MANUAL DE USO · V3.2

Contexto destilado para tu agente, sin releer medio repo.

MCP hub único · multi-proyecto | 11 tools | qwen3-embedding 2560d | Redis HNSW · COSINE

> Guía de uso del sistema **code-rag**: qué hace, cómo empezar, referencia de tools, flujos de trabajo, cómo funciona por dentro y buenas prácticas. Pensado tanto para un dev como para un agente que trabaje sobre un repo indexado. Versión del producto: **v3.2** · Store: Redis Stack (HNSW COSINE) · Embeddings + LLM: Ollama. Referencia técnica profunda: `docs/00-overview.md` y siguientes.

Contenido

---

1. Qué hace realmente

La mayoría de los agentes exploran un repo de la forma cara: abren un archivo grande, leen cientos de líneas, extraen una función útil y repiten en otro lado. Así se quema el presupuesto de tokens sin darse cuenta.

code-rag reemplaza eso por **recuperación estructurada**. Indexa el repo una vez —módulos, contratos, símbolos AST, docs, findings — y deja que tu agente traiga **solo el contexto que de verdad necesita**, ya destilado: cómo conecta un módulo, su API, quién lo llama, de qué depende, sus efectos, los modelos de datos, gotchas conocidos y bugs pasados.

> **La idea en una línea:** code-rag no es potente por estar instalado. Es potente porque el agente lo usa *antes* de leer archivos a lo bruto — y porque cada debug deja conocimiento que aflora la próxima vez.

Se auto-mejora: cada bugfix relevante se indexa como un *finding*, y un scoring de utilidad (refuerzo por uso + decaimiento de lo obsoleto) hace que lo probado y fresco aparezca primero. El coste marginal por debug baja con el tiempo.

---

2. Inicio rápido

En este repositorio el MCP ya está conectado en modo **hub**: un único servidor `code-rag` atiende todos los proyectos y resuelve cuál usar por request. No hay que configurar nada por proyecto ni pasar el slug: la tool ya consulta el repo del workspace abierto.

1 · Mirá qué proyectos hay

```
// lista los proyectos indexados y marca cuál está abierto acá
mcp__code-rag__list_projects
```

2 · Tu primera búsqueda. Ante cualquier duda —un bug, "dónde está la lógica de X", "cómo se conecta Y"— empezá por `search_context`. Es la puerta de entrada y devuelve contexto ya rankeado.

```
mcp__code-rag__search_context({ query: "cómo se calcula el ranking de resultados" })
```

Cada resultado abre con `📁 <slug> · branch: <branch>` para que sepas a qué proyecto apuntó. Para consultar **otro** proyecto, pasá `project:"<slug>"` a cualquier tool.

3 · Profundizá y editá. Con el contexto en mano, bajá al detalle con `get_card`, `get_symbol` o `get_file_outline`. Recién ahí abrí —con Read— **solo los archivos que vas a modificar**. El resto del contexto ya llegó por el MCP.

> **Setup en máquina nueva:** si el hub no está registrado (equipo nuevo o instalación vieja por-repo), corré `npm run mcp:sync`: registra el hub en todos los clientes y limpia las entradas por-repo. Después abrí una **sesión nueva** para que el cliente lo tome.

---

3. El paso que casi todos se saltan

Tener el MCP conectado hace que las tools estén disponibles. **No garantiza** que el agente deje de abrir archivos gigantes como turista con linterna. La diferencia entre "esto es increíble" y "lo instalé y no cambió nada" suele ser una sola instrucción al agente:

```
Antes de Grep/Read, usá mcp__code-rag__search_context para traer contexto del repo.
Preferí búsqueda de símbolos, outlines y recuperación puntual por encima de leer
archivos completos.
```

En este repo esa política ya vive en `CLAUDE.md`. Si trabajás sobre otro proyecto indexado, dejale al agente una línea equivalente: le enseñás a *navegar*, no a hurgar.

---

4. Modelo mental

Cuatro ideas explican el 90% del comportamiento del sistema:

#	Concepto	Qué es
A	**Dos tiers**	El **semántico** busca por intención (KNN vectorial, vía `search_context`). El **estructural** busca por nombre/ubicación —firma, outline, forma, callers— con lookup exacto o grafo, barato y sin GPU.
B	**Findings**	Un bug no obvio + su causa raíz + el fix, indexado. Es la unidad de auto-mejora: nace con más peso que una card normal y aflora cuando vuelve a hacer falta.
C	**Usefulness**	Score `[0.1, 3.0]` por doc que **sube con el uso** (hits + feedback) y **baja con el decay** nocturno de lo viejo y poco usado. Manda el ranking.
D	**Degradación limpia**	Si el box (Redis/Ollama) no responde o la licencia no pasa, las tools **avisan y dejan seguir** con Grep/Read. El RAG nunca es punto único de fallo de la sesión.

> **Redis es caché, los `.md` son la verdad.** Las cards y findings viven versionados en `cards/` y `knowledge/`: son el backup canónico. Redis es un índice vectorial **reconstruible** con `index:initial`. El conocimiento sobrevive a un flush.

---

5. Referencia de tools

Doce tools sobre `mcp__code-rag__*`. No hace falta elegir: `search_context` es la puerta y ya aflora símbolos; las demás son para profundizar. Todas aceptan un `project:<slug>` opcional.

Búsqueda y contexto

Tool	Qué hace	Inputs clave
`search_context`	**La tool estrella.** Contexto ya destilado + hits de símbolo, rankeado. Úsala ANTES de Grep/Read.	`query`, `filters?`, `k?`
`get_card`	Tarjeta completa de UN módulo cuando ya sabés cuál. Lookup directo, sin KNN.	`id` | `path` | `symbol`
`get_module_map`	Índice de un área/módulo sin abrir decenas de archivos.	`area`

Estructura de código

Tool	Qué hace	Inputs clave
`get_file_outline`	Outline de un archivo (símbolos + firma + líneas) sin abrirlo. Extracción JIT en vivo.	`path`
`get_symbol`	Detalle de un símbolo: firma, doc, ubicación, nº de callers.	`id` | `name`
`find_references`	Quién referencia un símbolo (análisis de impacto). Heurístico por nombre.	`symbol`
`get_type`	La forma de una interface / type / enum + sus usos.	`name`

Datos tabulares

Tool	Qué hace	Inputs clave
`query_dataset`	SQL **SELECT-only** (dialecto DuckDB) sobre datasets indexados (Excel/CSV/Parquet/SQLite…). Para cifras exactas que los agregados precalculados no cubren. Sin args → lista datasets; con `dataset` y sin `sql` → esquema.	`dataset?`, `sql?`, `project?`

Auto-mejora

Tool	Qué hace	Inputs clave
`record_finding`	Indexa un hallazgo de debugging. **Requiere** `confirmed:true` (normalmente vía `/aprende`). Escribe `knowledge/<slug>.md` + índice.	`title`, `rootCause`, `fix`, `filesTouched`, `modules`, `symptom?`, `confirmed`
`feedback`	Ajusta la utilidad de un resultado: útil `+0.3` / inútil `-0.5`. Mejora el ranking futuro.	`id`, `useful`

Mantenimiento del índice

Tool	Qué hace	Inputs clave
`reindex`	Refresca el índice ESTRUCTURAL del proyecto activo (símbolos, funciones, clases, cards de módulo) para que no driftee tras una ampliación del código. **No indexa en el acto:** encola el trabajo en Redis y el job-runner del box lo ejecuta async (progreso en el dashboard). **Mira COMMITS, no el working tree** — commiteá y pusheá antes o no verá tus cambios. Requiere `confirmed:true`.	`mode?` (`incremental` | `full`, default `incremental`), `confirmed`

Multi-proyecto (hub)

Tool	Qué hace	Inputs clave
`list_projects`	Lista los proyectos indexados y marca cuál está abierto. Para apuntar a otro, pasá `project:<slug>` a la tool que uses.	—

---

6. Flujos de trabajo

Repo nuevo o desconocido `search_context` → `get_module_map` → `get_file_outline`. Buscá por intención, orientate en el área, y recién después mirá el outline del archivo. Sin spelunking a ciegas.

Buscar y leer una función `search_context` → `get_symbol` → `Read` (solo eso). Buscá, identificá el símbolo, traé su detalle, y abrí únicamente lo que vas a editar. Ahí está el ahorro de tokens.

Análisis de impacto antes de cambiar algo `find_references` → `get_type` → `Grep` (confirmar). Mirá quién lo referencia y la forma del tipo. Ojo: `find_references` es heurístico por nombre — para cambios críticos, confirmá con Grep.

Una pregunta de cifras exactas `search_context` → `query_dataset`. Cuando los agregados precalculados no alcanzan (mediana de marzo filtrando región X, percentiles, group-by arbitrario), escribí un SELECT y DuckDB lo ejecuta sobre el dataset indexado.

Cerrar la sesión dejando conocimiento arreglás el bug → `/aprende` → `record_finding`. Al cerrar, el Stop-hook deja un draft. `/aprende` redacta el finding desde la conversación y lo indexa tras tu confirmación.

---

7. Bases de conocimiento y perfiles

code-rag no es solo para código. El núcleo es agnóstico; un **perfil** (`RAG_PROFILE`) declara, por base de conocimiento, qué se ingiere y con qué vocabulario le hablan las tools. Sin perfil ⇒ `code` (cero migración).

Perfil	Qué ingiere
`code`	Símbolos AST (ts-morph), cards LLM por módulo, findings. El comportamiento por defecto.
`docs`	Prosa: MD, TXT, PDF, DOCX, RST. Chunk por headings (~600 tokens).
`data`	Tabular: Excel/CSV/TSV en 3 niveles — esquema, registros fila-a-fila y agregaciones.
`business`	Documentos + datos de CRM/ERP/MRP combinados. Deriva a `query_dataset` cuando hace falta cálculo exacto.

La **ingesta universal** (v3.2) cubre además HTML, JSON/JSONL/YAML/XML, PPTX, ODT, RTF, logs, correos `.eml` / `.msg`, QIF/OFX, Parquet, SQLite e imágenes/PDF escaneado vía OCR opt-in (`RAG_OCR=1`). Los parsers binarios son dependencias opcionales: si faltan, avisa y sigue con el resto.

> **Dominios propios sin tocar código:** un preset en `~/.coderag/profiles.json` declara un dominio nuevo (contabilidad, laboratorio) que extiende `docs` / `data` / `business` sin editar `profiles.ts`. La detección contable, por ejemplo, reconoce una tabla cuenta+debe+haber y arma balance de sumas y saldos.

---

8. Cómo funciona la búsqueda

`search_context` no es un grep con bigote falso. El flujo es:

1. **Embedding asimétrico.** A la query se le antepone una instrucción de tarea (`Instruct: … Query: …`); los documentos van sin prefijo. Embedder fijo `qwen3-embedding:4b` a 2560d.

2. **KNN + rerank.** Se recuperan `k×3` vecinos por coseno y se re-rankean a `k` con el scoring: *similitud × (usefulness + recencia + hits)*.
3. **Refuerzo.** Lo recuperado suma un hit (sube su ranking futuro) y late al dashboard. El resultado se trunca a ~4000 chars por hit para no reventar el límite de tokens.

Práctico: usá una query precisa cuando sabés el nombre del símbolo; agregá `filters` (`type`, `module`, `kind`) cuando el repo es grande; describí la intención en lenguaje natural (ES o EN) cuando no sabés cómo se llama el código.

---

9. /aprende y la auto-mejora

El valor del RAG crece con los findings, pero pedirte que documentes al final de cada sesión no escala. El lazo lo automatiza en tres piezas:

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

- **Stop-hook.** Al cerrar la sesión detecta los archivos editados y deja un draft; sugiere escribir `/aprende`. No indexa (a prueba de fallos: nunca rompe el cierre).
- **/aprende.** Reconstruye la sesión desde la conversación + git diff, redacta el finding (síntoma, causa raíz, fix), te lo muestra y pide confirmar.
- **record_finding.** Al confirmar, escribe `knowledge/<slug>.md` e indexa con `usefulness:1.5`. Idempotente por título.

Después, cada vez que un finding se recupera suma un hit; `feedback` ajusta su utilidad; y si pasa 60+ días sin uso, el decay nocturno lo baja a piso (deja de aflorar, no se borra).

> **No indexes ruido.** Un finding bueno es conocimiento reutilizable. No registres renombres, formato, bumps ni experimentos a medio hacer. Si hubo varios bugs independientes, registrá **uno por finding**, no un cajón.

---

10. Buenas prácticas

Hacé

- Empezá por `search_context`; profundizá con outline/símbolo.
- Abrí solo los archivos que vas a editar.
- Verificá el símbolo antes de citar su implementación.
- Marcá `feedback` cuando un resultado ayudó (o no).
- Cerrá con `/aprende` si el fix fue un hallazgo no obvio.
- Reindexá con `reindex` (`confirmed:true`) cuando el codebase cambie de forma material — pero commiteá y pusheá antes: lee commits, no el working tree.

Evitá

- "Leé todo el repo y decime qué hace."
- Abrir todos los archivos candidatos por las dudas.
- Buscar a mano por el fuente hasta encontrarlo.
- Confiar en `find_references` para un cambio crítico sin Grep.
- Indexar findings triviales.

Al promptear al agente: "*usá search_context para ubicar el flujo de autenticación*", "*empezá por el module_map, después bajá a la clase de reintentos*", "*traé solo los métodos de facturación*". Le enseñás a navegar, no a rummagear.

11. Degradación y fallos

Síntoma	Causa probable → qué hacer
" ⚠ RAG no disponible"	El box está apagado / fuera de Tailscale, un modelo cargando, o la licencia no pasó (deliberadamente indistinguible). Seguí con Grep/Read sin bloquearte y reintentá.
Sin resultados / índice vacío	El repo puede no estar indexado, o hay un mismatch de **slug** (es case-sensitive en Redis; una diferencia de mayúsculas Mac↔box busca en un índice vacío). Confirmá con `list_projects`.
Primera consulta muy lenta (~85 s)	Cold-start del LLM. El MCP usa `RAG_HTTP_TIMEOUT_MS=300000` y `keep_alive:-1` mantiene los modelos residentes: la siguiente ya es rápida.
query_dataset: "archivo en otra máquina"	Los datasets viven donde se indexaron (el box). Si indexó el box y consultás desde la Mac, la consulta debe correr donde están los archivos.
El skill /aprende no aparece en el menú	Limitación de la extensión de VS Code (no de code-rag): igual funciona escribiéndolo directo. Si nunca se instaló, corré `npm run setup:claude`.

> **Principio transversal:** el RAG jamás debe ser un punto único de fallo. Toda tool envuelve su cuerpo en `degrade()`: si algo del box no responde, devuelve un aviso legible y el agente sigue explorando normal. Nunca lanza ni rompe la sesión.