Chunking Technique Research

Jarrod Connolly

June 9, 2026

Abstract

After building RAG systems for technical books, I kept asking which chunking strategy actually retrieves better. I built a research platform to answer that with real vector search, LLM-as-judge ground truth, and metrics such as NDCG, MAP, and MRR. I compare fixed-length and overlap splitters with heading-aware and linguistically scored approaches. Some of this is still a heuristic, not a settled result.

Introduction

What makes a good chunking technique, and how would you even know? Ask ten RAG tutorials and you get ten confident answers about fixed length, overlap, splitting on headings, and respecting token boundaries. The advice sounds settled. Nobody spends much time explaining why one approach should beat another, or how you would prove it if you tried.

HAL was one of my first real projects working with LLMs, a straightforward RAG chatbot over technical PDFs with a fun retro terminal UI. It worked well enough that I kept using it, until I started wondering what was actually ending up in my vector store. So I looked.

It was a mess. Chunks full of pipe-delimited table text that read like spreadsheet export instead of explanation. Table of contents lines that named a chapter but contained none of it. Index entries with page numbers and nothing else. Thousands of embeddings sitting in Qdrant, and a depressing share of them would never help answer a real question. They just ate storage and crowded the index.

The worse case was when those chunks came back in a query. You ask about register allocation and the retriever hands the LLM a chunk that is literally `Chapter 7 ..... 142`, or a three-column table of opcode names with no surrounding prose. The model tries to make sense of junk because that is what the pipeline gave it. Bad chunking does not just miss the right answer. It actively confuses the generation step with noise dressed up as context.

That is when opinions stopped being enough. I wanted numbers, not another blog post comparing two libraries. I wanted a controlled experiment where the same document, the same queries, and the same embedding model run through multiple chunking strategies, then get scored with metrics the information retrieval community actually trusts.

That question turned into a research platform. You upload a PDF, define evaluation queries, pick the chunking strategies you want to compare, and let the system run the full pipeline. It uses real embeddings

in Qdrant, an LLM that scores chunk relevance on a 0-3 scale, and standard metrics like NDCG, MAP, and MRR through pytrec_eval / pytrec_eval_terrier. TREC-format files land on disk next to the chunk text.

The project is not finished. I am still hardening the pipeline, still writing the paper draft, and still chasing the kind of results I would be willing to put on arXiv. But the build itself has been the point. This is the story of why I started, what I built, and what I am learning about the gap between “it works in a demo” and “I would stake a claim on it.”

The Question That Would Not Go Away

RAG tutorials make chunking sound solved. Pick 512 tokens, add 50 tokens of overlap, and call it a day. They talk about size. They rarely talk about what kind of text deserves to be a chunk in the first place.

I built HAL as one of my first real projects working with LLMs. It was a fairly straightforward RAG chatbot over technical PDFs, wrapped in a retro terminal UI that made the whole thing fun to poke at. The pipeline was the usual story: ingest documents, chunk them, embed them, retrieve on query, and generate an answer. That is where I hit the debris problem. I was using HAL daily, loading compiler texts and systems books, asking questions and getting answers that sometimes made no sense until I looked at what had actually been chunked. Chunking is the invisible layer under retrieval quality. Change the splitter and the same question gets different answers, but almost nobody measures the difference systematically.

So what would a real comparison look like?

I sketched requirements on a napkin that turned into a README:

1. Same source document for every method.
2. The same set of evaluation queries per document.
3. Real vector retrieval, not a simulation.
4. Ground truth relevance labels at scale.
5. Standard IR metrics, not hand-waved “seems better.”
6. Artifacts on disk that another researcher could inspect.

That list became the Chunking Research Framework. A web-first platform where the UI is not a convenience layer. It is how you actually run experiments.



Figure 1: Chunk flow

From HAL’s Chunker to a Research Hypothesis

HAL already had the seed of my custom approach. When I ingested PDFs for the chatbot, I did not blindly slice text. I converted to Markdown with pymupdf4llm, used Mistune to strip tables and code blocks, split on section headings, and scored each candidate chunk with spaCy. Sentence coherence, noun and verb density, length penalties for fragments. Keep the prose, drop the noise.

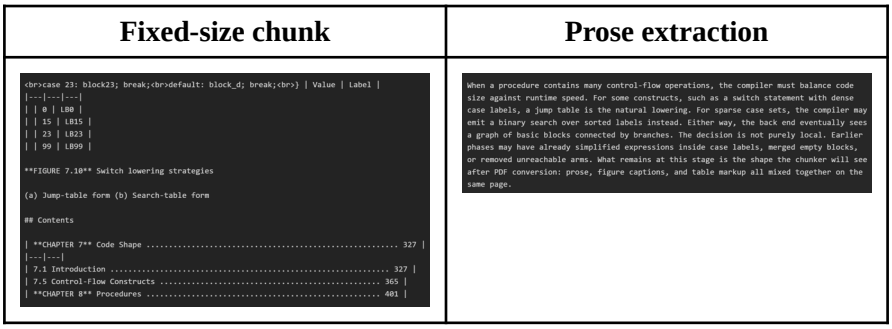
That scorer lived in a product context. “Good enough for HAL” is not the same as “measurably better for retrieval.” I wanted to test the hypothesis directly. Does linguistically-aware chunking outperform naive fixed-length splitting on technical documents, measured with proper retrieval metrics?

The NLP chunker in the research platform is an evolution of that HAL pipeline. Same instincts, but now it runs inside a controlled experiment where several baselines sit beside it.

The idea is simple. Tables and code blocks are often useless retrieval targets for prose questions. Strip them before chunking, and you stop polluting the vector index with structural debris.

That is a pragmatic cut, not the only answer. Some tables carry real information. A comparison of optimizer passes, a matrix of hyperparameters, a register layout. Right now I throw that away because raw table markup retrieves badly and confuses the LLM when it does surface. A future version of the pipeline might run those sections through a small local model first, ask it to reinterpret the table or code block as plain prose, and chunk the result like any other paragraph. Same embedding path, but the content would actually answer questions instead of looking like spreadsheet export. I have not built that yet. It is on the list of things worth trying once the evaluation framework can tell me whether it helps.

The scorer goes further. `LinguisticChunkScorer` evaluates each chunk on sentence structure, content density, and semantic richness, then filters by threshold or top-N selection. It is a heuristic. I know that. But now I have machinery to ask whether the heuristic pays off.



Fixed-size chunk	Prose extraction
 case 23: block23; break; default: block_d; break; } | Value | Label | |---|---| | | 0 | L80 | | | 15 | L815 | | | 23 | L823 | | | 99 | L899 | **FIGURE 7.10** Switch lowering strategies (a) Jump-table form (b) Search-table form ## Contents | **CHAPTER 7** Code Shape ..... 327 | |---|---| | 7.1 Introduction ..... 327 | | 7.5 Control-Flow Constructs ..... 365 | | **CHAPTER 8** Procedures ..... 401 |	When a procedure contains many control-flow operations, the compiler must balance code size against runtime speed. For some constructs, such as a switch statement with dense case labels, a jump table is the natural lowering. For sparse case sets, the compiler may emit a binary search over sorted labels instead. Either way, the back end eventually sees a graph of basic blocks connected by branches. The decision is not purely local. Earlier phases may have already simplified expressions inside case labels, merged empty blocks, or removed unreachable arms. What remains at this stage is the shape the chunker will see after PDF conversion: prose, figure captions, and table markup all mixed together on the same page.

Comparing Strategies, One Document, One Experiment

Every chunking strategy runs through the same dispatcher. Take a PDF, write artifacts to an experiment directory, optionally embed into Qdrant. The platform registers six methods today, from naive fixed-length baselines through to my NLP approach. That number is a snapshot, not the ceiling. I built the framework to grow as I learn more about what actually matters for retrieval.

The baselines are deliberately boring. That is the point.

- **Length**. Fixed character windows, the naive baseline everyone secretly uses.
- **Overlap**. Sliding windows with configurable overlap.
- **Recursive**. Structure-aware splitting when the document has hierarchy.
- **Token**. Chunk by tokenizer boundaries, closer to how LLM context windows think.
- **Markdown**. Split on headings in the preprocessed Markdown.
- **NLP**. My approach. Prose extraction, section splitting, linguistic scoring, threshold or top-N filter. This is the one I care most about improving.

I am also tracking chunking work that has come out since I started building. Chroma's chunking evaluation compares semantic splitters that break text at embedding-similarity boundaries against LLM-guided chunkers that let the model choose where to cut. Recent papers push on late chunking and contextual retrieval, embedding a full document before splitting or prepending surrounding context to each chunk so isolated fragments carry more meaning. NVIDIA's benchmarks found page-level splitting often beats arbitrary token

windows on structured documents. I want to add strategies like these to the platform, and deepen my own NLP scorer with semantic coherence checks rather than spaCy features alone. The evaluation pipeline stays fixed. Register a new method, run the experiment, read the NDCG.

An experiment is a container. Pick a processed document, select methods, tune parameters, run them one at a time. The UI shows live phase and progress because a full method run can take a long time. Chunking, then ground truth generation, then evaluation.

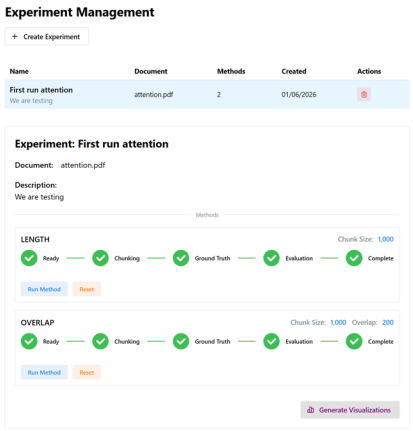


Figure 2: Chunk experiments

The Three-Phase Pipeline

Each method run walks through three phases. This is the core loop of the whole framework.

Phase 1: Chunking

The PDF was already converted to Markdown during upload preprocessing. The chosen chunker splits that text, writes chunk files, and embeds vectors into a per-experiment Qdrant collection. Isolation matters. Early versions shared a single collection and results could contaminate each other. That was one of the first scientific validity issues I had to fix.

Phase 2: Ground Truth (LLM-as-Judge)

For every query and every chunk, the system asks a local LLM a structured question. On a scale of 0-3, how relevant is this chunk to this query? Include a short explanation and a confidence score.

- **0**. Irrelevant
- **1**. Somewhat relevant
- **2**. Highly relevant
- **3**. Perfectly relevant

Manual relevance judging does not scale. I have done it in smaller contexts. It is painstaking. Using a capable local model through an OpenAI-compatible API makes large-scale evaluation practical. The judgments are stored on disk with provenance metadata. Model name, prompt version, embedding model, candidate selection mode.

The LLM is not doing the chunking. It is acting as an automated judge so I can score retrieval quality consistently across methods. That is a deliberate design choice, and it comes with known risks (model bias, prompt sensitivity) that I document rather than hide.

Phase 3: Evaluation

For each query, embed the query text, search Qdrant for the top 100 similar chunks, compare against ground truth, compute metrics. NDCG@100 is the primary number I watch. It rewards methods that rank highly relevant chunks near the top, not just somewhere in the pile.

The pipeline also exports TREC-format files and generates comparison plots. I wanted interoperability with the IR research community, not a bespoke scoring system that dies with the repo.

Building the Web UI (Because CLI Experiments Broke My Brain)

I started with scripts. That worked until it did not. Chunking research has too many steps, too many parameters, and too much waiting. Upload a document, wait for preprocessing, define queries, configure an experiment, run method one, wait, run method two, wait, try to remember which directory had the results.

The web interface became the primary way to work. React 19, Mantine 8, TanStack Query for polling. FastAPI backend with background tasks and a global lock so only one expensive job runs at a time. Upload and preprocess PDFs. Manage queries per document. Create experiments. Watch phase transitions in real time. Inspect results when a method completes.

Dashboard

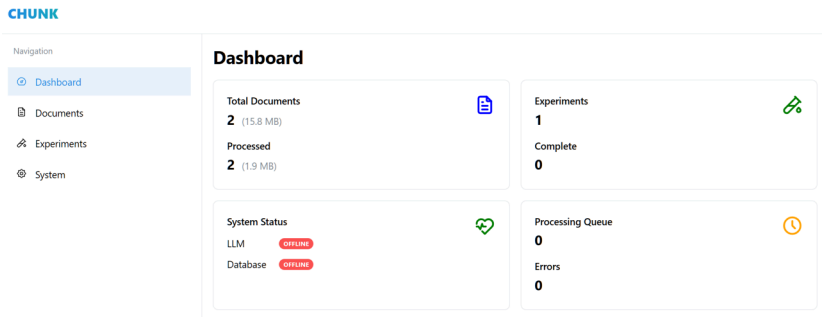


Figure 3: Chunk dashboard

Document Management

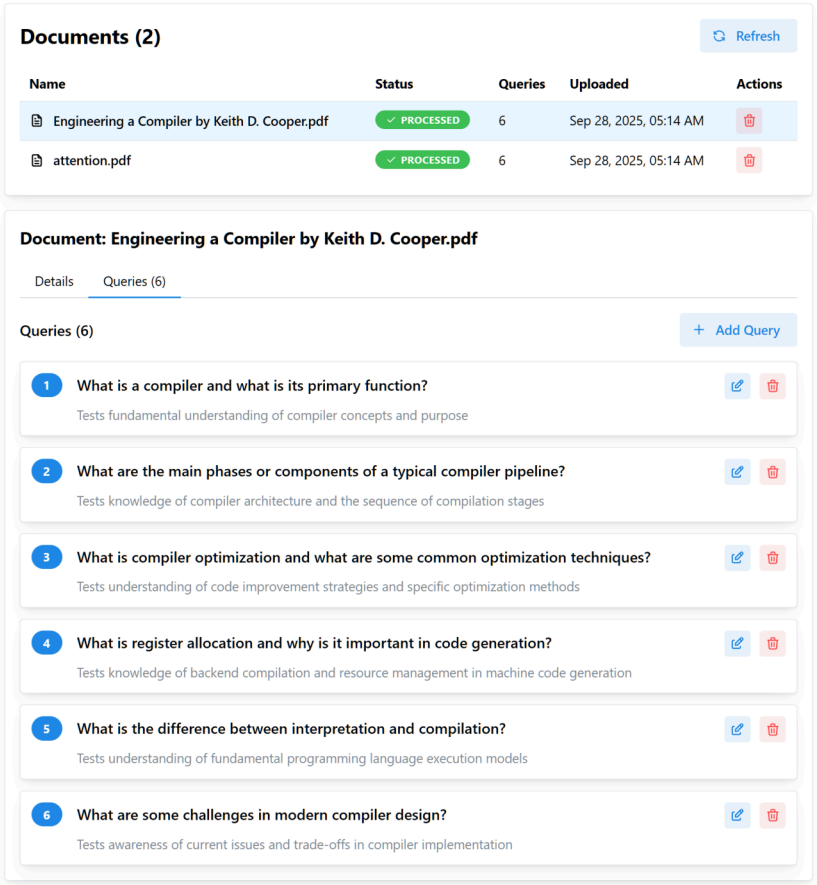


Figure 4: Chunk documents

The global lock is a simple database row. It is not elegant. A crash can leave a stale lock behind and you clear it from the System page. For a single-researcher workstation tool, that is an accepted trade-off for now. I would not ship this to a team without hardening it first.

Getting the Science Right (The Part That Kept Growing)

The platform worked end-to-end early. Upload, chunk, judge, score, visualize. I could run experiments and get numbers. Then I came back after a break, read the code with fresh eyes, and realized I could not trust comparative claims yet.

A full project review surfaced real threats to validity. Shared Qdrant collections with inconsistent clearing. Hardcoded embedding models with no provenance in artifacts. TREC ID mismatches that could break `pytrec_eval`. A brittle LLM judge that could fail silently. Selection bias between ground truth candidate pools and final retrieval.

I did not shelve the project. I wrote it down. Detailed experimentation notes, a prioritized fix list, and a plan for what “credible enough to publish” would actually require. Then I started working through that list, one threat to validity at a time, iterating toward science I could stand behind.

Per-experiment Qdrant collections. Configurable ground truth candidate modes (top-N, random, all). Embedding model and device in config with provenance recorded in every artifact. Centralized TREC ID

helpers. LLM judge retries, prompt versioning, structured output support. Parameter units documented in the UI so “chunk size 1000” means characters for length-based methods and tokens for the token method.

The P0 phase is complete as of May 2026. I can run an experiment now and believe the comparative numbers are not obviously wrong. That is a low bar. It is the bar I needed to clear before running the definitive experiment set.

The Paper Draft and the arXiv Goal

Somewhere along the way this stopped being “a side project with nice plots” and started being “my first research paper.”

I already have a LaTeX skeleton for the paper, with the usual sections in place from introduction through conclusion. Most of the structure is there. The introduction is still a stub. The engineering ran ahead of the writing again, which feels familiar. Complect was a conference talk long before it became a mature compiler. Same muscle, different problem.

The personal goal is straightforward. I want to be a published author, even if the first publication is a preprint on arXiv. Not for the line on a resume. Because I care about doing the thing properly. Notice where opinions outrun evidence. Teach myself what I need to know. Build the tooling to answer the question. Write it up so someone else can reproduce it.

Open questions I am still working through:

- What is the precise research question for paper one?
- How many documents and queries make a credible first study?
- How do I characterize the LLM-as-judge limitation without overselling the results?
- Workshop or pure preprint for the first submission?

I do not have final answers yet. I have a roadmap and a platform that is finally trustworthy enough to start generating those answers.

What I Have Learned So Far

Building this platform taught me things no tutorial would have.

Retrieval quality is an experimental question. You can tune chunk size by gut feel forever. Or you can define queries, run controlled comparisons, and look at NDCG. The second path is slower. It is also the only path that produces evidence.

Research prototypes rot differently than product code. A demo can look fine while silently producing wrong numbers. Shared vector collections, ID format drift, missing provenance. These do not crash the UI. They invalidate your conclusions. I had to learn to audit for scientific validity, not just “does the button work.”

The web UI was the right call. I resisted it at first. Scripts felt more “researcher authentic.” Then I spent an afternoon clicking through experiment state in a browser and never went back. Progress bars matter when ground truth generation means thousands of LLM calls.

My HAL chunker instincts might be right. Now I can test them. The whole point of the NLP method is to prefer coherent prose over fragments. I believe in the approach. Belief is not a result. The framework exists so I can find out.

Conclusion

I started this because RAG chunking advice felt anecdotal and I wanted evidence. I built a web-first research platform with multiple chunking strategies, real Qdrant retrieval, LLM-as-judge ground truth, and standard IR metrics. I hardened it through a painful but necessary validity review. I started a paper draft and mapped the path to an arXiv preprint.

The project is not complete. I still need to run the definitive experiment set, finish the analysis, and write the paper prose. But the shape is there. Upload a document, ask questions of it, run it through the strategies you want to compare, and measure which chopping approach helps a retrieval system find the right pieces.

If you have built RAG and wondered whether your chunking choices actually matter, they probably do. Measuring how much is the harder part. That is what I am building toward.