

# jDocMunch

## AI-native documentation navigation for developer agents

### Executive summary

Large language models are increasingly limited not by generation quality, but by how inefficiently they acquire knowledge from documentation. In most tooling stacks, the effective unit of information needed by an agent is a section or explanation, while the dominant unit of retrieval remains the file or an arbitrary text chunk. jDocMunch addresses this mismatch by indexing documentation according to its authored structure and returning coherent section-level payloads instead of whole files or fragmented chunks.

Across Kubernetes, LangChain, and SciPy benchmark corpora, the system reduced context requirements by orders of magnitude while preserving hierarchical meaning. In one Kubernetes case, a 95,051-byte reference file contained an 863-byte answer near byte offset 71,763; jDocMunch extracted the exact section instead of loading the full document. In one benchmark configuration against the full LangChain corpus, targeted section retrieval reduced context requirements from approximately 1.8 million tokens for naive full-corpus reading to approximately 2,200 tokens for section-level extraction.

Finding	Value	What it means
Kubernetes precision extraction	95,051 B -> 863 B	File-size and answer-size are often wildly mismatched; exact section extraction is materially more efficient.
LangChain MDX unlock	699 -> 5,973 sections (+754%)	Modern docs are often unreadable to naive parsers until MDX-specific preprocessing is applied.
SciPy corpus scale	430 files, 10,402 sections indexed in ~2.2 s	The approach works across dense legacy reStructuredText, not just modern Markdown sites.

## 1. Problem definition

AI agents are still reading documentation the way hurried humans skim oversized manuals: they open large files, scan linearly for clues, and pay to process large volumes of irrelevant text on the way to one small answer. This creates what we describe as the Documentation Tax: unnecessary token consumption, displaced context-window capacity, and structural meaning loss.

For documentation-heavy workflows, the effective unit of knowledge needed by the model is usually a coherent explanation, subsection, or code-adjacent narrative block. In naive systems, the retrieval unit is instead a file or an arbitrary chunk. jDocMunch is built around the opposite assumption: documentation should be navigated section-by-section according to authored hierarchy, not read file-by-file.

### Four measurable failure classes

1) excess token consumption, 2) context-window displacement, 3) semantic drift caused by irrelevant neighbors, and 4) instability introduced by probabilistic chunk retrieval.

## 2. Why existing approaches break down

### File scanning

Strength. Preserves full local context and requires little preprocessing.

Failure mode. Treats a 95 KB file and an 863-byte answer as roughly the same retrieval unit.

Consequence for AI agents. Large cost overhead, reduced room for reasoning, and latency tied to file size rather than answer complexity.

### Keyword search

Strength. Fast and easy to implement.

Failure mode. Returns lexical matches without preserving the authored explanation around them.

Consequence for AI agents. Agents often receive the right noun without the surrounding instructions, constraints, or caveats.

### Text chunking

Strength. Provides generic retrieval units and works reasonably for undifferentiated prose.

Failure mode. Chunk boundaries are not semantic boundaries. Heading-to-explanation relationships, code blocks, and explanatory prose are frequently severed.

Consequence for AI agents. The model receives fragmented knowledge and must infer missing structure, increasing hallucination risk.

### 3. System architecture

jDocMunch is an MCP-compatible documentation indexing and retrieval layer. Its implementation can be understood as a nine-step processing pipeline:

- Ingest raw documentation files from a local directory or repository checkout.
- Detect format and apply format-specific preprocessing.
- Parse headings, prose blocks, and code blocks into structural sections.
- Assign stable section identifiers and exact byte ranges.
- Persist section text, metadata, hierarchy, and optional summaries into an index.
- Accept an agent query and rank candidate sections using the active search pipeline.
- Return a target section identifier.
- Extract the exact section payload from recorded boundaries.
- Provide the agent with a coherent section-level context block instead of a whole file.

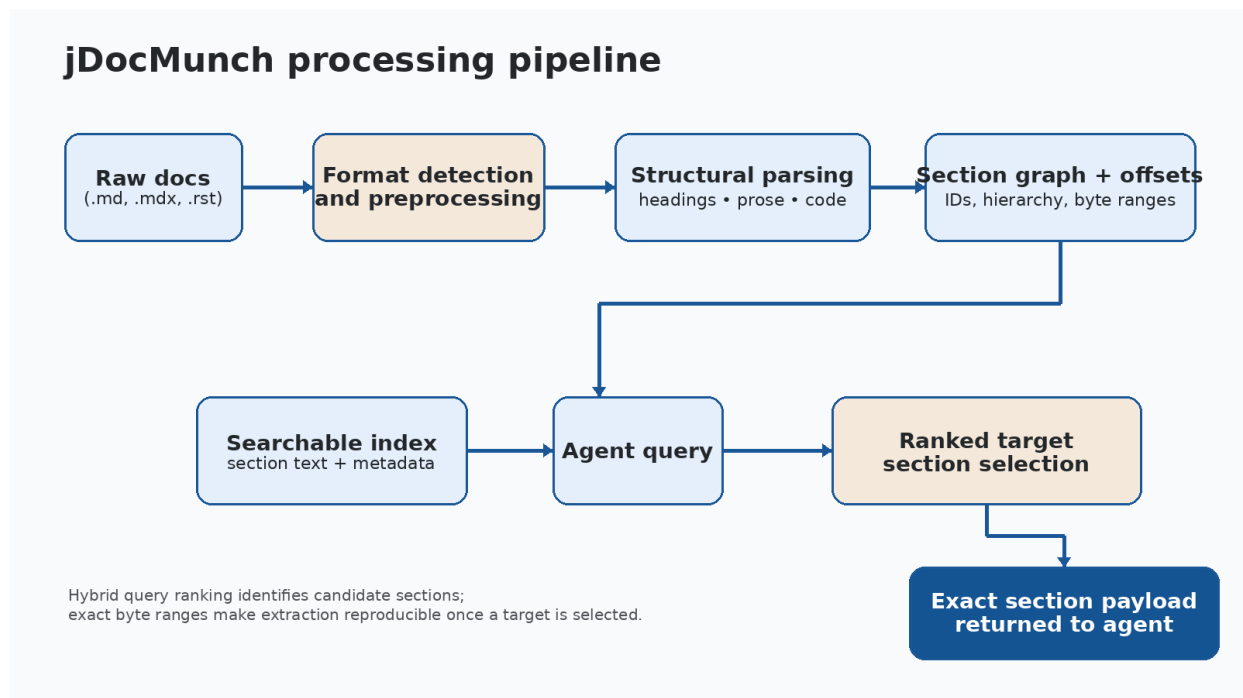


Figure 1. Raw documentation is preprocessed, structurally parsed, indexed, ranked, and then extracted at exact section boundaries.

### 4. Retrieval model and query flow

Target identification and structural extraction should be treated separately. Only the extraction phase is deterministic in the strict engineering sense; query-driven ranking remains configuration-dependent even when it is stable and repeatable within a benchmark setup.

#### 4.1 Target identification

The system ranks candidate sections for a query using the active search stack defined by the benchmark configuration. This phase may include semantic signals, lexical signals, metadata, or precomputed summaries depending on the corpus and tool configuration. Because ranking is query-

driven, this phase should be described as stable or repeatable within a configuration, not absolutely deterministic in the abstract.

4.2 Structural extraction

Once a target section has been identified, the extraction path is exact and reproducible. jDocMunch stores stable section IDs, hierarchy, and byte ranges; the returned payload is therefore a deterministic section boundary with stable structural context. The stronger claim is thus deterministic structural extraction, not universally deterministic semantic selection.

5. Benchmark methodology

The benchmark assumptions are made explicit below so the reported figures can be interrogated as engineering evidence rather than treated as opaque headline claims. The figures summarize what was actually measured and what each baseline means.

Corpus	Files / sections	Query set	Measurement notes
LangChain	500 files 5,973 sections after MDX support	6 targeted queries	Compared .md-only index with MDX-enabled index. Signal scores were assigned by the benchmark author using an internal 0–8 relevance rubric.
Kubernetes	1,569 files in corpus 500 indexed in test batch 4,355 sections	5 targeted queries	Queries issued in a single parallel call. Latency measured end-to-end in wall-clock milliseconds.
SciPy	430 files 10,402 sections	12 targeted queries plus 3 extraction checks	Latencies measured wall-clock. Token-savings figures were estimated from indexed-section and returned-section token counts in the benchmark environment.

Baseline definitions.

Naive corpus read means loading the full documentation corpus into model context before answering. This is intentionally a weak baseline, but it illustrates the upper bound of waste in file-oriented workflows. Likely-file skim means reading one or more files judged likely relevant in full. Where no chunked-RAG implementation was directly benchmarked, the absence of that comparison is stated explicitly rather than implied away.

### Scoring and measurement caveats.

The LangChain retrieval quality figures (1.8/8 to 6.8/8) were generated by the benchmark author using an internal relevance rubric rather than a multi-rater blinded study. They are useful directional evidence, not a claim of externally validated statistical significance. Token figures throughout the paper are benchmark-environment estimates, not universal production billing guarantees.

## 6. Benchmark results

Corpus	Key result	Latency	Context delta	Most honest takeaway
LangChain	699 -> 5,973 sections	~107 ms avg	~1.8M tokens for naive full-corpus reading -> ~2.2K in one LangChain benchmark configuration	MDX preprocessing made the real corpus visible to the parser in a benchmark against the full LangChain corpus.
Kubernetes	95,051 B file -> 863 B extracted	83–100 ms search; 754 ms batch extraction	~110x reduction for the 95 KB needle case	Exact section extraction beats file-scale reading when answer size is tiny.
SciPy	430 files, 10,402 sections in ~2.2 s	149 ms avg search	~843,000 tokens avoided across one benchmark session	The approach works on dense legacy reStructuredText, not only on modern docs.

## 7. Case studies

### 7.1 Kubernetes: the 95 KB needle problem

The Kubernetes query `kubectl authentication plugins` pointed into authentication.md, a 95,051-byte reference file. The relevant section began near byte offset 71,763 and the extracted answer payload was 863 bytes.

## 7.2 LangChain: MDX unlock

The LangChain benchmark demonstrated that parser visibility can be the bottleneck, not ranking quality. Queries such as ``tool calling openai functions example`` and ``langgraph state machine workflow`` depended on MDX-authored sections that were largely invisible before preprocessing. Prior to MDX support, the system indexed only 699 sections and missed 490 of 500 indexable files. Adding `strip_mdx()` and registering `.mdx` as a parseable extension expanded coverage to 5,973 sections, a 754% increase, and materially improved internal signal scores. This is a precise implementation story: a small preprocessing layer created a measurable knowledge-coverage gain.

## 7.3 SciPy: dense legacy technical docs

SciPy stress-tested the system against `reStructuredText`, mathematically dense content, and a corpus accumulated over many years. Queries such as ``interpolation curve fitting spline`` and ``array API compatibility backend agnostic`` were used to probe whether section-level navigation held up under highly technical material.

jDocMunch indexed 430 files and 10,402 sections in approximately 2.2 seconds, then completed 15 retrieval operations while avoiding roughly 843,000 tokens relative to file-oriented reading approaches. The significance is not only efficiency but breadth: the technique is not limited to modern Markdown documentation sites.

## 8. Limitations and boundary conditions

The system has clear strengths, but its behavior still depends on document structure, query quality, and corpus characteristics. The constraints below define where results may degrade or require additional engineering work.

- Poorly structured documentation. When headings are inconsistent, missing, or mechanically generated, section quality can degrade because structure carries less meaning.
- Flat reference pages. A single-page reference with minimal hierarchy may provide fewer strong structural cues than a narrative guide with clear heading ladders.
- Custom MDX or embedded components. Some sites rely on custom components or unusual JSX patterns that may require additional preprocessing beyond the current `strip_mdx()` rules.
- Query ambiguity. Underspecified prompts can still lead to imperfect candidate selection because target identification remains query-driven even when extraction is exact once a target is chosen.
- Indexing and storage tradeoffs. Larger corpora improve navigability but introduce storage, rebuild, and index-management costs that should be characterized in production environments.
- Evaluation limitations. The published benchmarks are internal and highly informative, but they are not yet a blinded multi-rater evaluation or an externally replicated benchmark suite.

## 9. Relationship to jCodeMunch

jDocMunch and jCodeMunch solve adjacent parts of the same agent-navigation problem.

jCodeMunch retrieves implementation artifacts: functions, classes, methods, and symbols.

jDocMunch retrieves explanatory artifacts: design rationale, architecture notes, tutorials, and configuration guidance. A concise framing is that jCodeMunch retrieves the how while jDocMunch retrieves the why.

This pairing suggests a practical future direction: code/documentation consistency checking. If code retrieval clearly indicates one implementation path while the retrieved documentation mandates another, the combined system can help flag documentation drift rather than forcing the agent to guess how implementation and explanation align.

For engineering leadership, that possibility matters because it turns the pair from a retrieval convenience into potential control-plane infrastructure. A system that can compare what the code does with what the documentation claims could support drift detection, audit workflows, and faster remediation when implementation and guidance diverge.

## 10. Conclusion

The evidence supports a focused thesis: documentation access for AI systems should be designed as navigation over structured knowledge rather than brute-force reading over files or arbitrary chunks. jDocMunch demonstrates that section-level structure, stable boundaries, exact extraction,

and format-aware preprocessing together yield materially better context efficiency for agent workflows.

The central shift is simple: the model should spend its reasoning budget on solving the problem, not on hunting for the paragraph. That is the practical contribution of jDocMunch and the basis for treating it as AI-native documentation navigation rather than generic document search.

Appendix A. Evaluation rubric

The LangChain benchmark used an internal 0–8 relevance rubric. The scale below is published so the figures can be interpreted as engineering evidence instead of opaque marketing numbers.

Score	Interpretation
0	Irrelevant or unusable.
2	Tangentially related but not answer-bearing.
4	Partially useful; requires substantial additional searching.
6	Mostly correct and actionable, but missing some context or precision.
8	Direct, high-signal, and sufficient as an answer source.

Appendix B. Representative query sets

LangChain queries:

- agent tool routing architecture
- rag retriever vectorstore differences
- langgraph state machine workflow
- memory buffer vs summary memory
- tool calling openai functions example
- prompt templates best practices

Kubernetes queries:

- pod scheduling affinity rules
- kubernetes persistent volume reclaim policy
- kubectl authentication plugins
- network policy ingress vs egress
- container runtime interface cri



### SciPy query examples:

- sparse matrix linear algebra solvers
- optimization gradient descent minimize
- interpolation curve fitting spline
- array API compatibility backend agnostic
- BLAS LAPACK compiler build system meson

## Appendix C. Recommended external validation plan

A stronger future edition should include an externally reproducible benchmark suite. The items below define a practical validation roadmap rather than a disclaimer: they show how the benchmark program can be expanded so outside evaluators can reproduce results, challenge assumptions, and compare jDocMunch against stronger baselines.

- Publish corpus snapshots or exact commit references for every benchmark corpus.
- Release the full query sets and lock them before testing.
- Run at least one explicit chunked-RAG baseline with documented chunk size, overlap, and reranker settings.
- Use multiple evaluators or blinded scoring for qualitative relevance judgments.
- Report cold-cache and warm-cache latency separately.
- Separate token estimates from billed model costs and identify the tokenizer used.