OKF v0.1 · English Handbook

Open Knowledge Format

Build an AI-maintained knowledge base — from beginner to enterprise

A knowledge base stored as plain Markdown files that humans and AI agents can read, write, and use together.

Compiled by **Supachai Jaturaprom [TumEz]**

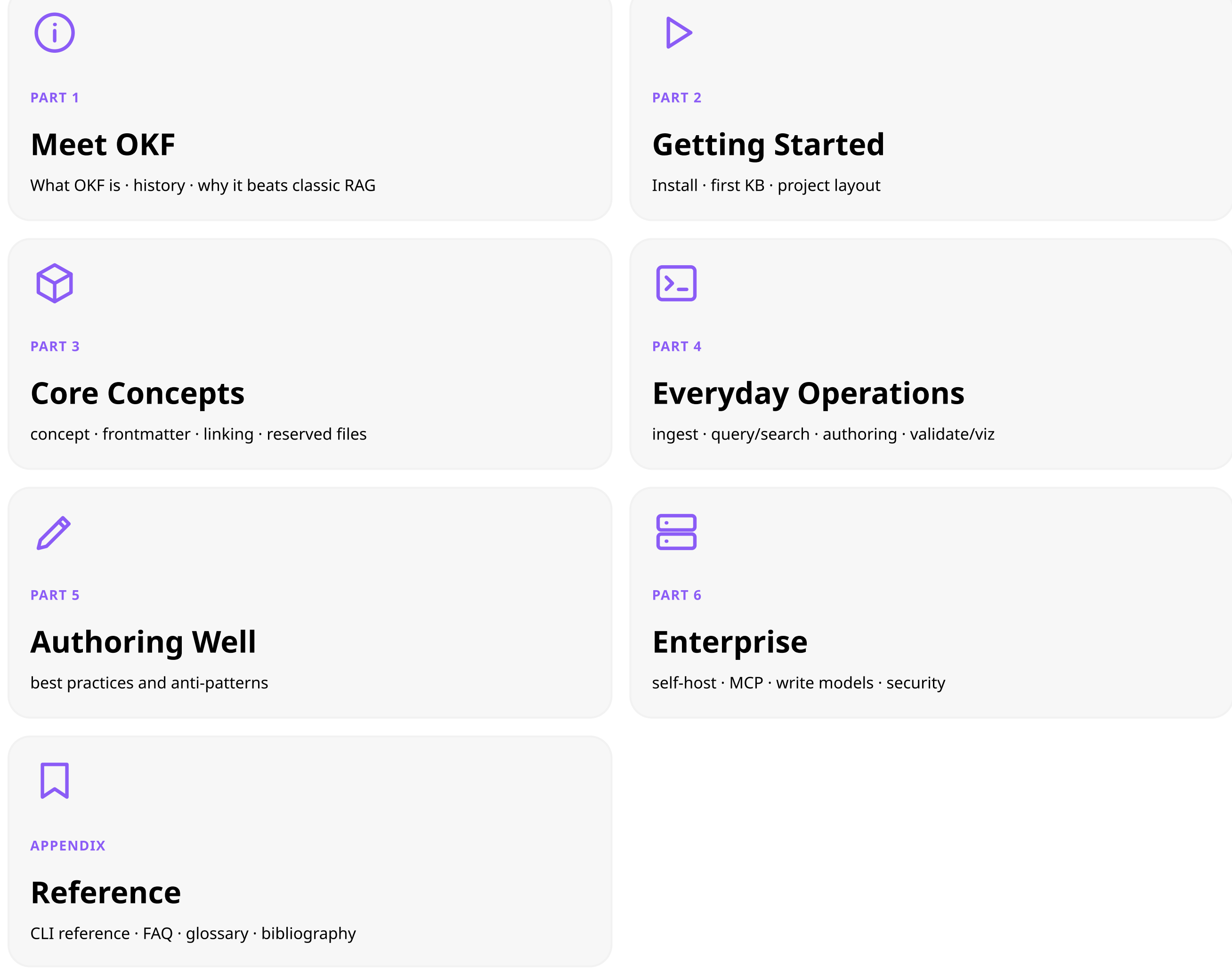
Written by Claude Code — Opus 4.8 (AI) · Updated 2026-06-15 · OKF v0.1

Preface

Welcome to **The Open Knowledge Format (OKF) Handbook**. This book walks you through building and operating an **AI-maintained knowledge base** with OKF — a knowledge base stored as Markdown files with YAML frontmatter that both humans and AI agents can use, without any SDK or special database.

> 🌐 อ่านฉบับภาษาไทยได้ที่ปุ่ม **ไทย** มุมขวาบน (Thai version available — click **ไทย** in the top bar)

Table of Contents


PART 1
Meet OKF
What OKF is · history · why it beats classic RAG

PART 2
Getting Started
Install · first KB · project layout

PART 3
Core Concepts
concept · frontmatter · linking · reserved files

PART 4
Everyday Operations
ingest · query/search · authoring · validate/viz

PART 5
Authoring Well
best practices and anti-patterns

PART 6
Enterprise
self-host · MCP · write models · security

APPENDIX
Reference
CLI reference · FAQ · glossary · bibliography

Who this book is for

- **Beginners** who want a searchable, vendor-neutral personal/team knowledge base
- **Developers / data teams** who want AI agents to reach internal knowledge systematically
- **Architects / platform teams** who need a shared system **across sessions and teams, on-prem**

No prior OKF knowledge needed — just basic command line and Git.

Conventions

- Commands you type go in code blocks · **technical terms** (`concept` , `frontmatter` , `bundle` , `MCP`) stay in English to match the source docs and code · blockquotes are warnings/tips

> **Versioning note:** OKF is a **v0.1** spec (published 2026-06-12 by Google Cloud). The core spec requires only the `type` field — most "best practices" here come from the LLM-wiki community and Google's reference implementation.

Source project (code + all tools): **https://github.com/supachai-j/open-knowledge-format-starter**

Start at What is OKF

Table of Contents

Part 1 — Meet OKF

What is OKF 5
Why OKF (vs. RAG) 8
History of Knowledge Bases 11
Core concepts you should know 17

Part 2 — Getting Started

Install 21
Your first knowledge base 23
Project layout 26

Part 3 — Core Concepts

Bundle, Concept, and Concept ID 29
Frontmatter (metadata) 31
Linking into a knowledge graph 34
Reserved files: index.md and log.md 36

Part 4 — Everyday Operations

Ingest: adding knowledge to the wiki 39
Query and Search 41
Adding and editing concepts 43
Validate and Visualize 46
Worked example: a bookstore KB 49

Part 5 — Authoring Well

Best practices and anti-patterns 53

Part 6 — Enterprise

Architecture overview 55
Self-hosted deployment 58
Write models: PR-gated and Lease 61
Search at scale and semantic 64
Security and governance 66

Appendix

Tools reference (CLI) 69
FAQ 72
Glossary 75
References 78

What is OKF?

Open Knowledge Format (OKF) is an open specification for storing an organization's "knowledge" as a **directory of Markdown files with YAML frontmatter**, so that both humans and AI agents can write, read, exchange, and use that knowledge — without needing any SDK, database, or specialized tooling.

> If you can `cat` a file, you can read OKF · If you can `git clone` a repo, you can share OKF

Background

OKF v0.1 was released on **12 June 2026** by the Data Cloud team at **Google Cloud** (Sam McVeety and Amir Hormati) as a way of turning the **"LLM-wiki pattern"** proposed by Andrej Karpathy into a portable, vendor-neutral standard.

The core idea of LLM-wiki is: instead of having an LLM re-discover raw documents from scratch every time a question is asked, let AI **progressively synthesize knowledge into organized, pre-linked Markdown pages**, then load only the relevant pages directly into context.

Core Components (see Part 3 for details)

Term	Short definition
Bundle	A directory of all knowledge files — the unit of distribution
Concept	One unit of knowledge = one `.md` file (e.g. a table, a metric, a playbook)
Concept ID	The file's path within the bundle with `.md` stripped, e.g. `tables/orders.md` → `tables/orders`
Frontmatter	The YAML block at the top of a file (stores metadata such as `type`, `title`, `tags`)
Link	A Markdown link between concepts = builds relationships into a graph

Example: a single concept file

File `tables/orders.md`:

```
---
type: BigQuery Table
title: Orders
description: One row per customer order
tags: [sales, orders]
timestamp: 2026-06-15T00:00:00Z
---

# Schema
| Column | Type | Description |
| :--- | :--- | :--- |
| order_id | STRING | Order identifier (unique) |
| customer_id | STRING | FK to [customers](customers.md) |

# Joins
Joined to [customers](customers.md) via `customer_id`
```

As you can see, this is plain Markdown that is immediately human-readable — just a small YAML header and links to other concepts.

3 Design Principles

1. **Minimally opinionated** — frontmatter requires only a single `type` field; everything else is defined by the producer.
2. **Producer and consumer are independently decoupled** — a bundle written by hand, generated by an agent, or exported by a pipeline can be read by any tool.
3. **A graph, not just a tree** — concepts link to each other via Markdown links, creating richer relationships than folder structure alone.

What OKF is NOT

- Not a fixed taxonomy — it does not prescribe which types must exist.
- Not an opinion on how knowledge must be stored, served, or searched.

- Not a replacement for domain-specific schemas (Avro, Protobuf, OpenAPI) — OKF **references** those; it does not absorb them.

Next, let's look at **why** you would choose OKF over traditional RAG → Why OKF?

Why OKF? (vs. RAG)

If you have ever built an AI question-answering system on top of organizational documents, you have probably used **RAG** (Retrieval-Augmented Generation) before. OKF is not here to kill RAG — it is here to address the weaknesses that RAG most commonly hits in production.

How traditional RAG works

```
Raw documents → chunk → embed → store in vector DB
                                        ↓ (at query time)
                   query → find nearest chunks → stuff into context →
answer
```

The LLM **re-discovers knowledge from scratch on every query** — no accumulation, no understanding of which chunk relates to what, contradicts what, or is stale.

Common RAG failure modes

Problem	What happens
Chunking artifacts	Sentences are cut mid-way — e.g. "employees may work from home, except during the first 90 days" may be reduced to just "employees may work from home" → confidently wrong answer
Knowledge decay	New documents keep arriving but old contradicting content stays → old content gets retrieved (a leading cause of RAG project failures)
Black box	Retrieval returns chunks that look similar but are not the best answer, and nobody can see why
No accumulation	A question requiring synthesis across 5 documents must be reassembled from scratch every time; nothing is built up permanently

How OKF is different

OKF shifts from "pull raw chunks at query time" to **"a wiki that AI maintains and pre-synthesizes"**:

- When a new source arrives, AI **reads, summarizes, and merges it into the existing wiki** — updating pages, fixing cross-references, and **flagging conflicts when new information contradicts old**.
- At query time, already-digested Markdown pages are loaded directly into context — no chunking, no vector math.
- Knowledge becomes **a compounding asset**: every new source added makes the wiki richer.

Dimension	RAG (raw)	OKF (pre-synthesized)
Storage format	Specialized vector DB	Plain Markdown files + git
Human-readable?	No (UI / binary)	Yes, with any text editor
Version control	Difficult	Built-in (git diff / PR / blame)
Knows about conflicts / staleness?	No	Can flag and supersede
Vendor lock-in	High	None (it's just files)

Is "RAG is dead" actually true?

No — most engineers view them as **separate layers, not an either/or choice**:

- **Layer 1 — wiki (OKF):** Core synthesized knowledge. Finding the answer here is the end of the search (fastest, highest signal).
- **Layer 2 — raw documents + vector search:** Used when the wiki does not yet cover the question (fall back to raw sources).
- **Layer 3 — LLM general knowledge:** Fills gaps that exist in neither the wiki nor the raw documents.

The OKF starter in this book supports both worlds: the wiki as the primary layer, plus **hybrid search (BM25 + semantic)** available as a next step when the wiki grows (see Part 6).

When to use / not use OKF

Good fit when: knowledge must be long-lived, reused, shared across multiple people or agents, auditable, and you want to avoid vendor lock-in.

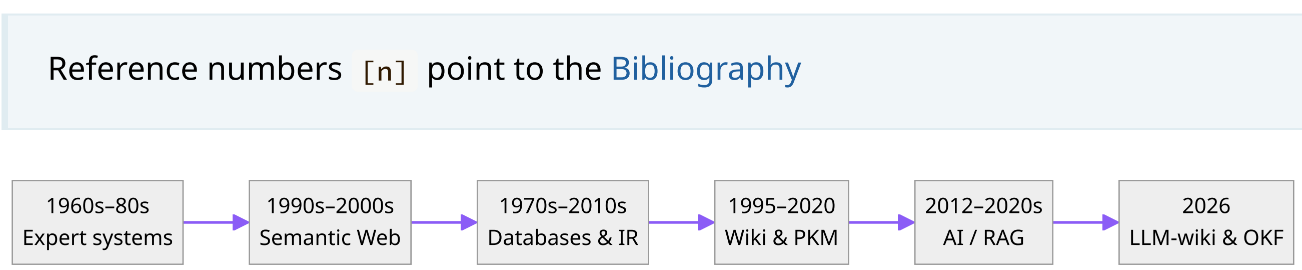
May not be worth it when: you have a massive corpus of raw documents that will never be synthesized (straight RAG is more cost-effective), or the data is single-use and throwaway.

Ready to get started → Install

History and Evolution of the Knowledge Base

For over sixty years, humanity has asked the same question again and again: "How do we make machines *remember* and *understand* what we know?" The answer has changed with every era — from hand-crafted rules, to semantic graphs, to word indexes, to interlinked notes, and most recently to AI that synthesizes knowledge on our behalf. This chapter is the story of that journey, and its (provisional) destination: OKF.

> Reference numbers `[n]` point to the Bibliography



Act 1 — The Era We Tried to Hand-Feed Knowledge to Machines

In the 1960s at Stanford, a group of scientists believed that intelligence in a narrow domain could be captured by **encoding expert knowledge as rules** and letting a machine reason over them. They built **DENDRAL** (1965), which analyzed molecular structures from mass spectrometry data on par with a trained chemist [1] — the first time the world saw that "knowledge," not "search," was the real key to AI.

A few years later, **MYCIN** (early 1970s, Edward Shortliffe) used around 600 IF-THEN rules to diagnose bloodstream infections with accuracy matching senior physicians — and it could **explain its own reasoning** [1]. The heart of MYCIN was an architecture that lives with us to this day: separating the **knowledge base** (what is known) from the **inference engine** (how to reason):

```
IF   infection-type = primary-bacteremia
AND  culture-site   = blood
AND  portal-of-entry = gastrointestinal-tract
THEN there is suggestive evidence (CF = 0.4) that the organism is
Bacteroides
```

But the dream hit a wall. In 1984, Douglas Lenat launched the **Cyc** project — an attempt to hand-encode *all* of human commonsense knowledge [1]. Decades and tens

of millions of dollars later, the world learned an expensive lesson called the **"knowledge acquisition bottleneck"**: hand-entering knowledge can never scale.

> 🧬 **DNA inherited by OKF:** Separating "knowledge (files)" from "the engine that uses knowledge (agent)" — OKF's producer/consumer architecture is a direct descendant of this idea.

Act 2 — The Era We Tried to Give "Meaning" to the Web

In 2001, the father of the web, **Tim Berners-Lee**, published an article in *Scientific American* dreaming of the **Semantic Web** [2] — a web that was not merely documents for humans to read, but data whose *meaning machines could understand*. He proposed storing knowledge as **triples** (subject–predicate–object), which are simply "edges in a graph":

```
@prefix ex: <http://example.org/> .
ex:TimBernersLee  ex:invented  ex:WorldWideWeb .
```

With **RDF, OWL, and SPARQL**, machines could infer new facts from the graph automatically [2]. It was beautiful in theory — but writing a correct ontology was far beyond what ordinary people could manage. By 2013, fewer than 2% of websites used semantic markup. What survived and flourished instead were the "easier to use" successors: **linked data**, **schema.org** (2011), and the **knowledge graph** [2].

> 🧬 **DNA into OKF:** Knowledge connected as a graph has enormous value — but OKF chose plain **Markdown links** (untyped) over strict RDF/OWL, so that people can actually write it.

Act 3 — The Era We Learned to Search Intelligently

While AI dreamed big, another line of work proceeded quietly — and changed the world. In 1970, **Edgar Codd** of IBM proposed the **relational model**, allowing data to be stored in tables and queried with SQL [3]. But it matched records exactly; it could not search free text and rank results by relevance.

The answer came from **Karen Spärck Jones** (1972), who proposed **IDF** — a simple but profound idea: *a word that appears in fewer documents is a stronger signal than one that appears everywhere* [3]. Combined with term frequency this became **TF-IDF**, and later **BM25** (~1994), which remains the standard for lexical search to this day. Here is a tiny example — the word "the" lives in every document and is worthless for discrimination, while the word "dog," found in a single document, stands out sharply:

Term	Documents	Distinctiveness
the	D1, D2, D3	0 (useless)
dog	D3	high

When **Doug Cutting** released **Lucene** (1999) and later **Elasticsearch** (2010), industrial-grade full-text search landed in everyone's hands [3]. The one limitation that lingered: it was *lexical* — searching "car" would miss "automobile" because it understood no meaning.

> 🧬 **DNA into OKF:** BM25 is still powerful and lightweight — `tools/okf-index.py` uses it as the primary search method.

Act 4 — The Era Knowledge Became Everyone's

In 1995, **Ward Cunningham** created **WikiWikiWeb**, the first website that anyone could edit [4]. Six years later, **Wikipedia** (2001) proved that humanity's collective knowledge could grow through open contribution [4].

But the most remarkable story had unfolded earlier, on the desk of a German sociologist. **Niklas Luhmann** had accumulated a **Zettelkasten** of some 90,000 index cards, each linked to others. He produced more than 50 books from it and left behind an immortal principle: **"The value lies not in the individual note, but in the links between notes."** [4]

The digital age rediscovered this principle around 2016–2020 through **Notion, Roam, and Obsidian** — all built on **Markdown +** `[[wikilinks]]`, which won the format wars for compounding reasons: *humans can read it without rendering, and machines can parse it without a special parser* [4].

```
---
title: "Zettelkasten Principle"
tags: [pkm]
---
The value of the card box lies in the **links between notes** — see also
[[Obsidian]]
```

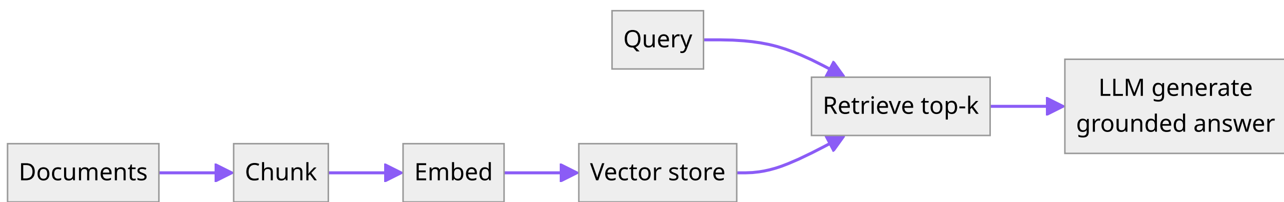
> 🧬 **DNA into OKF:** This is the most direct lineage — Markdown + YAML frontmatter + links between concepts is exactly what OKF looks like.

Act 5 — The Era "Meaning" Became Geometry

In 2013, **Tomas Mikolov**'s team at Google revealed something that looked like magic: **word2vec** converted words into vectors where "similar meaning = nearby position," making arithmetic of meaning possible [5].

king → - man + woman → ≈ queen

The years that followed accelerated relentlessly: **Google Knowledge Graph** (2012, "things, not strings"), **FAISS** (2017) searching billions of vectors, **BERT** (2018) giving the same word different meanings depending on context [5]. Then in 2020, **RAG** (Lewis et al.) arrived to fix the largest weakness of LLMs — fabrication — by retrieving real evidence to ground answers [5]:



And to combine the strengths of "exact term matching (BM25)" with "semantic matching (vector)," the world reached for **Reciprocal Rank Fusion** (RRF, 2009), producing the **hybrid search** that is today's default [5].

> 🧬 **DNA into OKF:** wiki = Layer 1 (pre-synthesized); RAG/vector = Layer 2 (mining raw sources); `okf-search.py` fuses BM25 + semantic with RRF.

Act 6 — The Present: When AI Takes Care of Knowledge Itself

In April 2026, **Andrej Karpathy** posted a short idea that ignited the entire field: **"LLM wiki"** [6] — instead of retrieving raw chunks on every query (as RAG does), have an agent **compile raw sources into organized, interlinked, continuously maintained Markdown**. Synthesize once at ingest; knowledge therefore **compounds** — the more you use it, the richer it gets, rather than starting from zero every time.

Two months later, on 12 June 2026, **Google Cloud** (Sam McVeety, Amir Hormati) made this pattern an open standard under the name **Open Knowledge Format (OKF) v0.1** [6] — a directory of Markdown files + YAML frontmatter requiring only a `type` field, with separate producer/consumer roles, portable across clouds and frameworks.

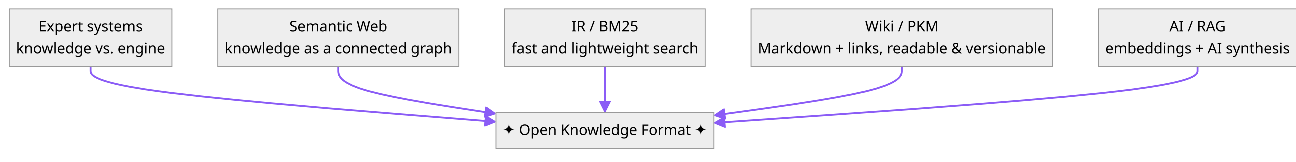
Before that, **MemGPT/Letta** (2023) had already demonstrated "LLM as OS" — managing memory in a tiered fashion (in-context = RAM, external = disk) and paving the way for agents with persistent memory [6].

Epilogue — And the Future That Is Coming

The direction ahead is a **self-maintaining knowledge base**: agents do not merely "query" knowledge — they *curate* it — checking whether information has gone stale (via `timestamp` / `log.md`), reconciling conflicts across concepts, and proposing updates for human approval before committing [6]. The next layer will likely be a **hybrid wiki + RAG** architecture: the pre-synthesized wiki serves as a fast index, while RAG fills gaps for data that changes too frequently to precompile — with agent memory as the runtime and multi-agent systems dividing responsibilities for curating and consuming knowledge, moving toward what is beginning to be called **"compiled-knowledge generation"**.

Why OKF Is the Sum of All Six Acts

OKF does not reinvent everything — it **fuses the best parts of every era into one**. Six streams converge into a single confluence:



Next, dive into the Core Concepts Explained (with Examples) introduced throughout this chapter.

Core Concepts Explained (with Examples)

This chapter summarizes the foundational concepts of Knowledge Base systems introduced in the History chapter — each topic includes a **short definition + concrete example + how it relates to OKF** so you can apply it immediately.

1. Knowledge Representation

What it is: A way of storing "what is known" so a machine can process it. Classic forms: production rules (IF-THEN), semantic networks (concept graphs), frames (slot-filler).

Example (frame):

```
FRAME: Bacteroides
  IS-A: Anaerobic-Gram-Negative-Rod
  Gram-stain: negative (default)
  Treatment: [metronidazole, clindamycin]
```

In OKF: Each concept (a `.md` file) is one unit of knowledge — the frontmatter holds queryable fields; the body holds detail.

2. Ontology

What it is: A machine-readable specification of the concepts and relationships in a domain (more than a taxonomy — it includes logic that enables inference).

Example (RDF triple): `TimBernersLee — invented — WorldWideWeb` (subject–predicate–object)

In OKF: Markdown links between concepts serve a similar role to triples but are **untyped** — the type of relationship lives in the prose (much lighter than OWL).

3. Inverted Index

What it is: A data structure that maps "term → list of documents containing that term" — the heart of full-text search.

Example:

Term	Documents
cat	D1, D2
dog	D3

Search "dog" → returns D3 immediately, without scanning every document.

In OKF: `okf-index.py` builds an in-memory inverted index to power BM25.

4. TF-IDF & BM25 (Relevance Ranking)

What it is: A scoring formula measuring how well a document matches a query — **TF** (how often the term appears in the document) × **IDF** (how rare the term is across the corpus = more distinctive); **BM25** improves on this with length normalization and saturation.

Example: The word "the" appears in every document → IDF = log(3/3) = 0 → score 0 (does not help discriminate). The word "dog" appears in only one document → high IDF → distinctive.

In OKF: BM25 is the primary search method in `okf-search.py` (lightweight, no extra dependencies).

5. Embeddings (Semantic Vectors)

What it is: Converting text into a numeric vector where "similar meaning = similar vector".

Example (word2vec): `king - man + woman ≈ queen` — semantic relationships become vector arithmetic.

In OKF: `okf-embed.py` generates embeddings for concepts using a local model (Ollama) for semantic search.

6. Vector / Semantic Search

What it is: Searching by vector proximity (e.g., cosine similarity) → captures meaning and synonymy that keyword search misses.

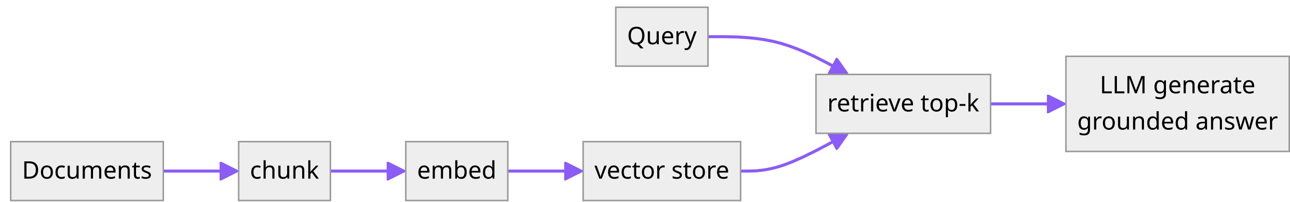
Example: Searching "car" finds a document written with "automobile" because their vectors are close.

In OKF: An optional layer (opt-in) — if embeddings/Ollama are unavailable it automatically falls back to BM25.

7. RAG (Retrieval-Augmented Generation)

What it is: Retrieving relevant information and inserting it into the LLM's context at query time to ground the answer (reduce hallucination, enable citation).

Example (5 steps):



In OKF: The wiki = Layer 1 (pre-synthesized; finding it in the wiki is sufficient); RAG = Layer 2 (mining raw documents when the wiki does not yet cover the topic).

8. Hybrid Search & RRF

What it is: Combining results from multiple retrieval methods (BM25 + semantic) using **Reciprocal Rank Fusion**: `score(d) = Σ 1/(k + rank)` (k=60).

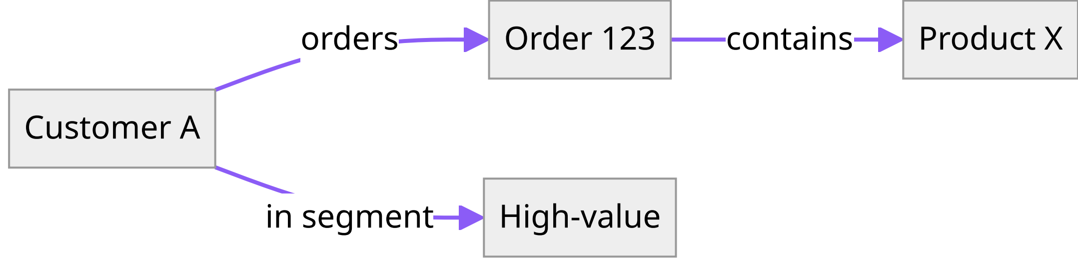
Example: A document that ranks highly in both BM25 and semantic search floats to the top, with no single signal dominating.

In OKF: `okf-search.py` uses RRF to fuse BM25 + semantic results.

9. Knowledge Graph

What it is: A graph of entities (nodes) + typed relationships (edges) — "things, not strings" — enabling entity-level disambiguation and reasoning.

Example:



In OKF: The entire bundle forms a knowledge graph (concepts = nodes, links = edges) — viewable with `okf-viz.py`.

Summary Table

Concept	What it captures	Example in OKF
Knowledge representation	Knowledge structure	concept + frontmatter
Ontology	Semantic relationships	Markdown links (untyped)
Inverted index / BM25	Exact term matching	`okf-index.py`
Embeddings / vector search	Semantic meaning	`okf-embed.py`
RAG	Grounding LLM answers	wiki (L1) + RAG (L2)
Hybrid / RRF	Fusing multiple signals	`okf-search.py`
Knowledge graph	Entities + relationships	bundle + `okf-viz.py`

See the Bibliography for the original papers and standards behind each concept.

Install

The OKF starter is a **pure-Python tool (no mandatory dependencies)** that works fully offline / air-gapped. Requirements:

- **Python 3.13+** (every tool uses the stdlib only)
- **Git** (for version-controlling your bundle)
- *(Optional)* **Ollama** if you want local semantic search
- *(Optional)* **Docker** if you want to self-host the server (Part 6)

Option 1 — Install as a Claude Code skill (recommended)

This approach lets every project and every session create and maintain an OKF bundle without needing to be inside the source repository.

```
git clone https://github.com/supachai-j/open-knowledge-format-starter.git
cd open-knowledge-format-starter

./install.sh                  # install globally → ~/.claude/skills/okf
(available to all projects)
./install.sh --project        # install for this project only →
./.claude/skills/okf
./install.sh --dir <path>     # install to a custom path
./install.sh --uninstall      # remove the installation
```

`install.sh` bundles the skill (the `SKILL.md` file + all Python tools + the embedded viewer library) into a self-contained skill folder. Afterwards, open Claude Code and type `/okf` or say *"init an OKF knowledge base here"* to get started.

> **How the skill works:** When invoked, Claude Code locates the skill and runs the scripts in its `scripts/` directory — regardless of whether it was installed globally or per-project.

Option 2 — Use the repo directly

If you prefer to work inside the repository itself (or are not ready to install the skill yet), all tools are available under `tools/`:

```
git clone https://github.com/supachai-j/open-knowledge-format-starter.git
cd open-knowledge-format-starter
python3 tools/okf-validate.py ./wiki          # → ✓ CONFORMANT with OKF v0.1
python3 tools/okf-viz.py ./wiki               # → wiki/viz.html (open in
browser)
```

Option 3 — Use as a GitHub template

The source repository is configured as a **template repository** — click **"Use this template"** on GitHub to create your own new repo with the full structure already in place.

Verify the installation

```
python3 --version                    # should be 3.13 or higher
python3 tools/okf-validate.py --help 2>/dev/null || python3 tools/okf-
validate.py ./wiki
```

If you see `✓ CONFORMANT with OKF v0.1`, you are ready to go. Let's create your first knowledge base → Create your first knowledge base

Build Your First Knowledge Base

Let's create your first OKF bundle in just a few minutes.

Step 1 — Scaffold the structure

Use `okf-init.py` to scaffold a new bundle in the current directory (or a directory you specify):

```
python3 tools/okf-init.py .            # scaffold in the current directory
# or if installed as a skill:
python3 ~/.claude/skills/okf/scripts/okf-init.py ./my-kb
```

Output:

```
Scaffolding OKF bundle in /path/to/project
  wrote: AGENTS.md
  wrote: raw/README.md
  wrote: wiki/index.md
  wrote: wiki/log.md
  wrote: wiki/getting-started.md
✓ done. Next: validate with okf-validate.py, then ingest sources from raw/.
```

> **Safe:** If a non-empty `wiki/` directory already exists, the script will **not overwrite** it unless you pass `--force`.

Step 2 — Check conformance

```
python3 tools/okf-validate.py ./wiki
# → ✓ CONFORMANT with OKF v0.1  (0 warning(s), 0 info)
```

Step 3 — View the graph

```
python3 tools/okf-viz.py ./wiki --name "My First KB"
# → wiki/viz.html  (single file, open in a browser instantly — no internet required)
```

Open `wiki/viz.html` to see concepts as nodes connected by edges. Click any node to view its details.

Step 4 — Start capturing knowledge

1. Drop your source files (PDFs, notes, exports) into the `raw/` folder.
2. Tell the AI agent to *"ingest raw/ into the wiki"* — the agent will extract key points for your approval before writing anything. (Details in the Ingest chapter.)

Writing concepts by hand

OKF is plain Markdown — you can create a concept yourself. For example, create `wiki/glossary/wau.md`:

```
---
type: Metric
title: Weekly Active Users (WAU)
description: The number of unique customers who placed at least one order in the past 7 days
tags: [growth, metric]
timestamp: 2026-06-15T00:00:00Z
---

# Definition
Count unique `customer_id` values that have at least one order in the last 7-day window.
```

Then run `okf-validate.py` again to confirm it is still conformant.

Quick-reference command table

Task	Command
Create a new bundle	`okf-init.py <dir>`
Check conformance	`okf-validate.py ./wiki`
View the graph	`okf-viz.py ./wiki`
Build the search index	`okf-index.py build ./wiki`
Search	`okf-search.py "question" --bundle ./wiki`

Next, understand the project layout that was scaffolded for you.

Project Layout

The OKF starter uses a **three-layer architecture** with clearly separated responsibilities.

```
my-kb/
├── AGENTS.md           ← Layer 3: schema + agent operating rules (read this first)
├── raw/                ← Layer 1: raw source material (read-only)
│   └── README.md
├── wiki/               ← Layer 2: OKF bundle (agent-managed)
│   ├── index.md        ← reserved file: table of contents (progressive disclosure)
│   ├── log.md          ← reserved file: change log
│   └── getting-started.md
└── tools/              ← tooling (validate, viz, index, ...)
```

What are the three layers?

Layer	Folder	Who writes	Role
1. Raw sources	`raw/`	Human	Source of truth — **read-only**, agents never modify
2. Wiki	`wiki/`	Agent	Synthesized knowledge (OKF bundle)
3. Schema	`AGENTS.md`	Human + agent jointly	Rules, conventions, and workflow that govern the agent

Key point: **`wiki/` is the "bundle root"**, so Concept IDs are counted from `wiki/`. For example, `wiki/tables/orders.md` has Concept ID = `tables/orders`.

`AGENTS.md` — The most important file

This is the first file an agent reads. It tells the agent:

- The structure of the bundle
- Frontmatter rules (which fields are required/recommended, the controlled vocabulary for `type`)
- Linking rules (use relative paths, never start with `/`)
- The ingest / query / validate workflow

> This file is what turns the AI into a "disciplined wiki curator" rather than a generic chatbot. You and the agent will gradually develop this file together to fit your domain.

If you use a different agent, the filename might be `CLAUDE.md` (Claude Code) or `GEMINI.md` — the content is the same.

`raw/` — Raw source material

Place your originals here: PDFs, meeting notes, exports, datasets.

- **Immutable** — agents read but never modify. This is the source of truth.
- Files in `raw/` are **not part of the bundle** (the bundle is `wiki/`); they are the *input* to the ingest process.
- In the real starter, `.gitignore` will **not commit files inside `raw/`** to prevent accidentally pushing private or large data (only `raw/README.md` is kept) — if you want to version-control your actual sources, comment out the `raw/*` line.

`wiki/` — OKF bundle

This folder is owned by the agent. It organizes concepts into categories. The initial scaffold contains:

```
wiki/
├── index.md              ← root table of contents (contains okf_version)
├── log.md                ← change log
├── getting-started.md    ← example concept
└── (yours) tables/  references/  playbooks/ ...
```

Commonly seen categories (from the Google reference implementation):

- `tables/` — tables (`type: BigQuery Table`)
- `datasets/` — datasets (`type: BigQuery Dataset`)
- `references/` — synthesized knowledge, e.g. `references/metrics/`, `references/joins/` (`type: Reference`)

Next, dive into the core concepts — start with Bundle, Concept, and Concept ID.

Bundle, Concept, and Concept ID

These three terms are the heart of OKF. Understand them and you understand half the spec.

Bundle (knowledge set)

A **bundle** is the directory containing all knowledge files — it is the **unit of distribution**. A single bundle can be shared in several forms:

- As a **git repository** (recommended — you get history, diffs, and code review for free)
- As a tarball / zip
- As a subdirectory inside a larger repository

In this project the bundle is the `wiki/` folder.

Concept (unit of knowledge)

A **concept** is one unit of knowledge, represented as **a single Markdown file**. It may describe:

- Tangible things — a database table, an API endpoint
- Abstract things — a business metric, a playbook, a process

The golden rule: one file = one concept. Do not pack three topics into one file.

A concept has two parts:

```
---
type: Metric              ← (1) Frontmatter — YAML block
title: ...
---
                          ← (2) Body — free-form Markdown
# Definition
...
```

Concept ID (concept identifier)

Concept ID = the file's path inside the bundle with `.md` removed.

File (inside `wiki/`)	Concept ID
`tables/orders.md`	`tables/orders`
`references/metrics/wau.md`	`references/metrics/wau`
`playbooks/incident.md`	`playbooks/incident`

Key point: **identity is tied to the file path**, so…

> ⚠️ **Renaming or moving a file = changing the Concept ID = breaking all inbound links.** Choose stable filenames from the start. Use lowercase with hyphens (kebab-case), e.g. `weekly-active-users.md`.

The advantage of using the path as identity: you get **sovereign identity** with no central registry — just look at the file's address to know which concept it is.

Tree structure, graph relationships

Folders provide a tree structure (parent-child), but concepts can link to each other freely via **Markdown links**, forming a **graph** that is richer than the folder hierarchy (see the Linking chapter).

```
wiki/
├── tables/orders.md ───┐  (link "joins-with")
├── tables/customers.md ◄┘
└── references/metrics/wau.md ──► tables/orders.md ("derived-from")
```

Next, examine the file header in detail → Frontmatter

Frontmatter (Metadata)

Frontmatter is a YAML block at the top of a file, delimited by `---`, that stores metadata intended for search, filtering, and indexing.

```
---
type: BigQuery Table                 # required
title: Orders                        # recommended
description: หนึ่งแถวต่อหนึ่งคำสั่งซื้อ    # recommended
resource: https://...                # recommended (if a real resource exists
to point to)
tags: [sales, orders]                # recommended
timestamp: 2026-06-15T00:00:00Z      # recommended (ISO 8601)
---
```

Required Fields

Field	Description
`type`	The kind of concept — this is the **only required field** in v0.1. Consumers use this value for routing, filtering, and display.

Recommended Fields (in order of importance)

Field	Description
`title`	A human-readable name. If omitted, consumers may infer it from the filename.
`description`	A one-line summary — **this is what an agent reads to decide whether to load the file**. Write it to be precise.
`resource`	URI of the real-world resource that the concept describes (may be omitted for abstract concepts).
`tags`	A YAML list for cross-domain categorisation, e.g. `[sales, revenue]`.

Field	Description
`timestamp`	The time of the most recent significant edit (ISO 8601 format).

> 💡 `description` **matters more than you think** — write it precisely, e.g. *"Number of unique customers with ≥1 order in the last 7 days"* is better than *"About WAU"*

Extension Fields

Producers **may add any fields they like**, and consumers **must tolerate unknown keys** (they must not reject the file). For example, add `owner`, `sla`, or `pii: true` as your domain requires.

Controlled Vocabulary for type

Although the spec does not enforce a taxonomy, `type` **values should be used consistently** — otherwise data aggregation becomes impossible. This starter uses the following set:

`BigQuery Table` · `BigQuery Dataset` · `Metric` · `Reference` · `Playbook` · `API Endpoint` · `Concept` · `Entity`

> **Convention from Google's reference bundle:** Synthesised or reference knowledge (joins, metric definitions, glossary) is typically placed under `references/` with `type: Reference`, while concrete assets live under `tables/` / `datasets/`.

Pitfalls

- **Do not over-stuff frontmatter** — keep it semantic (entity, intent, definition); do not dump keywords, because noise degrades search quality.
- **Use field names consistently** — if one file uses `author_name` and another uses `written_by`, aggregation breaks.

Body Below Frontmatter

The content section is free-form Markdown, but **structured markup** (headings, bullets, tables) is preferred over long prose paragraphs. Conventional headings (use when appropriate, in this order):

`# Overview` → `# Schema` → `# Common query patterns` (code: `sql`) → `# Joins` → `# Examples` → `# Citations`

Next, see how to connect concepts together → Linking as a knowledge graph

Linking as a Knowledge Graph

Concepts connect to one another via **standard Markdown links**, turning the directory into a **graph of relationships** that is richer than the parent–child structure of folders alone.

Use File-Relative Links Only

```
See the [customers](../tables/customers.md) table for the key used in the
join.
```

> **Do not begin a link with `/`** — this is an important and subtle rule.
>
> OKF spec §5.1 *recommends* absolute-style links (`/tables/customers.md`), claiming they are stable when files move. **However, Google's actual enrichment agent forbids them** because `/` -prefixed links **break GitHub rendering**, and every real Google bundle uses **relative** links throughout.
>
> **We follow the real implementation: relative links only** (`okf-validate.py` will warn if it finds a `/` -prefixed link).

Links Are Untyped

A link from concept A → B simply "asserts that a relationship exists", but the **type of that relationship** (parent–child, reference, joins-with, depends-on) lives in the **surrounding prose, not in the link itself**.

```
Joins with [customers](../tables/customers.md) via `customer_id` (many
orders → one customer)
           └─ untyped link          └─ relationship type is expressed in
this sentence
```

Consumers that build graph views treat every link as a directed, untyped edge.

Broken Links Are Permitted

A link pointing to a concept that **does not yet exist** is not an error — it represents **knowledge not yet written** (a placeholder / gap to be filled later). Consumers **must tolerate broken links**.

Rules for Good Linking

- ✅ Use relative paths only (`../tables/orders.md` , `customers.md`)
- ✅ Link only to concepts that actually exist (or intentionally leave placeholders)
- ❌ Do not over-link — one link per mention per section is enough
- ❌ Do not link from **headings**, inside **code blocks**, or in schema field-name lists
- ❌ Do not link a file to itself

Example: A Small Graph

```
references/joins/orders__customers.md
    ├──► tables/orders.md
    └──► tables/customers.md

metrics/weekly-active-users.md ──► tables/orders.md
playbooks/incident-response.md ──► metrics/weekly-active-users.md
```

When you open `viz.html` you will see this graph interactively, including **"Cited by"** (backlinks — which concepts link to this one).

Next, see the two special files that help navigate the graph → Reserved files: index.md and log.md

Reserved Files: index.md and log.md

OKF reserves two filenames that carry special meaning at every level of the directory tree, and **must not be used as concept files**.

File	Purpose
`index.md`	Directory table of contents (progressive disclosure)
`log.md`	Change history log

`index.md` — Table of Contents for Progressive Disclosure

Allows people and agents to **see what is available before opening any real file** — this is crucial for preventing context-window overflow during exploration. An agent reads `index.md` first to find relevant concepts, then drills into individual files.

Format: no frontmatter (except at the root index — see below); content is headings and lists in the form `* [Name](link) - short description`

```
# Subdirectories

* [tables](tables/index.md) - BigQuery tables and schema/join information
* [references](references/index.md) - Synthesised knowledge (joins, metric definitions)

# Concepts

* [tables/orders](tables/orders.md) - One row per order
```

> Producers may **auto-generate `index.md`**; consumers may **build it on the fly when reading** if it is absent.

`okf_version` at the Root Index

At the **bundle root only**, `index.md` may carry frontmatter — and only one field: `okf_version`, which declares the spec version this bundle conforms to:

```
---
okf_version: "0.1"
---
# Concepts
...
```

`log.md` — Change History

Records changes within that scope as a list **sorted newest first**, grouped under ISO date headings.

```
# Directory Update Log

## 2026-06-15
* **Update**: เพิ่มตาราง [Customer Metrics](references/metrics/cm.md)
* **Creation**: สร้าง [Dataplex Playbook](playbooks/dataplex.md)

## 2026-06-12
* **Initialization**: สร้างโครงสร้างไดเรกทอรีพื้นฐาน
```

Rules:

- Date headings **must** follow the ISO format `YYYY-MM-DD` (**no brackets**)
- Bold prefixes (`**Update**`, `**Creation**`, `**Deprecation**`) are **convention, not mandatory**

> 💡 Consistent prefixes make grepping easy, e.g. `grep '^- ' log.md` to view the latest entries.

Why These Two Files Matter

- `index.md` = **navigator** that lets an agent explore the bundle without loading all of it into context
- `log.md` = **readable audit trail** that shows how the wiki has evolved and helps an agent understand what was recently changed

> `okf-validate.py` checks that `index.md` has no frontmatter (except `okf_version` at the root) and warns if any date heading in `log.md` is not in ISO format.

That concludes the core concepts. Next, put it all into practice → Ingest: Adding Knowledge to the Wiki

Ingest: Adding Knowledge to the Wiki

Ingest is the process of taking raw source material and synthesizing it into concepts in the wiki. This is the heart of making the wiki "richer."

The Most Important Principle: Ingest Must Be Human-Supervised

> 🚫 **Do not run ingest automatically in the background (as a background daemon)**
>
> This is the most critical anti-pattern — a daemon that swallows everything it sees will **accumulate noise as fast as it accumulates signal**, and the wiki will **rot silently** until nothing in it can be trusted.
>
> Keep ingest as a **deliberate human command** — the human decision "is this source worth synthesizing?" is the **quality gate** that eliminates an entire class of failures.

Ingest Steps (Human-Supervised)

1. **Read the source** in `raw/`
2. **Read** `wiki/index.md` to see what concepts already exist
3. **Extract 5–15 key points** (claims/decisions/insights) worth keeping
4. **Show the extracted points + their proposed mapping to concepts for your approval — then wait before writing anything**
5. **Reconcile contradictions** — if new information conflicts with an existing concept, add a flag to the old file:

   ```
   > **CONTRADICTION FLAG**: New findings supersede this value. See
   references/metrics/new-wau.md
   ```

6. **Write/update the concept** (starting from the template), update `tags` + `timestamp`
7. **Update the relevant** `index.md`

8. **Add an entry to `log.md`** under today's date
9. **Run validate** before considering the work done

Using via AI Agent

If the skill is installed, simply say:

```
ingest raw/q3-strategy.pdf into the wiki
```

The agent will follow the steps above — extracting key points, **showing them for your approval first**, then writing the concept, and updating `index.md` and `log.md`.

Why Reconcile Contradictions Every Time

Suppose a paper says "Model Z is best" but the wiki has a page that says "Model X is best."

- **Traditional RAG:** both pages coexist; the agent may retrieve the old page and answer incorrectly with confidence.
- **OKF (agentic ingest):** every time knowledge is added, it **checks the surroundings** for conflicts, supersessions, and confirmations, then writes those relationships explicitly — the old page gets flagged "superseded, see B" and the new page gets context "updated from A." **Both pages are correct simultaneously** — the wiki has a consistent "present tense."

Tips

- **Ingest one source at a time** and stay present to review it — read the summary, check the updates, guide the focus.
- A single source may touch 10–15 pages in the wiki (concepts + entities + indexes + log).
- Start ingest **selectively** — no need to pour everything in at once. The cold-start problem is smaller than you think, because the topics you care about most tend to be covered first.

Next: once you have knowledge in the wiki, how do you query and search it? → Query and Search

Query and Search

There are two ways to retrieve knowledge from the wiki, depending on its size.

Method 1 — Query via Index (Small Wiki)

For small wikis (up to ~150 pages), reading `index.md` is sufficient:

1. Read `wiki/index.md` first to find relevant concepts.
2. Drill into those specific concept files.
3. Answer **only from the concepts loaded** and **always cite the Concept ID**.
4. If the coverage is incomplete, say so directly and offer to ingest additional sources.

Via agent: *"What does the wiki say about WAU?"* → agent reads the index → opens the concept → answers with citations.

Method 2 — Search with an Index (Large Wiki)

Beyond ~150 pages, scanning the index becomes slow. Build a search index instead:

```
# Build a BM25 index (once / rebuild when content changes)
python3 tools/okf-index.py build ./wiki

# Search
python3 tools/okf-search.py "how is WAU defined" --bundle ./wiki -k 8
```

Results (ranked by BM25):

```
mode: bm25-only (no embeddings — run okf-embed.py build)
   2.675  metrics/weekly-active-users   [Metric]  Number of unique
customers with orders...
   2.463  tables/customers              [BigQuery Table]  ...
```

> **What is BM25:** A keyword-based retrieval algorithm that scores relevance. Excellent for matching exact terms and codes (e.g. policy codes, column names).

> The tool is pure Python with no additional dependencies.

Hybrid Search (BM25 + Semantic)

If keyword search alone doesn't capture enough meaning, add a semantic layer using a local embedding model (Ollama):

```
ollama pull nomic-embed-text          # once (on-prem)
python3 tools/okf-embed.py build ./wiki   # build embeddings → wiki/.okf-embed.json
python3 tools/okf-search.py "active customers" --bundle ./wiki
# → mode: hybrid (bm25+semantic, RRF)
```

`okf-search.py` will **fuse BM25 + semantic results using Reciprocal Rank Fusion (RRF)** automatically.

> **Always safe:** If embeddings haven't been built yet, or Ollama isn't running, search will **fall back to BM25 automatically** and report the mode — semantic is a pure upgrade, not a hard dependency.

(Architecture details for hybrid search are in the Search at Scale and Semantic Search chapter.)

Comparison

Situation	Use
Small wiki (< ~150 pages)	Read `index.md` directly
Large wiki, matching keywords/codes	`okf-search.py` (BM25)
Need to match meaning/synonyms	Hybrid (BM25 + semantic)

Next: how to add and edit concepts → Adding and Editing Concepts

Adding and Editing Concepts

Start from the Template

Use `tools/concept-template.md` as your starting point:

```
---
type: Concept              # required — choose from the controlled vocabulary in AGENTS.md
title: <human-readable name>
description: <one-line summary — used to decide whether an agent will load this file>
resource: <URI of the real-world thing, or delete this line if purely abstract>
tags: [<tag1>, <tag2>]
timestamp: 2026-06-15T00:00:00Z   # ISO 8601 UTC
---

# <Title>
Use structure: headings, short bullets, tables — rather than long paragraphs.

# Related
Link to other concepts using relative paths, e.g. [orders](../tables/orders.md).
The relationship type belongs in this sentence, not in the link text itself.

# Citations
Cite the sources in raw/ that were used to synthesize this concept.
```

Steps to Add / Edit a Concept

1. Copy the template → name the file (kebab-case, stable).
2. Set `type` correctly (from the controlled vocabulary).

3. Write the body using structure + relative links.
4. Update `tags` + `timestamp`.
5. Update the `index.md` of that directory.
6. Add an entry to `wiki/log.md`.
7. Run `okf-validate.py`.

Conventional Section Order

Use when appropriate, in this order (derived from Google's enrichment prompt):

```
# Overview                  ← 1-3 paragraph intro: what it is, how it's used
# Schema                    ← column/field summary (nested RECORDs as sub-sections/tables)
# Common query patterns     ← 1-3 SQL snippets (```sql code blocks)
# Joins                     ← which concepts this joins to, via which keys
# Citations                 ← references (format: [1] [Title](url))
```

Example Reference Concept (join)

File `wiki/references/joins/orders__customers.md`:

````
---
type: Reference
title: Orders → Customers join
description: How to join the orders table to customers via customer_id
tags: [join, sales]
timestamp: 2026-06-15T00:00:00Z
---

Join [orders](../../tables/orders.md) with [customers]
(../../tables/customers.md)
via `customer_id` (many orders → one customer).

# Common query patterns
```sql
SELECT c.email, COUNT(*) AS orders, SUM(o.total) AS ltv
FROM orders o JOIN customers c USING (customer_id)
GROUP BY c.email;
```
````

After Editing — Definition of Done

- [ ] Concept has a non-empty `type` and a sharp `description`
- [ ] `timestamp` updated to current
- [ ] Relevant `index.md` updated
- [ ] Entry added to `log.md` under today's date
- [ ] `python3 tools/okf-validate.py ./wiki` passes

Next: validate and visualize the wiki → Validate and Visualize

Validate and Visualize

Validate — Check Conformance

Run this after every edit:

```
python3 tools/okf-validate.py ./wiki
# → ✓ CONFORMANT with OKF v0.1  (0 warning(s), 0 info)
```

Conformance Criteria (OKF v0.1)

A bundle passes when:

1. Every `.md` file that is not a reserved file has **parseable YAML frontmatter**
2. Every frontmatter has a **non-empty `type` field**
3. Reserved files (`index.md`, `log.md`) that exist follow the defined structure

Result Levels

Level	Meaning	Example
✗ **error**	Not conformant (must fix)	Missing frontmatter / missing `type` / index.md has disallowed frontmatter
! **warn**	Passes, but should fix	Link starts with `/` (breaks GitHub) / log heading is not ISO
· **info**	Not a problem	Broken link (spec §5.3 permits this)

> Consumers **must not reject** a bundle because of: missing optional fields, unknown `type`, extra keys, broken links, or a missing `index.md` — this is "permissive consumption," which keeps OKF usable even as bundles grow or get refactored.

Visualize — View the Knowledge Graph

```
python3 tools/okf-viz.py ./wiki --name "My Wiki"
# → wiki/viz.html  (single file, open in a browser)
```

`viz.html` is a **single self-contained HTML file** — it embeds the library (Cytoscape + marked) and the bundle data directly inside, **fetching nothing from the network when opened**. Suitable for air-gapped environments, file sharing, or committing alongside the bundle.

What the Viewer Shows

- **Force-directed graph** of all concepts, colored by `type`, with edges drawn from links in the content
- **Detail panel** for the selected concept: frontmatter + rendered body
- **"Cited by"** — backlinks (which other concepts link to this one)
- **Search box** (matches title/id/tags), **type filter**, and toggleable layouts

> By default it **embeds libraries from `tools/vendor/`**, enabling true air-gap use. To use a CDN instead, pass `--cdn`.

Try the Live Demo

Below is the `viz.html` of the example wiki included in this project (click a node to see its details; try searching and filtering by type):

🕸 **Interactive graph** — shown on the web only (iframes don't render in the PDF). View it online at: `https://supachai-j.github.io/open-knowledge-format-starter/en/viz-example.html`

Open full-screen →

Make It a Habit

Combine both commands after every editing session:

```
python3 tools/okf-validate.py ./wiki && python3 tools/okf-viz.py ./wiki
```

At an organizational level, CI runs validate on every PR and regenerates the viz automatically (see Part 6).

That wraps up the day-to-day usage section. Next, see guidelines for writing well → Authoring Guidelines and Anti-Patterns

Worked Example: a Bookstore KB

This chapter walks the whole path from zero to searchable — a small knowledge base for an online bookstore, with real commands. Use it as a template for your own domain.

1. Scaffold (init)

```
python3 tools/okf-init.py ./bookstore-kb
cd bookstore-kb
```

You get `AGENTS.md` + `wiki/{index.md, log.md, getting-started.md}` + `raw/`.

2. Add knowledge (simulated ingest)

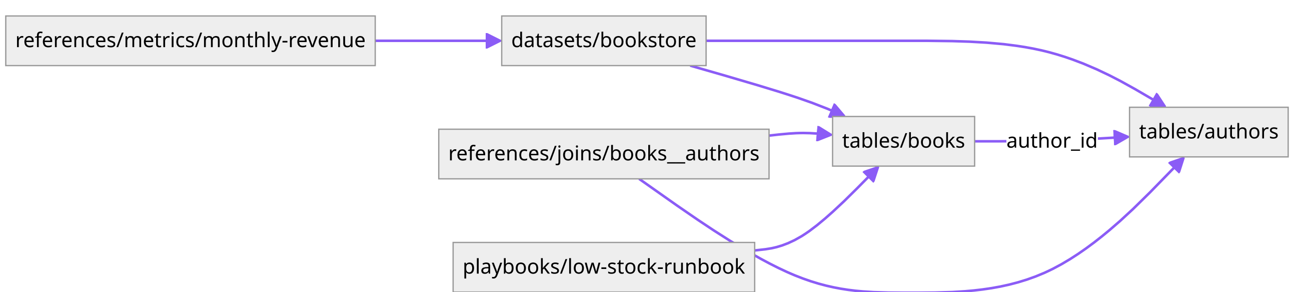
Create concepts in the canonical layout — two tables, a metric, a join, a playbook. Example `wiki/tables/books.md`:

```
---
type: BigQuery Table
title: Books
description: One row per book in the catalog.
tags: [catalog, books]
timestamp: 2026-06-16T00:00:00Z
---

# Schema
| Column | Type | Description |
| :--- | :--- | :--- |
| book_id | STRING | Book id (PK) |
| author_id | STRING | FK to [authors](authors.md) |
| price | NUMERIC | Price (USD) |
| stock | INT64 | Units in stock |

# Joins
Join [authors](authors.md) on `author_id` — see [Books → Authors]
(../references/joins/books__authors.md).
Low stock → [restock runbook](../playbooks/low-stock-runbook.md)
```

Do the same for `authors`, `references/metrics/monthly-revenue`, `references/joins/books__authors`, and `playbooks/low-stock-runbook`, then update `wiki/index.md` + `wiki/log.md`.

> In real use: drop a source into `raw/` and have the agent run the supervised INGEST flow (Ingest chapter) — it extracts claims, shows them for approval, then writes the concepts and updates index/log for you.

The resulting KB is this graph:



3. Validate conformance

```
python3 tools/okf-validate.py ./wiki
# → ✓ CONFORMANT with OKF v0.1  (0 warning(s), 0 info)
```

4. Search

```
python3 tools/okf-index.py build ./wiki                       # build the BM25 index
python3 tools/okf-search.py "how to join books to authors" --bundle ./wiki
# → references/joins/books__authors  [Reference]   ranked #1
```

Add semantic (optional, on-prem):

```
ollama pull nomic-embed-text
python3 tools/okf-embed.py build ./wiki
python3 tools/okf-search.py "who wrote the novels" --bundle ./wiki
# → mode: hybrid (bm25+semantic, RRF) — finds authors even with no keyword overlap
```

5. Visualize

```
python3 tools/okf-viz.py ./wiki --name "Bookstore KB"
open wiki/viz.html        # one file, opens offline
```

6. (Team) Share via MCP

To let the whole team/agents use it: commit `wiki/` to an internal git server and bring up the MCP server pointed at this bundle — agents can then `okf_search` (hybrid) and propose changes via PR/lease.

Summary

```
okf-init → (ingest concepts) → okf-validate → okf-index/okf-search → okf-viz → [MCP share]
```

That's the full loop! Next, see Authoring Well to grow the KB with quality.

Writing Guidelines and Anti-patterns

These guidelines layer on top of the spec (which only enforces `type`) to make the wiki reliable for both humans and agents.

Golden Rules

1. **One concept per file** — do not cram three topics into one file
2. **Write `description` for the agent** — this is the single line an agent reads to decide whether to load the file; make it specific
3. **Structure beats paragraphs** — headings, bullets (`**key** — value`), tables; models extract information from structured Markdown more accurately
4. **identity = path** — use stable kebab-case filenames; renaming breaks inbound links
5. **Consistent `type`** — use a controlled vocabulary
6. **Always cite sources** — every synthesized claim should be traceable to a file in `raw/`; add them under `# Citations`

Anti-pattern Table

Anti-pattern	Why it is bad	Do this instead
Automated background ingest	Accumulates noise as fast as signal; the wiki decays silently	Make ingest an explicit human-triggered command with review
Dumping raw PDFs into `wiki/`	Unreliable retrieval; breaks synthesis	Synthesize into a concept Markdown file; keep the raw file in `raw/`
Over-stuffed frontmatter	Noise reduces search precision	Keep `tags` semantic and minimal
Skipping heading levels (H1 → H3)	Breaks document structure for the model	Maintain H1 → H2 → H3 order
Paragraphs interspersed in lists	The list splits into fragments in the parser's view	Use nested paragraphs, or close the list first

Anti-pattern	Why it is bad	Do this instead
Inconsistent `type` /field names	Tools cannot aggregate the data	Use a controlled vocabulary
Vague anchor text ("click here")	No topic signal for the LLM	Use descriptive link text
Absolute paths starting with `/`	Breaks GitHub rendering	Use relative paths
Sacrificing readability for machines	The wiki must serve humans too	Structure for machines; clarity for humans

Core Spec vs. Community Best Practices

Keep these separate:

- **OKF v0.1 spec (very small):** enforces only `type` + index/log rules + conformance rules
- **Best practices (in this book):** mostly from the LLM-wiki community and the Google reference implementation — e.g., confidence decay, hybrid search, grouping `references/` — these are supplementary patterns, not requirements

Points Where Sources Disagree

- **"RAG is dead"** — most engineers say wiki = Layer 1, RAG = Layer 2 fallback; it is not either/or
- **Absolute vs. relative links** — the spec recommends absolute, but implementations use relative (we follow the implementation)
- **Four-dimension freshness scoring** — a guideline promoted by a vendor (Atlan), not part of the spec

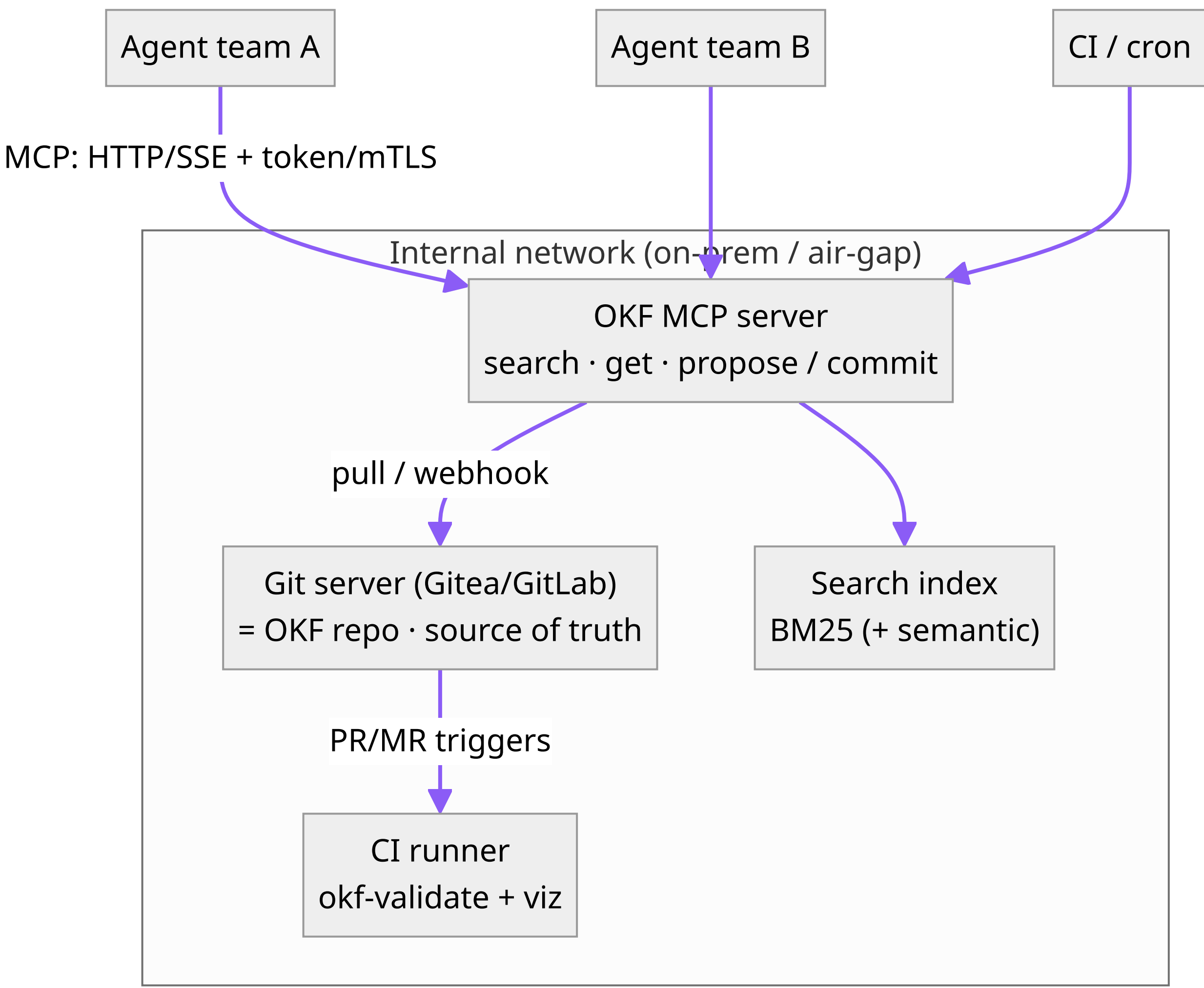
Next, move to the enterprise level → Architecture Overview

Architecture Overview (Enterprise)

When a knowledge base must be shared **across sessions and teams at the enterprise level** on-prem (internal network, works even air-gapped), the core idea fits in one sentence:

> **Git is the source of truth · the internal MCP server is the central access layer**
>
> Every session and every agent — regardless of framework or model — connects to **the same single internal MCP endpoint**. Git history *is* "cross-session memory" by nature · reads are instant · writes go through a gate (PR or lease)

Diagram



Components

#	Component	Self-host option	Role
1	**Git server**	Gitea / GitLab CE	Versioned source of truth (one repo per team, or monorepo + CODEOWNERS)
2	**OKF MCP server**	`server/okf_mcp_server.py`	Access layer that every agent connects to — read/search/propose; stdio + HTTP/SSE
3	**Search index**	`tools/okf-index.py` (+ embed)	Fast search once the wiki grows beyond ~150 pages
4	**CI gate**	Gitea Actions / GitLab CI	Blocks non-conformant merges + regenerates viz
5	**Reverse proxy**	Caddy / Traefik / nginx	TLS + auth (token / OIDC / mTLS) in front of MCP

Read Path (normal case — fast, no lock)

1. Agent calls `okf_search("how is WAU defined")` → receives ranked Concept IDs
2. `okf_get_concept("metrics/weekly-active-users")` → frontmatter + body loaded into context
3. `okf_read_index()` when progressive disclosure exploration is needed

Concurrent reads are unlimited with no contention.

Write Path

Two models to choose from (next chapter):

- **PR-gated (default):** write via branch + PR → CI checks → human/curator merges — safe, with audit/review
- **Lease/lock:** for heavy-write teams — per-concept lease prevents collision, writes directly to a shared branch

Meaning of "Cross-session / Cross-team"

- **Cross-session:** no per-session state — a new session `git pull`s everything a previous session wrote; the wiki compounds automatically
- **Cross-team:** *monorepo* + `CODEOWNERS` per subtree **or** *federated bundles* (one repo per domain, MCP server mounts multiple bundles and namespaces them by bundle name, e.g. `sales:tables/orders`)

Next: deploy it → Self-host Setup

Self-host Setup

`deploy/docker-compose.yml` brings up the full internal stack on a single VM — no external exposure, works even air-gapped.

Stack Overview

```
gitea   →  source of truth (git server)
okf-mcp →  access layer that agents connect to (built from deploy/Dockerfile)
proxy   →  Caddy: TLS + auth (token/OIDC/mTLS) in front of MCP
```

Steps

```
cd deploy
cp .env.example .env        # set OKF_GIT_REMOTE, token, etc.
docker compose up -d
```

Required `.env` values:

```
OKF_GIT_REMOTE=http://gitea:3000/okf/knowledge.git   # repo that MCP will clone/pull
OKF_GITEA_API=http://gitea:3000/api/v1               # for opening PRs automatically from propose_change
OKF_GITEA_TOKEN=<token>
OKF_GITEA_OWNER=okf
OKF_GITEA_REPO=knowledge
OKF_TOKEN=<long-random>                              # token agents must send to the proxy
OKF_READONLY=0                                       # 1 = read-only replica
```

Connecting an Agent to the MCP Endpoint

In Claude Code (or any MCP client), point to the internal URL:

```
{ "mcpServers": { "okf": {
    "transport": "http",
    "url": "https://okf.internal.example/mcp",
    "headers": { "Authorization": "Bearer ${OKF_TOKEN}" }
} } }
```

Local / Dev Mode (no network required)

Test the server without standing up the full stack:

```
python3 server/okf_mcp_server.py                       # stdio transport
# or HTTP:
OKF_MCP_TRANSPORT=streamable-http OKF_MCP_PORT=8765 \
  python3 server/okf_mcp_server.py
```

Tools the Server Exposes to Agents

Tool	What it does
`okf_search(query, k, type)`	Hybrid search (BM25 + semantic if available)
`okf_get_concept(id)`	Returns the frontmatter + body of a concept
`okf_list_concepts(prefix)`	Lists id/type/description
`okf_read_index(path)`	Reads `index.md` (progressive disclosure)
`okf_propose_change(...)`	Writes via branch + PR (PR mode)
`okf_acquire_lease / renew / release / list_leases`	Acquires write rights (lease mode)

Tool	What it does
`okf_commit_concept(..., token)`	Writes directly while holding a lease (lease mode only)

CI Conformance Gate

Set branch protection on `main` to require this check — non-conformant bundles cannot be merged.

- **Gitea Actions:** `.gitea/workflows/conformance.yml`
- **GitLab CI:** `ci/.gitlab-ci.yml`

Both run `okf-validate.py` (blocks on failure) + rebuild the index + regenerate `viz.html`.

Basic Security

- The MCP endpoint is behind a reverse proxy that handles TLS + auth (details in the Security and Governance chapter)
- `raw/` is excluded from the bundle (`.gitignore` prevents accidentally committing private data)
- Air-gap: `viz.html` bundles its libraries inline; semantic search uses on-premises Ollama — nothing leaves the network

Next: choose a write model → Write Models: PR-gated and Lease

Write Models: PR-gated and Lease

When multiple agents write knowledge concurrently, you must choose how to manage concurrency. The starter supports two models (plus a third as a concept). Select with the `OKF_WRITE_MODE` env var.

Model 1 — PR-gated (default)

`okf_propose_change(...)` **does not write to `main`** — instead it:

1. Creates branch `okf/<concept>-<id>` from `main`
2. Writes the file, commits, and pushes to the internal git server
3. Opens a PR/MR via the git server API and returns the URL
4. CI runs `okf-validate.py` (+ regenerates viz) → a human/curator reviews and merges

Full enterprise properties: **audit trail** (git log), **review/diff**, **rollback** (git revert), **no write conflicts** (merges applied one at a time in order), **quality gate** (CI + review)

Once merged to `main` → webhook → MCP server `pull` → reindex → all sessions see the update

> Suitable when **reviewing every change matters more than speed**

Model 2 — Lease/lock (direct write)

Enable with `OKF_WRITE_MODE=lease` for heavy-write teams. Uses a **lease (TTL-based write reservation)** to prevent two agents from editing the same concept simultaneously.

```
# per-agent flow:
okf_acquire_lease("tables/orders", ttl_seconds=300)   # → {token, expires_at}
# ... make edits ...
okf_commit_concept("tables/orders", frontmatter, body, token=...)   # server checks lease then write+commit+push
okf_release_lease("tables/orders", token=...)         # release when done
```

another agent requests the same concept ····locked··→ acquire_lease (TTL) → edit concept → commit_concept (verify token) → pull --rebase + push → release_lease

Another agent requesting the same concept receives `{error:"locked", held_by}` → it works on something else instead. Concurrency safety: the lease prevents duplicate concepts + `git pull --rebase` before push handles commits to different files (auto-merge).

Lease properties:

- **Advisory + self-expiring TTL** — a crashed agent does not leave a concept permanently locked (an expired lease can be "stolen")
- **Token-verified** — only the lease holder can commit/renew/release
- **Single-authority** — one MCP server issues all leases using atomic file creation (`O_CREAT|O_EXCL`); git is serialized with a lock

CLI for ops: `python3 tools/okf-lease.py list` · `... break <concept>` (admin force-release)

> Suitable when **write throughput matters more than reviewing every change**

Model 3 — Append-only proposals + curator (concept)

Agents drop proposals into `inbox/`; a **single curator agent** collects them, resolves conflicts, then merges into the wiki — the highest quality gate, consistent with the supervised-ingest philosophy (not yet implemented in the starter)

Comparison

Model	Speed	Review	Conflict	Best for
PR-gated	Moderate	Every change	None (sequential merge)	Governance, general teams
Lease/lock	Fast	Lightweight	Prevented by lease	Heavy-write teams
Curator	Slow	Maximum	Curator handles it	Quality above all else

> 💡 **Both modes can run against the same repo** — a heavy-write team uses the server in lease mode; everyone else uses PR-gated, all pointing to the same git remote

Next: search as the wiki scales → Search at Scale and Semantic Search

Search at Scale and Semantic Search

As the wiki grows, scanning a flat `index.md` becomes slow. This chapter covers search that scales, entirely on-prem.

By Wiki Size

Size	What to use
< ~150 concepts	`index.md` progressive disclosure is sufficient
> ~150	Build a BM25 index (`okf-index.py build`) — `okf_search` uses it automatically
Recall is insufficient	Add a semantic + RRF layer (below)
Very large / multi-team	Federate bundles, one MCP server per domain, cache index in memory and rebuild via webhook

Semantic Layer (on-prem)

Embed concepts using a self-hosted embedding model (via Ollama) — nothing leaves the network:

```
ollama pull nomic-embed-text          # one-time
python3 tools/okf-embed.py build ./wiki   # → wiki/.okf-embed.json
python3 tools/okf-search.py "active customers" --bundle ./wiki
# → mode: hybrid (bm25+semantic, RRF)
```

Reciprocal Rank Fusion (RRF)

`okf-search.py` combines **BM25 (lexical)** and **semantic (vector)** results using RRF:

```
fused(doc) = Σ_signals  1 / (RRF_K + rank)      (RRF_K = 60)
```

Each signal ranks its own set of docs, then the reciprocal of each rank is summed — docs that rank well across multiple signals rise to the top, without any single signal dominating.

Why two signals:

- **BM25** excels at exact keyword/code matching (policy codes, column names)
- **semantic** excels at meaning/synonym matching ("active customers" ↔ "weekly active users")

Automatic Fallback (important)

> If embeddings have not been built **or** Ollama is not running → search **automatically falls back to BM25** and reports the mode so you know. Semantic search is therefore a **purely opt-in upgrade with no hard dependency**.

This makes the system resilient: machines without Ollama can still search using BM25.

Scaling Further

- **Federate bundles** — separate repo per domain/team
- **One MCP server per domain** behind a single gateway
- **Cache index in memory** and rebuild incrementally when a webhook signals a merge
- Indexes/embeddings are **generatable artifacts** (built by CI / MCP server on demand) — no need to commit (`.okf-index.json`, `.okf-embed.json` are in `.gitignore`)

Next: security and governance → Security and Governance

Security and Governance

A self-hosted internal system must control who can read/write what, and must be auditable after the fact. OKF gets much of its governance for free from git.

Two-Layer Access Control

Git Layer

- Org/team membership on Gitea/GitLab
- **Branch protection** on `main` — enforces passing CI + review before merge

MCP Layer

Place the HTTP transport behind a reverse proxy with:

- **mTLS** (service-to-service) **or OIDC/SSO** (for humans) + a bearer token per agent identity
- Map identity → role:

role	allowed
`reader`	search / get / list only
`proposer`	+ `okf_propose_change` (branch/PR only, cannot touch `main`)
`curator`	may merge (via git server, not via MCP)

Set `OKF_READONLY=1` to run an MCP replica in read-only mode.

Caddy (proxy) Example — Bearer Token

```
okf.internal.example {
    # internal CA / self-signed for air-gap:
    #   tls /etc/caddy/okf.crt /etc/caddy/okf.key
    # or mTLS:
    #   tls { client_auth { mode require_and_verify trusted_ca_cert_file
/etc/caddy/internal-ca.crt } }

    @noauth not header Authorization "Bearer {$OKF_TOKEN}"
    respond @noauth "Unauthorized" 401
    reverse_proxy okf-mcp:8765
}
```

For SSO, replace the bearer check with `forward_auth` to an OIDC proxy (e.g. oauth2-proxy) and map identity → role.

Secrets and PII

- **`raw/` is not included in the bundle** and is already in `.gitignore` — prevents accidentally pushing private or large data
- **Do not put credentials in concepts** — concepts are knowledge, not a secrets store
- PR review helps catch sensitive data before it reaches `main` (another reason PR-gated writes are good in organizations)

Air-Gap (Closed Networks)

- `viz.html` embeds its libraries (Cytoscape + marked) inline — it does not fetch from a CDN at render time
- Semantic search uses a self-hosted embedding model (Ollama) — nothing leaves the network
- All tools are Python stdlib only (except the MCP server, which uses the `mcp` package — installable from an internal mirror)

Audit & Rollback (from git)

Task	Command
Who changed WAU?	`git log --follow wiki/metrics/weekly-active-users.md`
Revert incorrect knowledge	`git revert <sha>` → PR → merge
Change timeline	Read `wiki/log.md` or `git log`

Governance for Free

Because the wiki is files in git, every change has a **diff, blame, review, history, and rollback** — just like normal software development. Knowledge maintenance becomes an engineering workflow the team already knows.

End of the organizational section. See the appendix for reference → Tool Reference (CLI)

Tool Reference (CLI)

All tools are pure Python stdlib, located in `tools/` (or `scripts/` if installed as a skill).

okf-init.py — Create a New Bundle

```
python3 tools/okf-init.py [target_dir] [--force] [--date YYYY-MM-DD]
```

Creates `AGENTS.md` + `wiki/{index.md, log.md, getting-started.md}` + `raw/` · Does not overwrite a non-empty `wiki/` unless `--force`

okf-validate.py — Check Conformance

```
python3 tools/okf-validate.py [wiki_dir]
```

Exits 0 if conformant, 1 if not · error = missing frontmatter/`type`, index.md rule violation · warn = `/`-prefixed links, non-ISO log dates · info = broken link

okf-viz.py — Generate Graph Viewer

```
python3 tools/okf-viz.py [bundle] [-o out.html] [--name "Name"] [--cdn]
```

Produces a single self-contained `viz.html` (embeds Cytoscape + marked) · `--cdn` = load libraries from CDN instead of embedding

okf-index.py — BM25 Search Index

```
python3 tools/okf-index.py build [bundle] [-o index.json]
python3 tools/okf-index.py query "question" [-k 8] [--type Metric]
```

okf-embed.py — Embeddings (Ollama)

```
python3 tools/okf-embed.py build [bundle]
python3 tools/okf-embed.py query "question" [-k 8]
```

env: `OKF_OLLAMA_URL` (default `http://localhost:11434`), `OKF_EMBED_MODEL` (default `nomic-embed-text`)

okf-search.py — Hybrid Search (BM25 + semantic, RRF)

```
python3 tools/okf-search.py "question" [--bundle ./wiki] [-k 8] [--type ...] [--bm25-only]
```

Automatically falls back to BM25 if embeddings are absent or Ollama is not running.

okf-lease.py — Lease/Lock Concurrency

```
python3 tools/okf-lease.py acquire <concept> --owner <id> [--ttl 300]
python3 tools/okf-lease.py renew   <concept> --owner <id> --token <tok> [--ttl 300]
python3 tools/okf-lease.py release <concept> --owner <id> --token <tok>
python3 tools/okf-lease.py list
python3 tools/okf-lease.py break   <concept>      # admin force-release
```

env: `OKF_LEASE_DIR` (lease storage location), `OKF_LEASE_TTL` (default 300)

okf-selftest.sh — exercise the whole toolchain

```
bash tools/okf-selftest.sh
```

Runs 10 end-to-end checks (init → validate both a clean bundle and a deliberately-broken one → index → search → air-gap viz → lease → embed/hybrid if Ollama is up) · exits non-zero on any failure · suitable for CI.

install.sh — Install Skill

```
./install.sh                    # global  → ~/.claude/skills/okf
./install.sh --project          # project → ./.claude/skills/okf
./install.sh --dir <path>       # custom
./install.sh --uninstall
```

server/okf_mcp_server.py — MCP Access Layer

```
python3 server/okf_mcp_server.py            # stdio
OKF_MCP_TRANSPORT=streamable-http OKF_MCP_PORT=8765 python3
server/okf_mcp_server.py
```

Key env vars: `OKF_REPO_DIR`, `OKF_BUNDLE`, `OKF_BASE_BRANCH`, `OKF_WRITE_MODE` (`pr` | `lease`), `OKF_READONLY`, `OKF_GITEA_API/TOKEN/OWNER/REPO`, `OKF_AGENT_ID`

Frequently Asked Questions (FAQ)

How is OKF different from Obsidian / Notion?

They are quite similar (Markdown + frontmatter + links), but OKF is **specified** — it defines the minimal rules necessary for interoperability (e.g. `type` is required, reserved files, conformance rules) without mandating a tool. You can open an OKF bundle in Obsidian/MkDocs/Hugo directly because it is just Markdown.

Do I need Google Cloud / BigQuery?

No. OKF is vendor-neutral. The reference examples use BigQuery but `type` can be anything. This starter is not tied to any cloud provider.

Do I need an AI agent?

No. You can write concepts by hand yourself (they are just Markdown). AI agents simply handle the heavy lifting — summarizing, cross-referencing, filing, and bookkeeping.

Should links be relative or absolute?

Relative only. Do not prefix with `/` because it breaks GitHub rendering (see Linking). Although the spec suggests absolute links, the actual Google implementation uses relative ones.

Is a broken link an error?

No — it represents "knowledge not yet written." `okf-validate.py` reports it as info, not an error.

Can I fully automate ingest?

Strongly not recommended — a background daemon will accumulate noise until the wiki rots. Ingest should be a human-triggered command with a review step.

How large does the wiki need to be before search is necessary?

Around **~150 pages**. Before that, `index.md` is sufficient. Beyond that, use `okf-search.py` (BM25) and add semantic search when recall is insufficient.

Do I need Ollama to search?

No. BM25 works without Ollama. Semantic search is an optional upgrade. If Ollama is not running, search automatically falls back to BM25.

What do I do when multiple agents write concurrently?

Choose a write model: **PR-gated** (safe, with review) or **lease/lock** (faster, for write-heavy teams). See Write Models.

Does it work air-gapped (closed network)?

Yes — `viz.html` embeds its libraries, tools use stdlib only, semantic search uses on-prem Ollama, and git/MCP run internally.

Do `.okf-index.json` / `.okf-embed.json` need to be committed?

No — they are generatable artifacts and are already in `.gitignore`. CI/MCP server regenerates them on demand.

validate shows "missing type" — how do I fix it?

Every concept must have a non-empty `type` in its frontmatter. Add `type: ...` (chosen from the controlled vocabulary).

How do I migrate an existing wiki (Obsidian, etc.) into OKF?

In most cases: add a `type` field to each frontmatter, add `index.md` / `log.md`, convert links to relative paths, then run validate.

How do I verify the whole toolchain works?

Run `bash tools/okf-selftest.sh` — 10 end-to-end checks (init/validate/index/search/viz/lease + embed/hybrid if Ollama is up); exits non-zero on any failure. See the Worked Example for a hands-on full-loop walkthrough.

Glossary

Term	Definition
OKF (Open Knowledge Format)	An open specification for storing knowledge as a directory of Markdown files + YAML frontmatter (v0.1, Google Cloud, 2026)
Bundle	A directory containing all knowledge files — the unit of distribution (in this project: `wiki/`)
Concept	One unit of knowledge = one `.md` file
Concept ID	The file's path within the bundle with `.md` stripped, e.g. `tables/orders.md` → `tables/orders`
Frontmatter	A YAML block at the top of a file, delimited by `---`, storing metadata
Body	The Markdown content below the frontmatter
Link	A Markdown link between concepts = a relationship (untyped)
Citation	A link from a concept to an external source that supports a claim in the content
Reserved file	Reserved files: `index.md` (table of contents), `log.md` (change log)
Progressive disclosure	Showing the table of contents before opening the actual file — reduces context window overflow
Conformance	A bundle following the v0.1 rules (parseable frontmatter + `type` present + reserved files correctly structured)
type	The only required frontmatter field — specifies the kind of concept
Reference	A `type` used for synthesized knowledge (joins, metric definitions), typically under `references/`
RAG	Retrieval-Augmented Generation — fetching raw document chunks at query time and stuffing them into context
LLM-wiki pattern	A concept (Karpathy) where AI synthesizes knowledge into a continuously maintained Markdown wiki, instead

Term	Definition
	of re-fetching raw sources every time
Ingest	The process of taking raw sources and synthesizing them into wiki concepts (should be human-supervised)
Contradiction flag	A `> **CONTRADICTION FLAG**: ...` marker added when new information conflicts with existing content
BM25	A keyword-based ranking algorithm that scores document relevance
Embedding	A vector representing the semantic meaning of text (generated by a model, e.g. via Ollama)
Semantic search	Search by semantic similarity (cosine distance between embeddings)
Hybrid search	Combining BM25 + semantic search
RRF (Reciprocal Rank Fusion)	A method for merging ranked results from multiple signals: `Σ 1/(k + rank)`
MCP (Model Context Protocol)	An open standard for AI agents to connect to external tools and data
MCP server	A service that exposes tools (search/get/propose) wrapping a bundle for agent connections
PR-gated	A write model using branch + Pull Request + CI + review
Lease/lock	A write model that reserves exclusive rights to a concept with a TTL, preventing write conflicts
Lease	A TTL-based reservation that expires automatically, verified by a token
Curator	(Model 3) A single agent that aggregates proposals and merges them into the wiki
CODEOWNERS	A file defining owners per subtree (used to split responsibility across teams in a monorepo)
Federated bundles	Multiple repos/bundles per domain; MCP servers mount multiple bundles, namespaced by bundle name
air-gap	An environment with no internet connection (closed network)
Gitea / GitLab CE	Self-hostable git servers used as the source of truth

Term	Definition
AGENTS.md	A schema file that tells agents about the structure/rules/workflow (may be named `CLAUDE.md` / `GEMINI.md`)

References

Compiled references for the History and Foundations chapters. Numbers `[1]` – `[6]` correspond to the eras in the History chapter. Gathered through deep research (June 2026); primary sources are cited wherever possible.

[1] Era of Expert Systems & Knowledge Representation

1. Shortliffe, E. H. (1976). *Computer-Based Medical Consultations: MYCIN*. Elsevier/North-Holland.
2. Buchanan, B. G., & Shortliffe, E. H. (Eds.) (1984). *Rule-Based Expert Systems: The MYCIN Experiments of the Stanford Heuristic Programming Project*. Addison-Wesley.
3. Minsky, M. (1974). *A Framework for Representing Knowledge*. MIT AI Lab Memo 306.
4. Feigenbaum, E. A., Buchanan, B. G., & Lederberg, J. (1971). On Generality and Problem Solving: A Case Study Using the DENDRAL Program. *Machine Intelligence*, 6.
5. Lenat, D. B., & Guha, R. V. (1990). *Building Large Knowledge-Based Systems: Representation and Inference in the Cyc Project*. Addison-Wesley.

[2] Era of Ontologies & the Semantic Web

6. Berners-Lee, T., Hendler, J., & Lassila, O. (2001). The Semantic Web. *Scientific American*, 284(5), 34–43. https://www.lassila.org/publications/2001/SciAm.html
7. Gruber, T. (1993). A Translation Approach to Portable Ontology Specifications. *Knowledge Acquisition*, 5(2).
8. W3C (1999, rev. 2004, 2014). *Resource Description Framework (RDF)*. https://www.w3.org/RDF/
9. W3C (2004, rev. 2009). *OWL Web Ontology Language Overview*. https://www.w3.org/OWL/
10. schema.org (2011). *schema.org — shared structured-data vocabulary* (Google, Bing, Yahoo). https://schema.org/

[3] Era of Databases & Information Retrieval

11. Codd, E. F. (1970). A Relational Model of Data for Large Shared Data Banks. *Communications of the ACM*, 13(6), 377–387. https://dl.acm.org/doi/10.1145/362384.362685
12. Spärck Jones, K. (1972). A Statistical Interpretation of Term Specificity and Its Application in Retrieval. *Journal of Documentation*, 28(1), 11–21.
13. Robertson, S. E., Walker, S., Jones, S., Hancock-Beaulieu, M., & Gatford, M. (1994). Okapi at TREC-3. *Proceedings of TREC-3*.
14. Apache Software Foundation. *Apache Lucene* (Doug Cutting, 1999). https://lucene.apache.org/
15. Elastic (2010). *Elasticsearch* (Shay Banon). https://www.elastic.co/

[4] Era of Wikis & Personal Knowledge Management

16. Cunningham, W. (1995). *WikiWikiWeb* — Portland Pattern Repository. https://en.wikipedia.org/wiki/WikiWikiWeb
17. Wales, J., & Sanger, L. (2001). *Wikipedia*. Wikimedia Foundation. https://en.wikipedia.org/wiki/Wikipedia
18. Luhmann, N. (1981). Kommunikation mit Zettelkästen. In *Öffentliche Meinung und sozialer Wandel*. https://zettelkasten.de/introduction/
19. Obsidian.md (2020). *About Obsidian*. https://obsidian.md/about

[5] Era of AI: Embeddings, Vector Search & RAG

20. Mikolov, T., Chen, K., Corrado, G., & Dean, J. (2013). Efficient Estimation of Word Representations in Vector Space. arXiv:1301.3781. https://arxiv.org/abs/1301.3781
21. Singhal, A. (2012). Introducing the Knowledge Graph: things, not strings. *Google Blog*. https://blog.google/products/search/introducing-knowledge-graph-things-not/
22. Johnson, J., Douze, M., & Jégou, H. (2017). Billion-scale Similarity Search with GPUs (FAISS). arXiv:1702.08734. https://arxiv.org/abs/1702.08734
23. Devlin, J., Chang, M.-W., Lee, K., & Toutanova, K. (2018). BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding. arXiv:1810.04805 (NAACL-HLT 2019). https://arxiv.org/abs/1810.04805
24. Lewis, P., Perez, E., Piktus, A., et al. (2020). Retrieval-Augmented Generation for Knowledge-Intensive NLP Tasks. arXiv:2005.11401 (NeurIPS 2020).

https://arxiv.org/abs/2005.11401
25. Cormack, G. V., Clarke, C. L. A., & Buettcher, S. (2009). Reciprocal Rank Fusion Outperforms Condorcet and Individual Rank Learning Methods. *SIGIR 2009*.

[6] Present & Future: LLM-Wiki, OKF & Agent Memory

26. Karpathy, A. (2026). *LLM Wiki* [GitHub Gist]. https://gist.github.com/karpathy/442a6bf555914893e9891c11519de94f
27. McVeety, S., & Hormati, A. (2026, June 12). *How the Open Knowledge Format Can Improve Data Sharing*. Google Cloud Blog. https://cloud.google.com/blog/products/data-analytics/how-the-open-knowledge-format-can-improve-data-sharing
28. GoogleCloudPlatform (2026). *Open Knowledge Format — Specification & Reference Implementations* [GitHub]. https://github.com/GoogleCloudPlatform/knowledge-catalog/tree/main/okf
29. Packer, C., Wooders, S., Lin, K., et al. (2023). MemGPT: Towards LLMs as Operating Systems. arXiv:2310.08560. https://arxiv.org/abs/2310.08560

Reliability Notes

- **BM25 date** — The Okapi at TREC-3 paper (1994) is the primary citation, but the formula had been developing since the late 1980s; "~1994" refers to the key publication, not the first appearance.
- **OKF date** — Most secondary sources cite 12 June 2026, while the Google Cloud blog page shows 13 June (likely a timezone difference) — 12 June is used as the canonical publication date.
- **Karpathy's gist** — Sources place it between 3–4 April 2026 (X post on the 3rd, gist following on the 4th).
- The historical content is synthesised from primary sources with encyclopaedias and standards documents used as context. Readers should verify against the originals when high accuracy is required.