Resumen de la refactorización: de CLI patch-only a MCP server (stdio)

1) Punto de partida (herramienta CLI)

El proyecto original era una utilidad de consola para operar archivos de un repo de forma segura:

- **`read <path>`**: lee archivo y devuelve texto + hashes (hash "strict" y hash normalizado).
- **`read-range <path> <start> <end>`**: devuelve un rango de líneas (1-based) + hashes.
- **`apply-patch <path> <expectedHash> <diff>`**: aplica unified diff **solo** si el hash esperado coincide ("patch-only").

Fortaleza del diseño original: el patrón *patch-only enforcement* ya existía (evita escrituras ciegas y reduce conflictos).

---

2) Objetivo del cambio

Convertir esa herramienta en un **servidor MCP** (Model Context Protocol) para que **Claude Code / Codex** lo puedan invocar como *tools*.

Motivo clave: pasar de "reglas en instrucciones de texto" (CLAUDE.md/AGENTS.md) a reglas **enforced por código**, es decir:

- aunque el modelo "quiera" escribir fuera del repo, **no puede**,
- aunque mande un diff inválido, **se rechaza**,
- aunque intente aplicar cambios sin hash base correcto, **falla**.

---

3) Enfoque elegido (el "más limpio", no wrapper)

Se implementó un modo nuevo **`serve`** que corre un MCP server sobre **stdio** (JSON-RPC 2.0):

- **stdin**: requests JSON-RPC (una línea por mensaje)
- **stdout**: responses JSON-RPC (una línea por mensaje)
- **stderr**: logs/diagnóstico (para no "ensuciar" stdout)

Esto mantiene el **core** (FileGateway + Diff/Validator/Applier + Hash/Encoding) sin refactor grande, y agrega una capa MCP arriba.

---

4) Arquitectura final (capas)

Capa MCP (nueva)

- `StdioMcpServer`: lifecycle MCP + routing JSON-RPC
- `McpToolHandlers`: `tools/list` + `tools/call` + mapeo a operaciones del core
- `RepoPolicy`: *strictness real* (root guard, deny/allow, límites, read vs write)

Capa Core (existente)

- lectura, snapshot, hashing, encoding detection
- parseo/validación/aplicación de unified diff
- patch-only (hash match)

---

5) Tools MCP expuestos

Equivalentes al CLI, pero como *tools*:

1. **`file.read`**
   - Entrada: `{ path }`
   - Salida: contenido + `structuredContent` (hashes, encoding, newline, flags)
2. **`file.read_range`**
   - Entrada: `{ path, startLine, endLine }`
   - Salida: rango de líneas + hashes
3. **`file.apply_patch_only`**
   - Entrada: `{ path, expectedHash, diffText }`
   - Salida: "OK" + hashes nuevos

Errores de negocio (path fuera de root, hash mismatch, diff inválido, etc.) vuelven como resultado del tool con:

- `isError: true`
- mensaje explicativo (para que el cliente/modelo pueda corregir la llamada)

---

6) Strictness: políticas del lado servidor (RepoPolicy)

La "estrictez de verdad" quedó en el servidor, no en texto.

Decisiones principales:

- **DeniedDirs (lectura)**: `{ ".git", "node_modules", ".vs" }`
- **MaxFileSize**: **10 MB** (tope duro)
- **Root guard anti-bypass**: el root se guarda con separador final para evitar prefijos falsos (ej. `...\McpHostEvil`).

- **Separación lectura vs escritura** (**implementado**):
  - lectura puede permitir más (por ejemplo `bin/`)
  - **escritura** (apply_patch) agrega denegados adicionales (**bin/obj**) para evitar parches sobre artefactos de build.

---

7) Ajustes acordados e implementados (1, 3 y 4)

Estos son los cambios que Claude dijo que implementó y que cierran puntos de robustez:

1. **Notificaciones**

- Antes: si `id == null` se trataba como notification.
- Ahora: es notification solo si **no existe** la key `"id"` (evita ignorar requests malformados con `"id": null`).

3. **Tamaño de `file.read`**

- En vez de agregar parámetro `max_chars`, se aplica un **límite interno** (p.ej. 500k chars):
  - si el archivo excede, devuelve `isError=true` y sugiere `file.read_range`.
- Beneficio: evita JSON enormes por stdout y fallas de buffering.

4. **Policy read vs write**

- `file.apply_patch_only` valida el path con `forWrite=true`, denegando `bin/obj` aunque lectura pueda permitirlos.
- Beneficio: evita que un patch termine tocando outputs/artefactos.

(2) protocolVersion echo: **no implementado** por decisión (el server declara lo que soporta).
(5) Portabilidad `.mcp.json`: tratado como pauta operativa, no como cambio de código.

---

8) Operación (cómo se usa)

- Se ejecuta como:
  - `Mcp.exe serve --root <ruta_del_repo>`
- El cliente (Claude Code / Codex) lo lanza automáticamente según configuración.

Claude Code (template): `.mcp.json` con `type:"stdio"`, `command`, `args`.
Codex (conceptual): `config.toml` declarando el MCP server y allowlist de tools.

---

9) Checklist de pruebas recomendado

- Lifecycle: `initialize` → `notifications/initialized` → `tools/list` → `tools/call`
- stdout limpio: nada de logs por stdout
- Path traversal: `..\..\` y bypass por prefijo (`RootEvil`) deben fallar
- Deny dirs: `.git/.vs/node_modules` deben fallar siempre
- Read limit: archivo > 500k chars debe devolver error sugeriendo `read_range`
- Write policy: apply_patch debe fallar en `bin/obj`
- Patch-only: hash mismatch falla; diff inválido falla; diff válido aplica y devuelve hashes nuevos