USER MANUAL · V3.2

Distilled context for your agent, without re-reading half the repo.

Single MCP hub · multi-project | 11 tools | qwen3-embedding 2560d | Redis HNSW · COSINE

> Usage guide for the **code-rag** system: what it does, how to get started, tool reference, workflows, how it works under the hood and best practices. Written both for a developer and for an agent working on an indexed repo. Product version: **v3.2** · Store: Redis Stack (HNSW COSINE) · Embeddings + LLM: Ollama. Deep technical reference: `docs/00-overview.md` and following.

Contents

---

1. What it actually does

Most agents explore a repo the expensive way: they open a big file, read hundreds of lines, extract one useful function and repeat elsewhere. That burns the token budget without you noticing.

code-rag replaces that with **structured retrieval**. It indexes the repo once —modules, contracts, AST symbols, docs, findings— and lets your agent bring back **only the context it actually needs**, already distilled: how a module connects, its API, who calls it, what it depends on, its side effects, the data models, known gotchas and past bugs.

> **The idea in one line:** code-rag isn't powerful because it's installed. It's powerful because the agent uses it *before* reading files the brute-force way — and because every debug leaves knowledge that surfaces next time.

It self-improves: every relevant bugfix is indexed as a *finding*, and a usefulness score (reinforcement by use + decay of the stale) makes the proven and fresh surface first. The marginal cost per debug drops over time.

---

2. Quick start

In this repository the MCP is already connected in **hub** mode: a single `code-rag` server serves all projects and resolves which one to use per request. There's nothing to configure per project and no slug to pass: the tool already queries the repo of the open workspace.

1 · See which projects exist

```
// lists the indexed projects and marks which one is open here
mcp__code-rag__list_projects
```

2 · Your first search. For any question —a bug, "where is the logic for X", "how does Y connect"— start with `search_context`. It's the front door and returns already-ranked context.

```
mcp__code-rag__search_context({ query: "how the result ranking is computed" })
```

Each result opens with `📂 <slug> · branch: <branch>` so you know which project it pointed to. To query **another** project, pass `project:"<slug>"` to any tool.

3 · Go deeper and edit. With the context in hand, drill down with `get_card`, `get_symbol` or `get_file_outline`. Only then open —with Read— **only the files you're going to modify**. The rest of the context already arrived through the MCP.

> **Setup on a new machine:** if the hub isn't registered (new team or an old per-repo install), run `npm run mcp:sync`: it registers the hub across all clients and cleans up the per-repo entries. Then open a **new session** so the client picks it up.

---

3. The step almost everyone skips

Having the MCP connected makes the tools available. It **does not guarantee** that the agent will stop opening giant files like a tourist with a flashlight. The difference between "this is amazing" and "I installed it and nothing changed" is usually a single instruction to the agent:

```
Before Grep/Read, use mcp__code-rag__search_context to bring in repo context.
Prefer symbol search, outlines and pinpoint retrieval over reading whole files.
```

In this repo that policy already lives in `CLAUDE.md`. If you work on another indexed project, give the agent an equivalent line: you teach it to *navigate*, not to rummage.

---

4. Mental model

Four ideas explain 90% of the system's behavior:

#	Concept	What it is
A	**Two tiers**	The **semantic** tier searches by intent (vector KNN, via `search_context`). The **structural** tier searches by name/location —signature, outline, shape, callers— with an exact lookup or a graph, cheap and GPU-free.
B	**Findings**	A non-obvious bug + its root cause + the fix, indexed. It's the unit of self-improvement: it's born with more weight than a regular card and surfaces when it's needed again.
C	**Usefulness**	A `[0.1, 3.0]` score per doc that **rises with use** (hits + feedback) and **drops with the nightly decay** of the old and rarely-used. It drives the ranking.
D	**Clean degradation**	If the box (Redis/Ollama) doesn't respond or the license doesn't pass, the tools **warn and let you keep going** with Grep/Read. The RAG is never a single point of failure for the session.

> **Redis is cache, the `.md` files are the truth.** Cards and findings live versioned in `cards/` and `knowledge/`: they're the canonical backup. Redis is a **rebuildable** vector index via `index:initial`. The knowledge survives a flush.

---

5. Tool reference

Twelve tools under `mcp__code-rag__*`. No need to choose: `search_context` is the door and already surfaces symbols; the rest are for going deeper. All accept an optional `project:<slug>`.

Search and context

Tool	What it does	Key inputs
`search_context`	**The star tool.** Already-distilled context + symbol hits, ranked. Use it BEFORE Grep/Read.	`query`, `filters?`, `k?`
`get_card`	Full card of ONE module when you already know which. Direct lookup, no KNN.	`id` | `path` | `symbol`
`get_module_map`	Index of an area/module without opening dozens of files.	`area`

Code structure

Tool	What it does	Key inputs
`get_file_outline`	Outline of a file (symbols + signature + lines) without opening it. Live JIT extraction.	`path`
`get_symbol`	Detail of a symbol: signature, doc, location, number of callers.	`id` | `name`
`find_references`	Who references a symbol (impact analysis). Heuristic by name.	`symbol`
`get_type`	The shape of an interface / type / enum + its uses.	`name`

Tabular data

Tool	What it does	Key inputs
`query_dataset`	**SELECT-only** SQL (DuckDB dialect) over indexed datasets (Excel/CSV/Parquet/SQLite...). For exact figures the precomputed aggregates don't cover. No args → lists datasets; with `dataset` and no `sql` → schema.	`dataset?`, `sql?`, `project?`

Self-improvement

Tool	What it does	Key inputs
`record_finding`	Indexes a debugging finding. **Requires** `confirmed:true` (usually via `/aprende`). Writes `knowledge/<slug>.md` + index.	`title`, `rootCause`, `fix`, `filesTouched`, `modules`, `symptom?`, `confirmed`
`feedback`	Adjusts the usefulness of a result: useful `+0.3` / useless `-0.5`. Improves future ranking.	`id`, `useful`

Index maintenance

Tool	What it does	Key inputs
`reindex`	Refreshes the STRUCTURAL index of the active project (symbols, functions, classes, module cards) so it doesn't drift after a code addition. **Doesn't index on the spot:** it enqueues the job in Redis and the box's job-runner runs it async (progress on the dashboard). **Reads COMMITS, not the working tree** — commit and push first or it won't see your changes. Requires `confirmed:true`.	`mode?` (`incremental` | `full`, default `incremental`), `confirmed`

Multi-project (hub)

Tool	What it does	Key inputs
`list_projects`	Lists the indexed projects and marks which one is open. To point at another, pass `project:<slug>` to the tool you use.	—

---

6. Workflows

New or unfamiliar repo `search_context` → `get_module_map` → `get_file_outline`. Search by intent, get oriented in the area, and only then look at the file outline. No blind spelunking.

Find and read a function `search_context` → `get_symbol` → `Read` (that only). Search, identify the symbol, pull its detail, and open only what you're going to edit. That's where the token savings are.

Impact analysis before changing something `find_references` → `get_type` → `Grep` (to confirm). Look at who references it and the shape of the type. Careful: `find_references` is heuristic by name — for critical changes, confirm with Grep.

A question of exact figures `search_context` → `query_dataset`. When the precomputed aggregates aren't enough (median for March filtering region X, percentiles, arbitrary group-by), write a SELECT and DuckDB runs it over the indexed dataset.

Close the session leaving knowledge behind you fix the bug → `/aprende` → `record_finding`. On close, the Stop-hook leaves a draft. `/aprende` writes the finding from the conversation and indexes it after your confirmation.

---

7. Knowledge bases and profiles

code-rag isn't only for code. The core is agnostic; a **profile** (`RAG_PROFILE`) declares, per knowledge base, what gets ingested and with what vocabulary the tools speak to you. No profile ⇒ `code` (zero migration).

Profile	What it ingests
`code`	AST symbols (ts-morph), per-module LLM cards, findings. The default behavior.
`docs`	Prose: MD, TXT, PDF, DOCX, RST. Chunked by headings (~600 tokens).
`data`	Tabular: Excel/CSV/TSV at 3 levels — schema, row-by-row records and aggregations.
`business`	Documents + data from CRM/ERP/MRP combined. Falls through to `query_dataset` when exact computation is needed.

The **universal ingestion** (v3.2) also covers HTML, JSON/JSONL/YAML/XML, PPTX, ODT, RTF, logs, `.eml` / `.msg` emails, QIF/OFX, Parquet, SQLite and images/scanned PDF via opt-in OCR (`RAG_OCR=1`). The binary parsers are optional dependencies: if they're missing, it warns and continues with the rest.

> **Your own domains without touching code:** a preset in `~/.coderag/profiles.json` declares a new domain (accounting, lab) that extends `docs` / `data` / `business` without editing `profiles.ts`. Accounting detection, for example, recognizes an account+debit+credit table and builds a trial balance.

---

8. How search works

`search_context` is not a grep with a fake mustache. The flow is:

1. **Asymmetric embedding.** The query is prefixed with a task instruction (`Instruct: … Query: …`); documents go without a prefix. Fixed embedder `qwen3-embedding:4b` at 2560d.

2. **KNN + rerank.** `k×3` neighbors are retrieved by cosine and re-ranked to `k` with the scoring: *similarity × (usefulness + recency + hits)*.
3. **Reinforcement.** What was retrieved gets a hit (raising its future ranking) and pulses to the dashboard. The result is truncated to ~4000 chars per hit so it doesn't blow the token limit.

Practical: use a precise query when you know the symbol's name; add `filters` (`type`, `module`, `kind`) when the repo is large; describe the intent in natural language (ES or EN) when you don't know what the code is called.

---

9. /aprende and self-improvement

The RAG's value grows with findings, but asking you to document at the end of every session doesn't scale. The loop automates it in three pieces:

`Stop-hook` → `/aprende` → `record_finding` → `feedback` → `decay`.

- **Stop-hook.** On session close it detects the edited files and leaves a draft; it suggests writing `/aprende`. It doesn't index (fail-safe: it never breaks the close).
- **/aprende.** Reconstructs the session from the conversation + git diff, writes the finding (symptom, root cause, fix), shows it to you and asks to confirm.
- **record_finding.** On confirm, it writes `knowledge/<slug>.md` and indexes with `usefulness:1.5`. Idempotent by title.

After that, every time a finding is retrieved it gets a hit; `feedback` adjusts its usefulness; and if it goes 60+ days without use, the nightly decay drops it to the floor (it stops surfacing, it's not deleted).

> **Don't index noise.** A good finding is reusable knowledge. Don't record renames, formatting, bumps or half-baked experiments. If there were several independent bugs, record **one per finding**, not a junk drawer.

---

10. Best practices

Do

- Start with `search_context`; go deeper with outline/symbol.
- Open only the files you're going to edit.
- Verify the symbol before quoting its implementation.
- Mark `feedback` when a result helped (or didn't).
- Close with `/aprende` if the fix was a non-obvious finding.
- Reindex with `reindex` (`confirmed:true`) when the codebase changes materially — but commit and push first: it reads commits, not the working tree.

Avoid

- "Read the whole repo and tell me what it does."
- Opening every candidate file just in case.
- Hunting through the source by hand until you find it.
- Trusting `find_references` for a critical change without Grep.
- Indexing trivial findings.

When prompting the agent: "*use search_context to locate the authentication flow*", "*start with the module_map, then drop down to the retry class*", "*bring only the billing methods*". You teach it to navigate, not to rummage.

11. Degradation and failures

Symptom	Likely cause → what to do
" ⚠ RAG unavailable"	The box is off / outside Tailscale, a model is loading, or the license didn't pass (deliberately indistinguishable). Keep going with Grep/Read without blocking and retry.
No results / empty index	The repo may not be indexed, or there's a **slug** mismatch (it's case-sensitive in Redis; a case difference Mac↔box searches an empty index). Confirm with `list_projects`.
First query very slow (~85 s)	LLM cold-start. The MCP uses `RAG_HTTP_TIMEOUT_MS=300000` and `keep_alive:-1` keeps the models resident: the next one is already fast.
query_dataset: "file on another machine"	Datasets live where they were indexed (the box). If the box indexed and you query from the Mac, the query must run where the files are.
The /aprende skill doesn't show in the menu	A limitation of the VS Code extension (not of code-rag): it still works if you type it directly. If it was never installed, run `npm run setup:claude`.

> **Cross-cutting principle:** the RAG must never be a single point of failure. Every tool wraps its body in `degrade()`: if something on the box doesn't respond, it returns a readable notice and the agent keeps exploring normally. It never throws or breaks the session.