Modular Curriculum Format (MCF) 1.1

MCF Project

26 July 2026

Contents

Status

This document defines Modular Curriculum Format (MCF) 1.1. MCF is an open, human-readable source format for portable educational content. It standardizes authored educational intent, not a learning platform, user interface, learner record, or compiler output.

The key words **MUST**, **MUST NOT**, **REQUIRED**, **SHALL**, **SHALL NOT**, **SHOULD**, **SHOULD NOT**, **RECOMMENDED**, **NOT RECOMMENDED**, **MAY**, and **OPTIONAL** in this document indicate requirement levels.

MCF 1.1 is the current specification. The normative MCF 1.0 text, schemas, and compatibility fixtures are retained in versioned repository paths.

1. Design goals

MCF 1.1 has six goals:

1. provide a stable, internally consistent MCF 1.1 source contract;
2. make parsing and validation deterministic across independent implementations;
3. support portable courses, modules, lessons, question banks, and asset collections;
4. represent richer pedagogy, assessment, accessibility, localization, and reuse;
5. define safe interchange and extension boundaries; and
6. keep delivery, storage, accounts, analytics, and visual presentation outside the format.

2. Versions and compatibility

An MCF 1.1 package declares:

```
mcf: "1.1"
```

A conforming MCF 1.1 implementation MUST also read and validate packages declaring `mcf: "1.0"` according to the MCF 1.0 specification and schemas. It MUST NOT apply 1.1-only required fields or reinterpret 1.0 source using conflicting 1.1 rules. Diagnostics SHOULD use the current error-code catalog where an equivalent code exists.

A writer MAY preserve or emit MCF 1.0 only when explicitly requested. It MUST NOT silently change a package's declared version. Migrating source from 1.0 to 1.1 is a deliberate writer operation whose result MUST validate as 1.1.

MCF version strings are exact. A reader that does not support a declared version MUST report `MCF_VERSION_UNSUPPORTED`; it MUST NOT guess that a different version has equivalent semantics.

Additive changes may occur in a later 1.x version. Removing or reinterpreting existing valid source requires a new major version.

3. Package model

Every distributable unit is a package with a root `manifest.yaml`. The `kind` field identifies its entry model:

- `course`: an ordered collection of chapters;
- `module`: an ordered collection of lessons;
- `lesson`: one standalone `.mcf` lesson;
- `question_bank`: a reusable YAML question collection; or
- `asset_collection`: a reusable collection of declared assets.

For `mcf: "1.1"`, `kind` defaults to `course` when omitted.

3.1 Course package

```
course/
|-- manifest.yaml
|-- chapters/
|   `-- chapter-id/
|       |-- chapter.yaml
|       `-- lessons/
|           `-- lesson.mcf
`-- assets/
```

`manifest.yaml`, `chapters/`, and at least one declared lesson are required. `assets/` is optional. Ordering comes only from YAML declarations.

3.2 Module package

```
module/
|-- manifest.yaml
|-- lessons/
|   `-- lesson.mcf
`-- assets/
```

A module manifest contains an ordered non-empty `lessons` list. Each item has a `source` path relative to the package root.

3.3 Lesson package

```
lesson/
|-- manifest.yaml
|-- lesson.mcf
`-- assets/
```

A lesson manifest has an `entry` path identifying one `.mcf` file.

3.4 Question-bank package

```
bank/
|-- manifest.yaml
|-- questions.yaml
`-- assets/
```

A question-bank manifest has an `entry` path identifying a question-bank YAML file as defined in Section 18.

3.5 Asset-collection package

An asset-collection manifest declares one or more assets and has no content entry. It is intended for licensed, attributed, reusable media.

3.6 Package identity

Every package has an `id`. The pair `(id, version)` identifies a published package version. Authors SHOULD use a stable, globally controlled ID such as a reverse-domain identifier when packages will be distributed broadly:

```yaml
id: org.example.calculus-i
version: "1.1.0"
```

MCF does not operate a global package registry. Implementations MUST NOT assume that a short ID is globally unique.

4. YAML profile

MCF YAML documents and fenced YAML objects use the YAML 1.2 data model restricted to JSON-compatible values:

- mapping keys MUST be strings;
- duplicate mapping keys are invalid;
- custom tags, anchors, aliases, and merge keys are invalid;
- non-finite numbers are invalid;
- timestamps have no special MCF meaning and SHOULD be quoted strings;
- source MUST be UTF-8 without a byte-order mark; and
- readers MUST NOT execute or instantiate language-specific YAML objects.

Unknown core fields are invalid unless they occur inside an `extensions` value. Schemas define where `null` is allowed; otherwise a field MUST be omitted rather than set to `null`.

4.1 Rich-content scalar serialization

YAML decoding occurs before Markdown and TeX processing. A writer MUST serialize a rich-content string so that its decoded YAML value exactly equals the intended Markdown/TeX source. Writers SHOULD use a single-quoted scalar for single-line rich content containing TeX backslashes, and SHOULD use a literal block scalar, preferably `|-` when a trailing newline is not intended, for multiline or display-math rich content:

```yaml
explanation: '$12\times 12 = 144$.'
```

```yaml
explanation: |-
  $$
  \sqrt{x^2}=|x|
  $$
```

An equivalent double-quoted scalar is possible when every intended literal TeX backslash is YAML-escaped:

```yaml
explanation: "$12\\times 12 = 144$."
```

The following is a valid YAML scalar, but is a pitfall:

```yaml
explanation: "$12\times 12 = 144$."
```

It decodes `\t` as a tab and therefore does not contain the intended `\times` TeX command. Similar failure modes affect `\frac`, `\sqrt`, and `\%` in double-quoted scalars. A reader MUST pass the decoded scalar value into the rich-content pipeline without reconstructing, repairing, or inferring lost source escapes. Plain scalars may work, but SHOULD NOT be recommended for arbitrary rich content because YAML punctuation can change their interpretation.

5. Identifiers, paths, and references

5.1 Identifiers

Local identifiers MUST match:

```
[a-z][a-z0-9._-]*
```

Chapter and lesson IDs are unique within a course. Activity and question IDs are unique within a lesson. Option, matching-item, ordering-item, rubric, criterion, and level IDs are unique within their containing object. Asset IDs are unique within the package.

IDs SHOULD remain stable when titles or file names change.

5.2 Paths

Package paths:

- use forward slashes;
- are relative, never absolute;
- MUST NOT begin with a Windows drive prefix such as `C:/`;
- MUST NOT contain an empty segment, `.` segment, or `..` segment;
- MUST NOT contain a backslash or NUL;
- are resolved case-sensitively; and
- MUST resolve to a regular file or declared directory within the package.

A missing referenced local file makes the package invalid. URL references are allowed only where the relevant field permits them.

5.3 Package references

A relationship or dependency may identify a package by `id`, optional `version`, and optional `href`. An `href` may be a safe package-relative archive path satisfying Section 5.2 or an absolute `https` URL. Other URI schemes and arbitrary non-path strings are invalid. Network resolution is not required for source validation unless explicitly requested by the user.

6. Common metadata

The following optional metadata fields may occur on packages, chapters, lessons, and activities unless a schema narrows their use:

```yaml
description: "A first course in calculus."
authors:
  - "Example Author"
license: CC-BY-4.0
subjects:
  - mathematics
  - calculus
keywords:
  - derivatives
  - rates of change
level:
  label: undergraduate
  identifier: first-year
estimated_duration: PT45M
prerequisites:
  - statement: "Understand functions and algebra."
learning_outcomes:
  - id: derivative-definition
    statement: "Explain a derivative as an instantaneous rate of change."
    framework:
```

```yaml
    name: "Example Mathematics Framework"
    identifier: "CALC.DERIVATIVE.1"
    uri: https://example.org/outcomes/CALC.DERIVATIVE.1
```

`subjects` and `keywords` are non-empty lists of non-empty strings. `level` is either a non-empty string or an object containing a human-readable `label` and optional `identifier` and `uri`.

`estimated_duration` is an ISO 8601 duration string. Calendar years and months SHOULD NOT be used because their elapsed length is context dependent.

A prerequisite is an object with a required `statement` and optional `identifier`, `uri`, and `package`.

A learning outcome has a required `statement`, an optional local `id`, and an optional `framework`. A framework has a required `name` and optional `identifier` and `uri`. MCF does not define a universal subject, level, or curriculum vocabulary.

7. Relationships, dependencies, and localization

A manifest MAY declare relationships:

```yaml
relationships:
  - type: translation_of
    package: org.example.calculus-i
    version: "1.1.0"
    language: en-CA
    href: dependencies/calculus-i-en.mcf.zip
```

Core relationship types are:

- `translation_of`;
- `has_translation`;
- `is_version_of`;
- `has_version`;
- `derived_from`;
- `requires`; and
- `supplements`.

`package` is required. `version`, `language`, and `href` are optional.

Language values MUST be well-formed BCP 47 language tags under RFC 5646, including its grandfathered forms, extension structure, and private-use form. Underscore-separated locale identifiers are not language tags. Schema-aware validators MUST implement the `bcp47` format as grammar validation rather than substituting a loose locale regular expression. Readers MUST compare language tags case-insensitively and SHOULD preserve author casing. A translated package SHOULD preserve stable IDs for semantically corresponding content. Translation does not imply byte-for-byte structural identity.

Dependencies use `requires`. A package validator MUST report unresolved required dependencies when validation is performed in a closed package set. It MAY report them as unavailable rather than invalid during offline validation of one package.

8. Assets, attribution, and accessibility

A package MAY declare assets:

```yaml
assets:
  - id: velocity-video
    source: assets/video/velocity.mp4
    media_type: video/mp4
    title: "Velocity demonstration"
    description: "A cart accelerates along a measured track."
    creator: "Example Laboratory"
    license: CC-BY-4.0
    attribution_url: https://example.org/velocity
    integrity: sha256-BASE64_VALUE
    language: en
    captions:
      - source: assets/captions/velocity-en.vtt
        language: en
        kind: captions
    transcript:
      source: assets/transcripts/velocity.md
      language: en
    audio_description:
      source: assets/audio/velocity-description.mp3
      language: en
```

Required asset fields are `id` and `source`. `media_type` SHOULD be an IANA media type. `integrity`, when present, uses Subresource Integrity syntax and MUST match the packaged bytes.

Caption entries require `source`, `language`, and `kind`. Core kinds are `captions`, `subtitles`, and `descriptions`. Transcript and audio-description objects require `source` and MAY declare `language`.

An `alternates` entry identifies another packaged rendition of the same educational asset. It requires `source` and `purpose`; optional fields are `language`, `media_type`, and `description`. Core purposes are `accessible_alternative`, `low_bandwidth`, `print`, and `transcoded`. An alternate is not a translation relationship between packages.

Every local file referenced by an asset record MUST exist. Asset records do not replace Markdown alternative text. An informative image MUST still have useful alt text at each use site; decorative images use empty alt text.

Rich content may refer to a declared asset as `asset:ASSET_ID` or use a relative file path. A declared asset enables attribution and alternatives to travel with the source.

9. Course, chapter, and lesson structures

9.1 Course manifest

```yaml
mcf: "1.1"
kind: course
id: intro-physics
title: "Introduction to Physics"
language: en
version: "1.1.0"
chapters:
  - source: chapters/motion
```

Required course fields are `mcf`, `id`, `title`, `language`, and `chapters`. `kind` defaults to `course`. Each chapter item contains a `source` directory under `chapters/`.

9.2 Chapter

```yaml
id: motion
title: "Motion"
description: "Distance, velocity, and acceleration."
lessons:
  - lessons/01-introduction.mcf
  - lessons/02-velocity.mcf
```

Required fields are `id`, `title`, and a non-empty ordered `lessons` list. Lesson paths are relative to the chapter directory.

9.3 Lesson

A `.mcf` lesson is UTF-8 Markdown with YAML frontmatter followed by one or more activities:

```
---
id: velocity
title: "Velocity"
---

:::mcf-activity
type: notes
id: velocity-notes
title: "Understanding velocity"
:::

Velocity is displacement over time.

:::mcf-end
```

Required frontmatter fields are `id` and `title`. Common metadata, `rubrics`, `completion`, `relationships`, and `extensions` are optional.

10. Lesson lexical grammar

The following grammar is normative together with the rules below:

```
lesson          = frontmatter, separation, activity,
                  { separation, activity }, trailing ;
frontmatter     = delimiter, newline, yaml-lines, delimiter, newline ;
activity        = activity-open, newline, activity-yaml,
                  header-close, newline, activity-body, activity-end ;
activity-open   = ":::mcf-activity" ;
header-close    = ":::" ;
activity-end    = ":::mcf-end" ;
delimiter       = "---" ;
separation      = { blank-line | comment } ;
trailing        = { blank-line | comment } ;
newline         = LF ;
```

Additional rules:

1. Lines are normalized from CRLF to LF before parsing. Lone CR is invalid.
2. The first line MUST be `---`; leading content and UTF-8 BOMs are invalid.
3. Delimiter and container-marker lines have no leading or trailing characters.
4. The frontmatter closes at the next exact `---` line.
5. The activity header closes at the next exact `:::` line.
6. The activity body ends at the next exact `:::mcf-end` line that is not inside a fenced code block.
7. CommonMark fenced-code rules determine whether a marker is inside a fence.
8. Activities cannot nest.
9. Outside frontmatter and activities, only blank lines and complete HTML comments are permitted. Comments do not carry MCF semantics.
10. An unterminated frontmatter, activity header, code fence, or activity is invalid.
11. A literal marker at the start of a body line MUST be placed in a fenced code block or written using a character reference or other semantically equivalent Markdown representation.

The YAML portions are interpreted using Section 4. Rich-content parsing begins only after structural containers and `mcf-question` fences have been identified.

11. Activities

MCF 1.1 activity types are:

- `notes`: instructional content;
- `practice`: formative work;
- `assessment`: evaluative work; and

- `assignment`: work intended for learner submission.

Every activity requires `type` and `id`; `title` and common metadata are optional.

Practice and assessment activities MAY define:

```yaml
randomize: true
question_pool_size: 5
```

`question_pool_size` is a positive integer no greater than the number of questions in the activity. Randomization changes delivery order, never authored identity or answer semantics.

Assessment activities MAY define `passing_score` from 0 through 1.

Assignment activities require:

```yaml
submission:
  modes: [text, file]
```

Core submission modes are `text`, `file`, and `url`. Optional fields are `minimum_files`, `maximum_files`, `accepted_media_types`, and `maximum_file_size`. A maximum size is a positive integer byte count. An assignment MAY reference a rubric ID and SHOULD set `evaluation: manual`. `submission` and an activity-level `rubric` are valid only on `assignment`. When both file limits occur, `minimum_files` MUST NOT exceed `maximum_files`.

MCF defines expected submission shape, not upload transport, storage, identity, deadlines, or gradebook behavior.

12. Rich content and media

Rich content uses CommonMark-compatible Markdown. Readers MUST support paragraphs, headings, emphasis, lists, links, block quotes, fenced code, and images. Readers SHOULD support GitHub-style tables.

Math uses `$...$` inline and `$$...$$` for display math. Math is identified from rich-content source before Markdown transformations are applied to the contents of the math span. The contents of a recognized inline or display math span are opaque TeX source for Markdown purposes. Readers MUST NOT remove or reinterpret TeX backslashes inside recognized math spans. Readers MUST preserve the original TeX payload even if they cannot render it; see Section 4.1 for YAML scalar serialization guidance.

Images use Markdown:

```
![Graph](../../../assets/images/graph.svg)
![Graph](asset:graph)
```

Audio and video use:

```
@[audio](../../../assets/audio/example.mp3 "Example")
@[video](../../../assets/video/example.mp4 "Example")
@[video](asset:velocity-video "Velocity demonstration")
@[video](youtube:VIDEO_ID "Online video")
```

The media target is a relative path, declared asset reference, or registered provider reference. MCF 1.1 registers only `youtube:VIDEO_ID`. Unknown provider schemes require an extension.

Raw HTML is untrusted rich content. A reader MAY sanitize or omit it and MUST NOT execute scripts merely because they occur in MCF source. Writers SHOULD prefer portable Markdown.

Media may occur wherever rich content is allowed, including prompts, option text, hints, explanations, feedback, rubric descriptions, and assignment instructions.

13. Question objects

Questions occur in `mcf-question` fenced blocks:

````
```mcf-question
id: q1
type: multiple_choice
prompt: "Which quantity includes direction?"
options:
  - id: speed
    text: "Speed"
  - id: velocity
    text: "Velocity"
    feedback: "Correct: velocity is a vector."
answer: velocity
hint: "Think about vectors."
explanation: "Velocity includes magnitude and direction."
points: 1
required: true
evaluation: automatic
```
````

The fence content is one YAML mapping. Common required fields are `id`, `type`, and `prompt`.

Common optional fields:

- `hint`;
- `explanation`;
- `points`, a non-negative number defaulting to `1`;
- `required`, a boolean defaulting to `true`;
- `evaluation`;
- `learning_outcomes`, a list of local outcome IDs; and
- `extensions`.

`prompt`, `hint`, `explanation`, option `text`, and `feedback` are rich-content strings. A block scalar is recommended for multiline content.

Evaluation modes are:

- `automatic`: evaluated from an objective answer;
- `manual`: evaluated by a person or external process;
- `completion`: evaluated only against response-completion requirements; and
- `ungraded`: collected without correctness or completion evaluation.

Default evaluation is `automatic` for objective types, `manual` for `essay`, `ungraded` for `open_response`, and `manual` for assignment submissions.

13.1 Evaluation matrix

Question evaluation modes are normative:

Question type	Permitted modes	Default
`multiple_choice`	`automatic`	`automatic`
`multiple_select`	`automatic`	`automatic`
`true_false`	`automatic`	`automatic`
`numeric`	`automatic`	`automatic`
`short_answer`	`automatic`	`automatic`
`matching`	`automatic`	`automatic`
`ordering`	`automatic`	`automatic`
`essay`	`manual`, `completion`	`manual`
`open_response` without completion fields	`ungraded`	`ungraded`
`open_response` with completion fields	`completion`	`completion`

An essay referencing a rubric MUST use `manual`. An explicit mode that conflicts with this matrix is invalid.

Activity evaluation modes are:

Activity type	Permitted modes	Default
`notes`	`ungraded`	`ungraded`
`practice`	`automatic`, `manual`, `completion`, `ungraded`	derived
`assessment`	`automatic`, `manual`, `completion`	derived
`assignment`	`manual`, `completion`, `ungraded`	`manual`

For practice and assessment, an omitted mode is derived after question resolution: `manual` if any required question is manual; otherwise `completion` if any required question is completion-based; otherwise `automatic`. An activity explicitly marked `ungraded` has no score.

14. Question types

14.1 Multiple choice

`multiple_choice` requires at least two options and `answer`, which is one option ID. Option IDs are unique. An option requires `id` and `text` and MAY include `feedback`.

A multiple-choice option MAY define `weight` from 0 through 1. When any option has a weight, every option MUST have one and the answer option MUST have weight `1`. Selecting an option awards `weight * points`. Without weights, the answer option awards all points and every other option awards zero. Weights are not valid on multiple-select options, whose partial-credit algorithm is defined below.

14.2 Multiple select

`multiple_select` requires at least two options and a non-empty, duplicate-free `answer` list containing option IDs. Answer order is insignificant.

`scoring` is `exact` or `partial`, defaulting to `exact`. Exact scoring awards all points only when selected IDs equal answer IDs.

Partial scoring is:

```
max(0, correct_selected / correct_total
       - incorrect_selected / incorrect_total) * points
```

When there are no incorrect options, the second fraction is zero. A reader MUST not round the result except for display.

14.3 True/false

`true_false` requires a boolean `answer`.

14.4 Numeric

`numeric` requires a numeric `answer`. It MAY define:

```yaml
unit: m/s
tolerance:
  absolute: 0.1
  relative: 0.01
```

As a shorthand, `tolerance` MAY be a non-negative number and means absolute tolerance.

A response is correct when either enabled test succeeds:

```
absolute: abs(response - answer) <= absolute
relative: abs(response - answer) <= relative * abs(answer)
```

If both are present, satisfying either is sufficient. When the answer is zero, relative tolerance alone is invalid. Default absolute tolerance is zero.

`unit` is a trimmed, case-sensitive expected label. MCF 1.1 does not define unit conversion. A reader MUST NOT treat a numerically equal response with a different supplied unit as correct unless an extension defines conversion.

14.5 Short answer

`short_answer` requires exactly one of:

```yaml
answer: "MCF"
```

or:

```yaml
answers:
  - "MCF"
  - "Modular Curriculum Format"
```

An answer matches when it matches any accepted answer after normalization:

```yaml
normalization:
  trim: true
  case_sensitive: false
  collapse_whitespace: false
  unicode: NFC
```

Defaults are: trim is true, case sensitivity is false, whitespace collapsing is false, and Unicode normalization is NFC. Allowed Unicode values are `NFC`, `NFD`, `NFKC`, `NFKD`, and `none`.

14.6 Essay

`essay` represents manually evaluated extended writing. It MUST NOT define an objective answer. It MAY reference a rubric and MAY define completion requirements from Section 14.7. Default evaluation is `manual`.

14.7 Open response

`open_response` represents a reflection, survey response, prediction, journal entry, or other response without an objectively correct answer. It MUST NOT define an answer or rubric.

It MAY define:

- `minimum_words`;
- `minimum_sentences`;
- `keywords`; and

- `minimum_keywords`.

These are completion conditions only. If any condition is present, evaluation defaults to `completion`; otherwise it defaults to `ungraded`.

For both `essay` and `open_response`, word, sentence, and keyword counting MUST follow the conformance algorithm:

1. normalize text to Unicode NFC;
2. replace Markdown syntax characters with spaces while retaining text;
3. split words at Unicode whitespace and punctuation;
4. discard empty tokens;
5. compare keywords using Unicode case folding; and
6. count a sentence when a non-empty run ends in `.`, `!`, or `?`.

Implementations MAY display richer language-aware estimates but conformance results use this algorithm.

14.8 Matching

`matching` requires at least two `premises`, at least two `responses`, and an `answer` mapping from every premise ID to one response ID:

```yaml
type: matching
prompt: "Match each term."
premises:
  - id: velocity
    text: "Velocity"
responses:
  - id: vector
    text: "Magnitude and direction"
answer:
  velocity: vector
scoring: partial
reuse_responses: false
```

Premise and response IDs are unique within their respective lists. `reuse_responses` defaults to false. When false, answer response IDs cannot repeat. Exact scoring requires every pair to match. Partial scoring awards `correct_pairs / total_pairs * points`.

14.9 Ordering

`ordering` requires at least two items and an `answer` list containing every item ID exactly once:

```yaml
type: ordering
prompt: "Order the steps."
items:
  - id: simplify
```

```
    text: "Simplify."
  - id: limit
    text: "Take the limit."
answer: [simplify, limit]
scoring: partial
```

Exact scoring requires the complete order. Partial scoring awards `correct_positions / total_items * points`.

15. Feedback and scoring

`hint` is guidance available before correctness is known. `explanation` explains the authored answer or evaluation intent. Option or item `feedback` is associated with that response.

Implementations choose when feedback is displayed, but MUST preserve its association and MUST NOT present feedback attached to an unselected option as though the learner selected it.

Points are authored weights, not gradebook records. An activity score is:

```
sum(question earned points) / sum(question available points)
```

Questions with `points: 0` do not affect the denominator. An activity with zero available points has no numeric score. Manual or pending evaluations prevent a final activity score unless the implementation clearly reports a provisional score.

16. Rubrics

Rubrics may be declared in a package manifest or lesson frontmatter:

```
rubrics:
  - id: mathematical-explanation
    title: "Mathematical explanation"
    criteria:
      - id: correctness
        description: "The mathematics is correct."
        levels:
          - id: complete
            description: "Correct and fully justified."
            points: 4
          - id: partial
            description: "Mostly correct."
            points: 2
          - id: insufficient
            description: "Substantial errors."
            points: 0
```

A rubric requires `id`, `title`, and at least one criterion. A criterion requires `id`, `description`, and at least two levels. A level requires `id`, `description`, and non-negative `points`.

Within one criterion, level points MUST be unique. A rubric score is the sum of the selected level points, at most one level per criterion. MCF defines the rubric and calculation, not reviewer identity, comments, moderation, or storage.

A question or assignment refers to a rubric by local ID. Every reference MUST resolve in the nearest lesson scope or package scope; lesson rubrics shadow package rubrics with the same ID.

17. Completion rules

Completion rules express author intent, never learner state. In MCF 1.1 they MAY occur only in lesson frontmatter, so every activity and question reference is resolved within that lesson:

```yaml
completion:
  all:
    - activity: introduction-notes
      requirement: viewed
    - activity: derivative-practice
      requirement: attempted
    - activity: derivative-assessment
      requirement: passed
      minimum_score: 0.7
```

A completion expression contains exactly one of `all` or `any`, each a non-empty list of conditions. Nested expressions are allowed to a maximum depth of eight.

A condition identifies one lesson-local `activity` or `question` and one requirement:

- `viewed`;
- `attempted`;
- `answered`;
- `submitted`;
- `passed`; or
- `manually_marked_complete`.

`minimum_score` is from 0 through 1 and is allowed only with `passed`. The target ID MUST resolve within the containing lesson. Package- and chapter-level completion expressions are not defined in MCF 1.1.

The format does not define event collection, learner identity, progress storage, unlocking, deadlines, overrides, or synchronization.

18. Question banks

A question-bank entry is a YAML mapping:

```yaml
id: calculus-basics
title: "Calculus basics"
questions:
  - id: limit-001
    type: numeric
    prompt: "Evaluate the limit."
    answer: 2
```

Required fields are `id`, `title`, and a non-empty `questions` list. Questions use the same objects as `mcf-question` fences.

An activity can reference bank questions:

````
```mcf-question-ref
bank: org.example.calculus-basics
question: limit-001
```
````

`bank` identifies a declared `requires` relationship or a question-bank package available in the package set. `question` is the stable bank-local ID.

References are resolved before randomization. Readers MUST preserve the bank and question identities in normalized models and diagnostics. An unresolved reference is invalid during closed-set validation.

19. Extensions

MCF 1.1 supports namespaced extensions without allowing them to redefine core meaning:

```yaml
extensions:
  org.example.geogebra:
    required: false
    data:
      source: assets/constructions/triangle.ggb
```

Extension keys MUST be lowercase reverse-domain names with at least three dot-separated segments. An extension value requires `required` and `data`.

A reader:

- MUST preserve unknown extensions when performing a lossless round trip;
- MAY ignore an unknown extension with `required: false` after reporting an informational diagnostic;
- MUST stop semantic processing of the containing object and report `MCF_EXTENSION_REQUIRED_UNSUPPORTED` for an unknown required extension; and
- MUST NOT allow an extension to change validation or meaning of core fields.

Executable extension content remains subject to the security rules.

20. Capability declarations

An implementation MAY publish a capability document:

```yaml
mcf_capabilities: "1.1"
implementation:
  name: Example Reader
  version: "2.0.0"
mcf_versions: ["1.0", "1.1"]
conformance:
  - validator
  - reader
package_kinds:
  - course
  - module
  - lesson
question_types:
  - multiple_choice
  - numeric
  - open_response
features:
  - archive
  - accessibility_assets
extensions:
  - org.example.geogebra
```

A capability declaration reports support; it does not change package validity. Advertising a conformance class asserts support for all mandatory behavior of that class and version. Partial implementations MUST list concrete capabilities and MUST NOT claim full conformance.

Because 1.0 reading and validation are mandatory for every conforming MCF 1.1 implementation, a capability document claiming any conformance class MUST list both `"1.0"` and `"1.1"` in `mcf_versions`.

21. Archive convention

The standard archive suffix is `.mcf.zip`. The ZIP root contains `manifest.yaml` directly.

Archive entries:

- use UTF-8 names and forward slashes;
- MUST satisfy Section 5 paths;
- MUST NOT be absolute, duplicated, encrypted, symlinks, hard links, devices, sockets, or other special files;
- MUST NOT escape the extraction root after normalization;
- MUST have matching local-header and central-directory names; and
- MUST be rejected if their declared or actual expansion exceeds an implementation's documented resource limits.

Readers MUST validate paths before extraction. Implementations SHOULD process archives without writing untrusted entries to disk and SHOULD document limits for entry count, individual size, total expanded size, and compression ratio.

For deterministic conformance testing, the canonical package-reader profile rejects an archive exceeding any of: 4,096 entries, 64 MiB for one expanded entry, 512 MiB total expanded bytes, or a 200:1 per-entry compression ratio. Implementations MAY enforce stricter limits. If a trusted deployment explicitly raises a limit, it MUST still recognize the canonical over-limit fixture and be able to report `MCF_ARCHIVE_LIMIT_EXCEEDED` when tested with the canonical profile.

Archive entry order has no semantic meaning. Source ordering remains declarative.

22. Security and trust

Valid MCF source is not automatically trusted content.

Readers and compilers:

- MUST prevent package-root escape;
- MUST NOT execute raw HTML scripts, embedded code, SVG scripts, or extensions by default;
- MUST treat `javascript:`, `data:text/html`, `file:`, and equivalent active URI schemes as unsafe;
- MUST sanitize generated HTML;
- MUST require an explicit user or administrator policy before fetching remote resources;
- MUST bound YAML nesting, archive expansion, file sizes, and parser work; and
- SHOULD verify declared integrity before using an asset.

An implementation MAY reject content according to a stricter security policy. It MUST distinguish a source-invalid diagnostic from a policy rejection.

23. Validation and error reporting

A valid MCF 1.1 package has:

- valid YAML under Section 4;
- the required layout and fields for its package kind;
- resolvable and safe local paths;
- valid BCP 47 languages and declared metadata;
- valid lesson grammar;
- supported core activity and question types;
- unique, well-formed IDs;
- valid type-specific answers and scoring;
- resolved local rubrics, completion targets, assets, and closed-set references;
- valid extension declarations; and
- a manifest whose declared order is preserved.

Diagnostics have:

```yaml
code: MCF_QUESTION_DUPLICATE_OPTION_ID
severity: error
message: "Option ID 'a' occurs more than once."
file: chapters/test/lessons/test.mcf
line: 14
column: 3
object_id: q1
```

`code` and `severity` are portable. Message wording and source location precision are implementation-specific. Core error codes are defined in `conformance/error-codes.yaml`.

Severity values are `error`, `warning`, and `info`. A package is invalid when one or more `error` diagnostics are produced.

24. Conformance classes

24.1 Validator

A conforming validator MUST parse every core package kind, apply every normative validation rule, and emit applicable standard error codes. It need not render content.

24.2 Reader

A conforming reader MUST perform validator behavior, preserve ordering and identity, expose all core semantic fields, and avoid silently discarding unsupported required content.

24.3 Writer

A conforming writer MUST produce valid source, stable non-conflicting IDs, deterministic ordering, declared dependencies, and lossless preservation of unknown optional extensions when editing existing content.

24.4 Package reader

A conforming package reader MUST safely process directory and `.mcf.zip` packages, enforce Section 21, and provide identical source semantics for both.

24.5 Full implementation

A full implementation satisfies validator, reader, writer, and package-reader requirements. Rendering, compilation, or learner-state support is not required for full format conformance.

25. Canonical conformance materials

The repository's `fixtures/valid`, `fixtures/invalid`, schemas, error catalog, and suite manifest are normative conformance materials. When prose, schema, and fixture behavior appear to conflict, implementations MUST report the conflict; the normative specification text controls until the repository is corrected.

Schemas validate data shape. Rules involving file resolution, uniqueness across files, Markdown structure, scoring, security policy, or reference graphs require semantic validation.

26. Out of scope

MCF 1.1 does not define:

- generated HTML, CSS, JavaScript, or compiler output;
- visual themes or interaction design;
- learner accounts or identity;
- stored responses, progress, attempts, or grades;
- classrooms, rosters, gradebooks, or institutional roles;
- deadlines assigned to particular learners or cohorts;
- analytics, recommendations, notifications, or payments;
- certificates or badges;
- synchronization or cloud storage;
- AI tutor behavior;
- code execution or laboratory simulation runtimes; or
- accessibility conformance of a particular rendered application.

MCF carries portable author intent. Implementations control delivery and operations.