and record length
* Fail fast on structurally invalid input
* Prefer streaming or incremental parsing for large files
* Validate headers and structural definitions before processing data rows

11.3. Mixed-Tool Interoperability

CSV++ files may transit through tools unaware of the extended semantics, potentially resulting in loss of structure or unintended reinterpretation. Implementations used in security-sensitive pipelines SHOULD explicitly validate inputs and avoid implicit trust when moving between CSV-aware and CSV++-aware tools.

11.4. Encoding Issues

Files SHOULD use UTF-8 encoding. Implementations SHOULD detect and handle encoding errors. A BOM (Byte Order Mark) MAY be present.

11.5. IANA Considerations

This document has no IANA actions.

CSV++ files use the text/csv media type as defined in [RFC4180]. The format is fully backward compatible with standard CSV parsers; implementations unaware of the extensions defined in this document will process CSV++ files as conventional CSV, ignoring extended semantics.

Change Log

Changes from -01 to -02:

* Header row is now REQUIRED (MUST) in all CSV++ documents; headerless files are not supported.

* Removed language suggesting the header row is optional (was: "MAY be a header record").

* Removed references to auto-identification of field types from data without a header.

* Clarified that the tilde (~) default array delimiter applies to first-level (top-level) arrays only. Nested arrays MUST explicitly specify a delimiter distinct from all enclosing levels; using empty brackets ([]) in a nested array is invalid.

* Added Quoting and Escaping section defining leaf elements. RFC 4180 quoting MUST only be applied to leaf elements (atomic values not further split by any delimiter). Quoting non-leaf values (entire arrays or structured fields) is explicitly invalid and MUST be rejected by parsers.

Changes from -00 to -01:

* Enhanced Motivation section to contrast with JSON/XML
* Added "When to Use CSV++" section
* Improved scaping example on 4.4
* Updated Security section to include CSV injection considerations.

References

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>.

[RFC4180] Shafranovich, Y., "Common Format and MIME Type for Comma-Separated Values (CSV) Files", RFC 4180, DOI 10.17487/RFC4180, October 2005, <https://www.rfc-editor.org/info/rfc4180>.

[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>.

Informative References

Appendix A. Grammar (ABNF)

```
csvpp-file     = header-row data-rows

header-row     = field *(field-sep field) CRLF
data-rows      = *(data-row CRLF)
data-row       = value *(field-sep value)

field          = simple-field / array-field /
                 struct-field / array-struct-field
simple-field   = name
array-field    = name "[" [delimiter] "]"
struct-field   = name [component-delim] "(" component-list ")"
array-struct-field = name "[" [delimiter] "]"
                     [component-delim] "(" component-list ")"

component-list = component *(component-delim component)
component      = simple-field / array-field /
                 struct-field / array-struct-field

name           = 1*field-char
field-char     = ALPHA / DIGIT / "_" / "-"
delimiter      = CHAR
component-delim = CHAR

value          = quoted-value / unquoted-value
quoted-value   = DQUOTE *(textdata / escaped-quote) DQUOTE
unquoted-value = *textdata
escaped-quote  = DQUOTE DQUOTE
textdata       = <any character except DQUOTE, CRLF, or field-sep>
```

Appendix B. Complete Examples

```
id,cust,items[~]^(sku^name^qty^price^opts[;]:(k:v))
1,Alice,S1^Shirt^2^20^sz:M;col:blu~S2^Pant^1^50^sz:32
```

Figure 13: E-commerce Order

Acknowledgments

This specification was inspired by the HL7 Version 2.x delimiter hierarchy and the need for a simple, human-readable format for hierarchical data that maintains compatibility with existing CSV tools.

Author’s Address

Marcelo Caldas
Independent
Roswell, Georgia
United States of America
Email: mscaldas@gmail.com