Article Standards & Template Pack

What this is. The typed foundation of your system of record: five article templates with a governance-grade metadata schema, gold-standard examples, and the writing standards that make knowledge work for human *and* machine readers.

When you need it. Right after your Module 1 placement, if you're building fresh (Level 0–1). In an afternoon of mapping, if you already have KCS article standards (Level 2) — your existing structure survives; it gains typing, metadata, and machine-legibility rules.

Which maturity level it serves. Installs the Level 1→2 floor; required substrate for everything above it.

1. Why typed articles

An untyped knowledge base is a pile of pages. A typed one is a system: every article declares what kind of thing it is, and that declaration drives everything downstream — which template disciplines its structure, which checks gate its approval, how it chunks for retrieval, when it comes up for review, and how an AI is allowed to use it.

Five types cover a support organization's knowledge. Resist inventing more until these overflow:

Type	The question it answers	Typical origin
Issue/ Resolution	"Something is wrong — how do I fix it?"	A resolved case (the KCS heartbeat)
How-To	"How do I accomplish this task?"	Recurring guidance requests; product launches
FAQ	"What's the quick answer to this common question?"	Question clusters with short, stable answers
Known Error	"Is this a bug? What do I do until it's fixed?"	Confirmed defects with workarounds
Candidate	(not knowledge yet) — "the AI proposes this is worth keeping"	The Capture Loop; exists only to be curated

The Candidate is the type most teams have never seen, and it's the one that marks this as an AI-era system: a structured proposal that carries its own provenance and demand evidence, waiting at the gate. Candidates live in a queue, never in the knowledge base proper.

2. Anatomy: frontmatter is governance, body is payload

Every article is one file. The YAML frontmatter carries everything the *system* needs — identity, type, ownership, applicability, lifecycle. The Markdown body carries everything the *reader* needs, in the type's fixed sections.

This split is load-bearing. Retrieval filters on frontmatter *before* ranking by relevance (a draft article is physically incapable of grounding an answer). Review cadences read `volatility`. Ownership reports read `owner`. The DIFF gate reads `version`. None of that works if governance data lives as prose.

The full field-by-field reference is `metadata-schema.yaml` in this module. The rules that matter most:

- `id` **is stable and never reused.** `KO-000142` means the same article forever, across renames, moves, and platform migrations. Citations depend on this.
- `status` **has exactly three values** — `draft`, `validated`, `archived` — and only `validated` content may ground an answer or be shown as trusted. There is no "mostly done."
- `owner` **is never empty.** A team name is fine; a blank is an orphan, and orphans are how corpora rot.
- **Applicability is metadata, not prose.** `product`, `product_version`, `domain`, `audience` live in frontmatter where a retrieval layer can filter on them — not buried in a sentence.
- `volatility` **(** `evergreen` / `medium` / `high` **) drives review cadence.** Set it at creation; Module 5 turns it into a schedule.
- `source` **records provenance** — the case, conversation, or candidate the article came from. Trust is architectural.

3. Writing standards: sufficient to solve, in the customer's language

The classical KCS standards survive intact:

- **Capture the customer's phrasing.** The article should contain the words a person actually used ("screen goes black when I share"), because that's what the next person — or the retrieval layer — will search. Do not sanitize into support-speak.
- **Sufficient to solve, not a novel.** Enough context to recognize the issue and enough steps to resolve it. If a section doesn't help someone solve or decide, cut it.
- **One issue per article.** A grab-bag article can't be cited, can't be trusted, and can't be retired cleanly. Split it.

4. Your next reader is an AI

New standards, for the reader that assembles answers out of your articles' parts. Seven rules:

1. **Sections must survive amputation.** Retrieval takes sections, not whole articles. Each section must make sense alone: no "as mentioned above," no "see the previous step," no pronouns whose referent lives in another section.

2. **Resolution steps are complete in themselves.** The Resolution section restates what's being resolved in its first line. A reader landing there — human or machine — needs zero scrollback.
3. **Applicability is explicit and structured.** Which product, which versions, which plan tier, which platform — in frontmatter, filterable. An answer engine that can't tell whether an article applies will guess, and the Precision Paradox says its confident guess costs you trust.
4. **Stable IDs make citations mean something.** `KO-000142` cited today must resolve tomorrow. Never recycle IDs; archive, don't delete.
5. **Freshness is declared, not implied.** `last_validated` plus `volatility` lets a retrieval layer prefer current truth over archived truth — and lets it *say so* in the answer.
6. **Use the type's fixed section headers, always.** `## Environment`, `## Issue`, `## Cause`, `## Resolution` (and their per-type equivalents) are not style preferences — they're the seams a chunker splits on and the labels retrieval trusts. Consistency here is what "structure survives chunking" means.
7. **State the negative space.** If a fix does *not* apply to some version or configuration, say so explicitly. Machines don't infer exceptions; they need them written down.

The published standard: MRK-1.0

The machine-facing core of these rules is published as **Machine-Ready Knowledge (MRK-1.0)** — an actual JSON Schema (2020-12), served at `machinereadyknowledge.com/mrk-1.0.schema.json`, with the standard's prose at machinereadyknowledge.com. The relationship between the two contracts:

- `metadata-schema.yaml` **(this module) is the operational contract** — the richer field set your program *runs* on: status, volatility, provenance, review signals.
- **MRK-1.0 is the interchange projection** — what "machine-ready" means when an article leaves home: an export, a partner feed, a surface your customers' AI agents read.

The projection is mechanical:

Operational (this module)	MRK-1.0	Note
`id: KO-000142`	`id: ko-000142`	lowercase on export; MRK ids match `^[a-z][a-z0-9-]*-[0-9]+$`
`status: validated`	`lifecycle: current`	only `validated` articles export — drafts have no MRK form
`status: archived`	`lifecycle: retired` — or `superseded`, with the successor's `supersedes` naming the old id	rule 10's "new article with a link," formalized
`last_validated`	`verified`	
`product`, `product_version`, `audience`	`applies_to: {plan, region, version}`	applicability stays structured, never prose
`owner`, `title`, `type`	`owner`, `title`, `type`	unchanged

Operational (this module)	MRK-1.0	Note
Rule 1 (sections survive amputation)	`sections[].self_contained: true`	a **human attestation**, made at the curation gate (Module 4) — never inferred

Structural conformance is checkable with the standard `check-jsonschema` CLI against the served schema. Conformance attests structure; the editorial half — sections that truly stand alone, ids that never move — is a process property, and the gates are what make it true.

5. Stable IDs and linking

- **Format**: `KO-` + a six-digit, zero-padded, monotonically increasing number (`KO-000142`). Candidates use `CAND-` + four digits; they get a fresh `KO-` id if promoted.
- **Assignment**: from a single counter you control (a text file in the repo, a KB auto-number — anything that never repeats). Ranges per team if you need parallel assignment (team A: 0–499999, team B: 500000+).
- **Linking**: link between articles by ID in the `links:` frontmatter list, with an optional relation note (`KO-000188 # supersedes`). Explicit links are what make impact analysis possible later ("what references the feature we just changed?") — this is rung-2 muscle you start building on day one, for free.

6. Quality bar, by section

For Issue/Resolution (the workhorse — per-type variants in each template):

Section	Passes when	Fails when
Environment	Product, version, platform, and relevant config — scannable in five seconds	"Various versions"; environment mixed into the issue narrative
Issue	The problem in the customer's words, including the exact error text	Paraphrased into support jargon; error message missing or truncated
Cause	The actual mechanism, one paragraph	Speculation presented as fact; or restating the symptom
Resolution	Numbered steps a stranger could execute; first line restates what's being fixed	Steps that assume tribal knowledge; "reinstall and see"
Notes	Genuinely optional context: exceptions, related issues by ID	A dumping ground for what belonged above

7. Ten ways articles go wrong

1. The grab-bag — three issues, one article, zero citability.
2. The mystery environment — a perfect fix for an unstated version.
3. Support-speak — the customer's words scrubbed out, findability scrubbed with them.

4. The scavenger hunt — resolution steps that reference "the above."
5. The orphan — no owner, so no one ever retires it.
6. The immortal draft — unvalidated content quietly treated as true.
7. The prose applicability — "this mostly affects newer versions" instead of metadata.
8. The recycled ID — a citation that now points at something else.
9. The silent edit — content changed, `version` and `last_validated` untouched.
10. The resurrection — archived content edited back to life instead of a new article with a link.

8. Using the templates

The `templates/` folder contains the five types as ready-to-use files; `examples/` contains one gold-standard filled example of each. Guidance inside templates lives in HTML comments (`<!-- like this -->`) — invisible when rendered, deleted as you fill.

Adapter notes:

- **Git (recommended rung 1)**: commit `templates/` into your repo as-is; one file per article; `validate.py` (in the toolkit's tools) checks frontmatter in CI.
- **Zendesk / Salesforce / Freshdesk / Jira SM**: map frontmatter fields to article/custom fields (every platform supports the core set: type, product, version, owner, review date). Keep the section headers verbatim in the body. If a field has nowhere to live, put the frontmatter block at the top of the body as text — ugly, but it keeps the data with the article for the day you migrate.
- **Notion / Confluence**: frontmatter fields become database properties / page properties; body sections stay literal headings.

What this module assumes from its siblings: nothing. It's the module *others* assume. Module 3's prompts generate drafts in these formats; Module 4's gates check these fields; Module 5's lifecycle reads `volatility` and `last_validated`; Module 6 counts what these articles do in the wild.

Contextkeeping is a KCS-aligned operating model. KCS® is a service mark of the Consortium for Service Innovation; this material is not affiliated with or endorsed by the Consortium.