



jCodeMunch-MCP: Symbol-Precise Code Retrieval for Token-Efficient AI Agents

A TECHNICAL PERFORMANCE REPORT

Version	1.1.1
Author	J. Gravelle
Date	March 8, 2026
Repository	github.com/jgravelle/jcodemunch-mcp
PyPI	pypi.org/project/jcodemunch-mcp/

Open-source Model Context Protocol server for symbol-precise, token-efficient code retrieval.

jCodeMunch-MCP: Symbol-Precise Code Retrieval for Token-Efficient AI Agents

Contents

jCodeMunch-MCP: Symbol-Precise Code Retrieval for Token-Efficient AI Agents

A Technical Performance Report

1. Abstract
2. The Problem: Context Window Waste at Scale
3. Architecture
4. Benchmark: Three Real-World Repositories
5. Tool Reference
6. Security Model
7. MCP Client Compatibility
8. Token Economy: Scaling Impact
9. Limitations and Non-Goals
10. Benchmark Scorecard
11. Sources

1. Abstract

LLM coding agents routinely waste context window budget on retrieval rather than reasoning. The common pattern is coarse file-level exploration: open a source file, consume imports, comments, adjacent helpers, and unrelated declarations, then extract one target function or type from the result. jCodeMunch-MCP replaces that workflow with symbol-precise retrieval. It indexes repositories once using tree-sitter AST parsing, stores normalized symbol metadata plus byte offsets into cached raw files, and exposes the result through the Model Context Protocol (MCP) for low-overhead search and retrieval.

The current server exposes 12 MCP tools and operates over stdio, making it usable from any MCP-compatible client. In benchmarked open-source repositories, single-symbol and focused structural queries routinely reduce retrieval volume by 80 to 98%, with peak reductions near 99% in large-file needle-extraction cases. Representative baseline comparisons from the project documentation show “find a function” dropping from roughly 40,000 input tokens to about 200. The result is not a semantic program-analysis system or editing engine. It is a deterministic code-retrieval layer designed to make agent navigation materially cheaper, faster, and easier to reason about.

2. The Problem: Context Window Waste at Scale

Code agents typically retrieve at file granularity while the useful unit of information is usually a symbol, a small cluster of related symbols, or a structural overview. The project README uses three representative workloads to show the mismatch between those units.

Agent Task	Traditional (tokens)	jCodeMunch (tokens)	Reduction
Find a function	~40,000	~200	~99.5%
Understand a module's API	~15,000	~800	~94.7%
Explore repo structure	~200,000	~2,000	~99.0%

At Claude Opus pricing of \$15 per 1 million input tokens, 1,000 “find a function” queries cost:

- **Naive file reading:** 40,000,000 tokens → **\$600**

- **jCodeMunch:** 200,000 tokens → \$3
- **Cost avoided:** \$597

```
Naive:      1,000 × 40,000 = 40,000,000 tokens
jCodeMunch: 1,000 ×    200 =   200,000 tokens

Cost = tokens / 1,000,000 × $15
Naive cost      = 40 × $15 = $600
jCodeMunch cost = 0.2 × $15 = $3
Savings         = $597
```

The main engineering point is not just lower spend. It is that retrieval architecture dominates spend. Once code navigation becomes symbol-addressed instead of file-addressed, the marginal cost of exploration collapses.

3. Architecture

3.1 Indexing Pipeline

jCodeMunch is an ahead-of-time structural indexer. It walks a repository once, extracts symbols, writes metadata and cached source, and then serves subsequent lookups from that index.

```
Source code (GitHub API or local folder)
|
▼
Security filters (path traversal, symlinks, secrets, binary, size)
|
▼
tree-sitter parsing (language-specific grammars via LanguageSpec)
|
▼
Symbol extraction (functions, classes, methods, constants, types)
|
▼
Post-processing (overload disambiguation, content hashing)
|
▼
Summarization (docstring → AI batch → signature fallback)
|
▼
Storage (JSON index + raw files, atomic writes)
|
```

That pipeline is what makes the server cheap at query time. Parsing and extraction are paid once during indexing. Search and retrieval operate on stored structure instead of reparsing source on every question.

3.2 Symbol Extraction: The `LanguageSpec` Pattern

The parser follows a language-registry pattern centered on `LanguageSpec`. Each supported language declares how symbol-bearing AST nodes should be interpreted by the generic extractor.

```
@dataclass
class LanguageSpec:
    ts_language: str
    symbol_node_types: dict[str, str]
    name_fields: dict[str, str]
    param_fields: dict[str, str]
    return_type_fields: dict[str, str]
    docstring_strategy: str
    decorator_node_type: str | None
    container_node_types: list[str]
    constant_patterns: list[str]
    type_patterns: list[str]
```

The generic extractor handles the common case for the supported registry languages. The architecture and parser modules also make room for custom walkers where a grammar's naming model does not fit the generic pattern cleanly. That is the right trade: common cases stay declarative; difficult grammars get bespoke AST walking rather than being forced through a bad abstraction.

Two post-processing passes complete extraction:

1. **Overload disambiguation**

Duplicate symbol IDs receive numeric suffixes such as `~1` and `~2`.

2. **Content hashing**

SHA-256 hashes are computed from symbol source content for change detection and retrieval verification.

3.3 Symbol ID Scheme

Stable symbol IDs use this format:

```
{file_path}::{qualified_name}#{kind}
```

Examples:

```
src/main.py::UserService.login#method  
src/utils.py::authenticate#function  
config.py::MAX_RETRIES#constant
```

These IDs remain stable across re-indexing as long as file path, qualified name, and symbol kind do not change. That stability matters because agent workflows are often multi-step: search first, retrieve second, compare later. If symbol IDs drift on every refresh, the workflow falls apart.

3.4 Storage and Retrieval

The storage layer uses a two-level design:

- **JSON index** for symbol metadata, file hashes, language counts, summaries, and byte ranges
- **Raw file cache** for full source text, enabling direct byte-range retrieval without reparsing

Each indexed symbol stores `byte_offset` and `byte_length`. Retrieval tools use those coordinates to read only the requested span from cached source. The `get_symbol` tool returns one indexed symbol by stable ID; `get_symbols` batches the same operation; `get_file_content` exposes line-range retrieval from cached files when the task requires a wider local window rather than a single symbol.

Indexes are stored under `~/.code-index/` by default and can be relocated via `CODE_INDEX_PATH`. Incremental re-indexing compares stored file hashes with current hashes and reprocesses only changed files. Writes are atomic: metadata is written to a temporary file and then renamed into place, preventing partially written indexes from being observed by clients.

3.5 Search Algorithm

`search_symbols` uses weighted scoring after applying optional prefilters such as `kind`, `language`, and `file_pattern`.

Match Type	Weight
Exact name match	+20
Name substring	+10
Name word overlap	+5 per word
Signature match	+8 full / +2 per word
Summary match	+5 full / +1 per word
Docstring / keyword match	+3 full / +1 per word

Exact identifier equality carries the highest weight because symbol search is frequently literal or near-literal. Signature text is next because developers often search by API shape when the exact name is unknown. Summaries and docstrings add semantic glue without outranking direct structural hits. Zero-score results are excluded entirely.

3.6 Summarization Tiers

Summarization follows a three-tier fallback path:

1. **Docstring-derived summary**
2. **Optional AI batch summary**
3. **Signature fallback**

AI summarization is optional. When API keys or a local OpenAI-compatible endpoint are configured, batch summarization can use Anthropic, Gemini, or local models. Without any model configuration, the server still performs symbol extraction, indexing, search, and retrieval normally.

4. Benchmark: Three Real-World Repositories

The repository includes benchmark documents for three open-source frameworks spanning JavaScript, Python, and Go: Express, FastAPI, and Gin. The benchmark figures below are taken from those project benchmark documents. Where the benchmark reports token savings directly, those numbers are used as-is. Where a benchmark demonstrates precision retrieval qualitatively but does not publish exact byte offsets, the paper does not invent them.

4.1 Repo 1 — expressjs/express

Corpus: 34 files, 117 symbols indexed, index time under 2,000 ms

Queries (issued in parallel):

Query	Latency	Tokens Saved
middleware routing request handler	7 ms	21,843
error handling next function stack	5 ms	18,539
response send json headers content type	3 ms	12,445

Deep-dive: one precision retrieval

The benchmark retrieves `tryRender` from `lib/application.js` in 35 ms. The returned span is only the target helper, seven lines long, rather than the surrounding application module. The benchmark's file-size comparison places `lib/application.js` at approximately 25,000 characters, making this a representative large-file needle extraction rather than a trivial whole-file echo.

What this demonstrates: In a compact but symbol-dense JavaScript codebase, jCodeMunch turns broad natural-language queries into ranked symbol hits quickly, then resolves the exact target helper without paying to load the surrounding module. The retrieval win is structural, not semantic: the tool narrows the read set before the model starts reasoning.

4.2 Repo 2 — fastapi/fastapi

Corpus: 156 files, 1,359 symbols indexed, index time under 5,000 ms

Queries (issued in parallel):

Query	Latency	Tokens Saved
request validation dependency injection	22 ms	3,676
middleware exception handler error response	20 ms	48,491
WebSocket connection streaming	18 ms	97,223
OAuth2 security authentication bearer token	20 ms	40,175

Deep-dive: one precision retrieval

The benchmark retrieves `request_validation_exception_handler` from `fastapi/exception_handlers.py` in 49 ms. The returned span is the exact asynchronous handler implementation at lines 20 to 26, rather than the surrounding file. This is a clean example of targeted retrieval doing exactly what an agent needs: return the implementation, not the room it happens to be standing in.

What this demonstrates: In a mature Python framework, the main value is not just large-file reduction. It is high-confidence structural discovery across a wide symbol graph. The WebSocket and OAuth2 benchmark queries show that the search layer can surface relevant symbols across multiple modules without forcing file-at-a-time exploration.

4.3 Repo 3 — gin-gonic/gin

Corpus: 40 files, 805 symbols indexed, index time under 3,000 ms

Queries (issued in parallel):

Query	Latency	Tokens Saved
router middleware handler context	24 ms	40,374
binding validation struct JSON request	19 ms	11,854
response JSON render template write	69 ms	28,329

Deep-dive: one precision retrieval

The benchmark retrieves `Context` from `context.go` in 17 ms, returning the target type definition with inline comments intact. The benchmark characterizes `context.go` as a large file with more than 3,000 lines, making this a strong example of type-precise extraction from a heavy source file.

What this demonstrates: In Go codebases with large, central types, the gain comes from retrieving a definition and its local explanatory comments without hauling in thousands of adjacent lines. The 69 ms response-rendering query is an outlier within the published benchmark set and likely reflects a broader or noisier match set rather than the cost of a single-symbol retrieval path.

4.4 Benchmark Highlights

Repo	Corpus	Representative Win
expressjs/express	34 files, 117 symbols	3 to 7 ms query latency on symbol search; targeted helper retrieval from a large application module
fastapi/fastapi	156 files, 1,359 symbols	up to 97,223 tokens saved on a single benchmark query; exact exception-handler retrieval in 49 ms
gin-gonic/gin	40 files, 805 symbols	central type retrieval from a 3,000+ line file in 17 ms; 40,374 tokens saved on router/context search

This benchmark set demonstrates three distinct strengths: fast ranked lookup in compact JavaScript projects, cross-module discovery in Python frameworks, and precise extraction from large Go source files. The retrieval path remains useful even when the benchmark does not publish exact byte offsets, because the measurable win is still the same thing: dramatically smaller read sets.

5. Tool Reference

Tool	Purpose
<code>index_repo</code>	Index a GitHub repository into the local symbol store
<code>index_folder</code>	Index a local directory with repository-style filtering and caching
<code>list_repos</code>	List indexed repositories available to the server
<code>get_file_tree</code>	Return the indexed file tree for a repository
<code>get_repo_outline</code>	Return a repository-level structural overview
<code>get_file_outline</code>	Return the symbol outline for a single file
<code>search_symbols</code>	Search indexed symbols by name, signature, summary, or related text
<code>get_symbol</code>	Retrieve one symbol by stable ID
<code>get_symbols</code>	Retrieve multiple symbols in one request
<code>search_text</code>	Perform text search over cached repository contents
<code>get_file_content</code>	Return raw cached file content, optionally by line range

Tool	Purpose
<code>invalidate_cache</code>	Remove cached index and raw content for a repository

Representative two-step lookup workflow

```
{
  "tool": "search_symbols",
  "arguments": {
    "repo": "fastapi/fastapi",
    "query": "request validation exception handler",
    "language": "python",
    "max_results": 3
  }
}

{
  "results": [
    {
      "symbol_id": "fastapi/
        exception_handlers.py::request_validation_exception_handler#function",
      "name": "request_validation_exception_handler",
      "kind": "function",
      "file": "fastapi/exception_handlers.py",
      "line": 20,
      "summary": "HTTP 422 handler for request validation failures."
    }
  ],
  "_meta": {
    "powered_by": "jcodemunch-mcp by jgravelle · https://github.com/
      jgravelle/jcodemunch-mcp",
    "timing_ms": 20.0,
    "tokens_saved": 48491,
    "total_tokens_saved": 656482,
    "cost_avoided": {
      "claude_opus": 0.7274
    },
    "total_cost_avoided": {
      "claude_opus": 9.85
    }
  }
}

{
  "tool": "get_symbol",
  "arguments": {
    "repo": "fastapi/fastapi",
    "symbol_id": "fastapi/
      exception_handlers.py::request_validation_exception_handler#function",
```

```

        "verify": true,
        "context_lines": 0
    }
}

{
  "symbol": {
    "id": "fastapi/
exception_handlers.py::request_validation_exception_handler#function",
    "kind": "function",
    "file": "fastapi/exception_handlers.py",
    "line": 20,
    "end_line": 26,
    "source": "async def request_validation_exception_handler(\n
        request: Request, exc: RequestValidationError\n) -> JSONResponse:\n
        \n        return JSONResponse(\n            status_code=422,\n            content={\"detail\": jsonable_encoder(exc.errors())},\n        )"
  },
  "_meta": {
    "powered_by": "jcodemunch-mcp by jgravelle · https://github.com/
jgravelle/jcodemunch-mcp",
    "timing_ms": 49.0,
    "content_verified": true,
    "tokens_saved": 150,
    "total_tokens_saved": 656632,
    "cost_avoided": {
      "claude_opus": 0.0023
    },
    "total_cost_avoided": {
      "claude_opus": 9.8523
    }
  }
}

```

6. Security Model

The indexing pipeline applies five defensive layers before content is cached or exposed:

1. Path traversal prevention

Candidate paths are resolved and checked against the permitted repository root.

2. Symlink escape protection

Symlinks are skipped by default, and enabled links are validated to ensure they do not escape the allowed root.

3. Secret-file exclusion

Known credential-bearing filenames and patterns are excluded before parsing and caching.

4. Binary file detection

Binary content is screened out using extension filtering and content inspection.

5. Configurable file size limits

Oversized files are excluded from indexing to bound parser cost and cache growth.

The packaging pipeline also includes a structural guardrail to prevent credential bundling into distribution artifacts.

7. MCP Client Compatibility

jCodeMunch implements the MCP stdio protocol and is designed to be client-agnostic. The repository documentation explicitly demonstrates stdio-based installation and configuration rather than a client-specific integration layer. In practice, that makes the server suitable for Claude Code, Claude Desktop, VS Code hosts that support MCP, Cursor, Windsurf, Cline, Google Antigravity, and any other MCP-compatible host.

Copy-paste config snippet

```
{
  "mcpServers": {
    "jcodemunch": {
      "command": "uvx",
      "args": ["jcodemunch-mcp"]
    }
  }
}
```

A single-command Claude Code install is also documented:

```
claude mcp add jcodemunch uvx jcodemunch-mcp
```

8. Token Economy: Scaling Impact

For the representative “find a function” workload documented by the project, the baseline comparison is approximately 40,000 tokens for naive file reading versus about 200 tokens for symbol-precise retrieval.

Daily Queries	Naive (tokens/day)	jCodeMunch (tokens/day)	Saved	Daily Cost Avoided (Opus)
10	400,000	2,000	398,000	\$5.97
100	4,000,000	20,000	3,980,000	\$59.70
1,000	40,000,000	200,000	39,800,000	\$597.00

Savings are tracked in the `_meta` field of tool responses and persisted across sessions in `~/.code-index/_savings.json`. The project also supports an anonymous community savings meter, with reporting disabled by setting `JCODEMUNCH_SHARE_SAVINGS=0`.

9. Limitations and Non-Goals

jCodeMunch is deliberately narrow in scope.

- It does not provide LSP diagnostics, completions, or refactoring services.
- It does not perform real-time file watching.
- It does not attempt semantic program analysis such as type inference or call-graph construction.
- It does not build a global cross-repository index.
- It does not implement editing workflows.

That narrowness is an engineering advantage. It keeps the retrieval path deterministic, inspectable, and cheap, which matters even more in multi-agent systems where debugging incorrect context is harder than debugging incorrect code.

10. Benchmark Scorecard

Metric	
Result	

Supported languages documented	15
MCP tools exposed	12
Index: FastAPI benchmark corpus	✓ 156 files / 1,359 symbols
Express query latency	✓ 3–7 ms
FastAPI query latency	✓ 18–22 ms
Gin query latency	✓ 19–69 ms
Benchmark queries reported	✓ 10 across three corpora
Largest file navigated in benchmark narrative file	✓ 3,000+ line Go file
Peak token reduction	✓ ~99%
Typical token reduction	✓ 80–98%
AI summaries required for operation	✗ optional
Secret files auto-excluded	✓ yes
Incremental re-index	✓ yes
Atomic writes	✓ yes
Client lock-in	✗ none

Methodology note: Benchmark figures are drawn from the repository’s published benchmark documents for Express, FastAPI, and Gin. This scorecard reports published corpus sizes, latencies, and token-savings figures directly. It does not fabricate byte-precision measurements where the benchmark documents do not provide them.

11. Sources

- README.md
- ARCHITECTURE.md
- SECURITY.md
- LANGUAGE_SUPPORT.md
- pyproject.toml

- `src/jcodemunch_mcp/server.py`
- `src/jcodemunch_mcp/parser/languages.py`
- `src/jcodemunch_mcp/parser/extractor.py`
- `src/jcodemunch_mcp/storage/index_store.py`
- `src/jcodemunch_mcp/tools/search_symbols.py`
- `src/jcodemunch_mcp/tools/get_symbol.py`
- `benchmarks/jCodeMunch_Benchmark_Express.md`
- `benchmarks/jCodeMunch_Benchmark_FastAPI.md`
- `benchmarks/jCodeMunch_Benchmark_Gin.md`