# 3DCF/doc2dataset: A Token-Efficient, Numerically-Robust Document Layer for RAG and Fine-Tuning Pipelines

Yevhenii Molchanov
3DCF-Labs
github.com/3DCF-Labs/doc2dataset

December 2025

**Abstract**

Large Language Models (LLMs) are increasingly adapted to private corpora via retrieval-augmented generation (RAG) and fine-tuning. Both approaches require transforming heterogeneous documents—PDF reports, internal policies, technical documentation—into structured, token-efficient, and numerically faithful datasets. This "doc-to-dataset" transformation is typically implemented as ad-hoc scripts with limited reproducibility, weak observability, and no common standard.

We present **3DCF/doc2dataset**, an open document layer and pipeline that standardizes this process through three key contributions: (1) a normalized document representation schema with macro-cells supporting layout preservation and importance scoring, (2) **NumGuard**, a novel numeric integrity mechanism using SHA-1 hashes to detect numeric drift across pipeline stages, and (3) a configurable pipeline supporting **30+ file formats** (PDF, HTML, JSON, CSV, LaTeX, images with OCR, and more), producing task-specific samples (QA, summarization, RAG triples) with multi-framework exports to HuggingFace, LLaMA-Factory, Axolotl, and OpenAI fine-tuning formats.

Our evaluation on diverse corpora demonstrates that 3DCF macro-cell contexts require **5–6$\times$ fewer tokens** than naive baselines for QA tasks while achieving higher accuracy. NumGuard achieves perfect recall (1.0) on all A-bucket numeric corruptions across 18,501 guards extracted from financial reports. The Rust-based implementation processes documents at scale with comprehensive metrics and is released under Apache-2.0 license.

**Keywords:** Document Processing, Retrieval-Augmented Generation, Large Language Models, Fine-Tuning, Numeric Integrity, Token Compression, Dataset Generation

## 1 Introduction

Organizations increasingly seek to leverage Large Language Models (LLMs) that understand *their* documents: financial reports, risk disclosures, regulatory filings, contracts, API references, and internal wikis. Two dominant strategies have emerged for domain adaptation:

1. **Retrieval-Augmented Generation (RAG)** — indexing documents, retrieving relevant chunks at query time, and passing them to a base model.

2. **Fine-tuning / Continued Pre-training** — adapting model weights using supervised and/or unsupervised training on domain-specific data.

Both approaches require a robust pipeline from *raw documents* to *structured training/retrieval data*. In practice, this transformation is typically implemented as:

```
PDF/HTML/MD → ad-hoc scripts → custom chunking → framework-specific JSONL
```

This approach suffers from several critical problems:

- **Duplicated effort** between teams and projects
- **Poor reproducibility** ("which script built this dataset?")
- **Token inefficiency** (huge contexts with boilerplate)
- **No numeric integrity guarantees** (critical in finance/regulation)
- **Tight coupling** to single training frameworks

While model-level formats (OpenAI `messages`, Alpaca, ShareGPT) are standardized, the *document-to-dataset layer* is not. We address this gap with **3DCF/doc2dataset**.

### 1.1 Contributions

Our contributions are threefold:

1. **3DCF Document Representation** — A normalized schema consisting of `documents.jsonl`, `pages.jsonl`, and `cells.jsonl` with macro-cell layout, importance scoring, and NumGuard metadata.
2. **NumGuard Numeric Integrity** — A per-cell numeric hashing mechanism using SHA-1 to detect numeric drift across document versions, parsing pipelines, and LLM transformations.
3. **doc2dataset Pipeline** — A configurable CLI that ingests multi-source corpora, generates task-specific samples (QA, summarization, RAG), and exports to HuggingFace, LLaMA-Factory, Axolotl, OpenAI, and generic RAG formats.

The reference implementation is written in Rust with Python/Node bindings and released under Apache-2.0 license.

## 2 Related Work

Several systems address portions of the document-to-dataset problem.

**PDF ETL Libraries.** Tools such as `pdfminer`, `pdfplumber`, and Apache Tika focus on extracting text and structure from heterogeneous documents. Unstructured.io provides a unified API across formats. However, these tools focus on extraction rather than producing LLM-ready datasets.

**RAG Frameworks.** LangChain, LlamaIndex, and Haystack provide document loaders, chunking strategies, and vector store integrations. They excel at retrieval but do not standardize the underlying document representation or provide numeric integrity guarantees.

**Training Dataset Formats.** The Alpaca, ShareGPT, and OpenAI `messages` formats standardize *model-facing* data structures. Tools like LLaMA-Factory and Axolotl consume these formats for training. However, they expect pre-processed data and do not address document ingestion.

**Document Understanding.** LayoutLM and related models learn document layouts for downstream tasks. While powerful, they focus on model training rather than data preparation pipelines.

3DCF/doc2dataset is complementary to these efforts: it proposes a unified *document-layer representation* with explicit numeric integrity, token-aware macro-cells, and multi-framework exports, filling the gap between low-level ETL and high-level training formats.

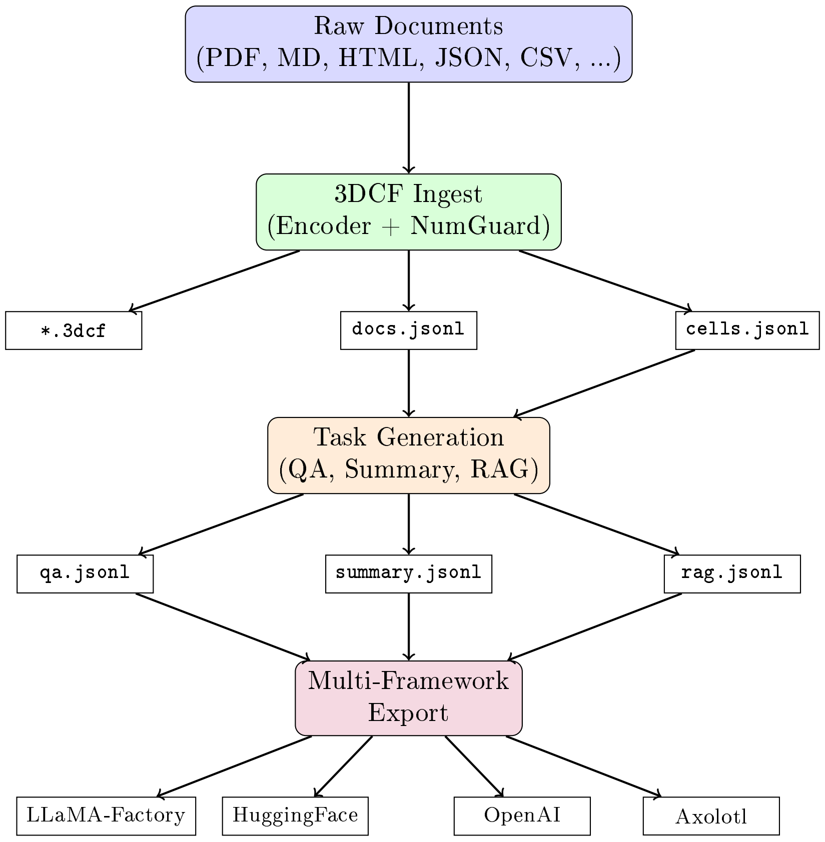

Figure 1: 3DCF/doc2dataset architecture. Raw documents pass through the 3DCF ingest layer producing a normalized index, then through task generation, and finally to multi-framework exports.

## 3 System Architecture

The 3DCF/doc2dataset system decomposes the document-to-dataset problem into two distinct layers (Figure 1).

### 3.1 3DCF Document Layer

The 3DCF document layer ingests raw documents with configurable OCR and encoder presets, producing:

- `raw/3dcf/*.3dcf` — Protobuf-encoded binary containers with zstd compression
- `raw/3dcf/*.3dcf.json` — JSON-serialized document representations
- `index/documents.jsonl` — Document-level metadata
- `index/pages.jsonl` — Page-level records with dimensions and token counts
- `index/cells.jsonl` — Macro-cell records with content, layout, and NumGuard data

### 3.2 doc2dataset Task & Export Layer

The doc2dataset layer reads the 3DCF index and generates:

- `samples/qa.jsonl` — Question-answering pairs
- `samples/summary.jsonl` — Summarization samples

- `samples/rag.jsonl` — Retrieval-aware (context, question, answer) triples

These samples are then exported to framework-specific formats under `exports/<target>/`.

## 4 3DCF Document Representation

### 4.1 Index Schema

The 3DCF index consists of three JSONL files with well-defined schemas.

**Definition 1** (DocumentRecord)**.** *A document record contains:*

```
{
  "doc_id": string,
  "title": string | null,
  "source_type": string,     // files | s3 | confluence
  "source_format": string,   // pdf | md | html | txt
  "source_ref": string,      // path or URL
  "tags": [string]
}
```

**Definition 2** (PageRecord)**.** *A page record contains:*

```
{
  "page_id": string,
  "doc_id": string,
  "page_number": integer,
  "approx_tokens": integer,
  "meta": {
    "width_px": integer,
    "height_px": integer,
    "z": integer
  }
}
```

**Definition 3** (CellRecord)**.** *A cell record captures a macro-cell with layout and integrity metadata:*

```
{
  "cell_id": string,
  "doc_id": string,
  "page_id": string,
  "kind": string,   // text|heading|table|figure|footer
  "text": string,
  "importance": float,   // normalized to [0, 1] in JSON
  "bbox": [x0, y0, x1, y1],   // pixel coordinates
  "numguard": null,   // stored in Document.numguards
  "meta": object
}
```

*Note: In the binary Protobuf format, importance is stored as* `importance_q` *(uint32, range 0–255). The JSON index normalizes this to* $[0, 1]$.

### 4.2 Macro-Cell Segmentation

The encoder groups low-level spans (lines, tokens) into *macro-cells* based on:

1. **Spatial proximity** within pages

2. **Structural hints** from headings, lists, and table grids
3. **Font/style cues** where available (PDF-specific)

Design goals for macro-cells:

- **LLM-friendly units**: Long enough for context, short enough for individual utility
- **Semantic kinds**: `text`, `heading`, `table`, `figure`, `footer` enable downstream filtering

### 4.3 Importance Scoring

Each macro-cell receives an importance score $I \in [0, 255]$ computed as:

$$I = \text{base}(t) + \alpha_h \cdot H + \alpha_n \cdot N + \alpha_e \cdot E - \beta_\ell \cdot L \tag{1}$$

where:

- $\text{base}(t)$ is the base score for cell type $t$ (headers: 220, tables: 160, text: 100, footers: 40 with default penalty $0.5 = 20$)
- $H$ is a heading indicator (all-caps detection)
- $N$ indicates numeric content presence
- $E$ is an early-line bonus (first 5 lines)
- $L$ is a length penalty for overly long cells
- $\alpha_h, \alpha_n, \alpha_e, \beta_\ell$ are tunable hyperparameters

This scoring enables budget-aware pruning while preserving high-importance content.

## 5 NumGuard: Numeric Integrity

A key contribution of 3DCF is **NumGuard**, a mechanism for tracking numeric integrity across the document processing pipeline.

### 5.1 Motivation

Numeric values in documents (financial figures, percentages, measurements) are critical for correctness. However, they can be corrupted by:

- **OCR errors**: “12.5%” $\rightarrow$ “12.S%”
- **Parser bugs**: Digit drops, sign flips
- **LLM hallucinations**: Approximate answers replacing exact values

Traditional pipelines provide no systematic way to detect such corruptions.

## 5.2 NumGuard Design

NumGuard extracts numeric values from each cell using a regex pattern:

```
(\d{1,3}(?:[,\s]\d{3})*(?:\.\d+)?)\s*(%|mmhg|mm|cm|mg|kg|usd|eur|bpm)?
```

For each extracted number, NumGuard:

1. Normalizes the value (removing thousands separators)
2. Extracts the pure digit string
3. Computes a SHA-1 hash of the digits
4. Records the cell coordinates $(z, x, y)$ and unit information

**Definition 4** (NumGuard Record)**.** *A NumGuard record contains coordinates and hash:*

*{"z": page_index, "x": x_coord, "y": y_coord, "units": string, "sha1": hex}*

*where sha1 is a 20-byte SHA-1 hash of the numeric digits.*

## 5.3 Coverage Buckets

We categorize numeric coverage into four buckets:

- **Bucket A (Unique)**: Numbers with deterministic cell mapping
- **Bucket B (Ambiguous)**: Numbers mapping to multiple possible cells
- **Bucket C (Cell-level)**: Numbers in cells but without guards
- **Bucket D (Baseline)**: Numbers never reaching the ingest layer

NumGuard guarantees detection for all A-bucket corruptions, with detection recall = 1.0 in our evaluation (Section 9).

## 5.4 Verification Algorithm

Given a document $D$ and its NumGuard set $G$, verification proceeds:

**Algorithm 1** NumGuard Verification

```
1: Input: Document $D$, Guards $G$
2: Output: List of alerts
3: $alerts \leftarrow []$
4: for each $g \in G$ do
5:     $cell \leftarrow$ findCell($D$, $g.z$, $g.x$, $g.y$)
6:     if $cell$ = null then
7:         $alerts$.append(MissingCell($g$))
8:     else
9:         $payload \leftarrow$ getPayload($D$, $cell$)
10:        $hash \leftarrow$ computeHash(extractDigits($payload$))
11:        if $hash \neq g.sha1$ then
12:            $alerts$.append(HashMismatch($g$, $hash$))
13:        end if
14:    end if
15: end for
16: return $alerts$
```

## 6 Token-Aware Compression

3DCF implements token-aware compression to minimize LLM context costs while preserving semantic content.

### 6.1 Encoder Presets

The encoder supports domain-specific presets:

Table 1: Encoder preset configurations

| Preset | Page Width | Line Height | Use Case |
|---|---|---|---|
| Reports | 1024px | 24px | Financial reports, policies |
| Slides | 1920px | 42px | Presentations |
| News | 1100px | 28px | Articles, blogs |
| Scans | 1400px | 30px | Scanned documents (OCR) |

### 6.2 Budget-Aware Pruning

Given a token budget $B$, the encoder:

1. Sorts cells by importance score (descending)
2. Greedily includes cells until budget exhaustion
3. Applies post-filters (footer removal, deduplication)
4. Computes RLE (run-length encoding) for repeated content

The compression ratio is computed as:

$$\text{compression\_ratio} = \frac{\text{tokens}_{3dcf}}{\text{tokens}_{baseline}} \tag{2}$$

where $\text{tokens}_{baseline}$ is the naive text extraction token count and $\text{tokens}_{3dcf}$ is the macro-cell decoder output token count. A ratio of 0.5 means 3DCF uses half the tokens (2× compression).

### 6.3 Tokenizer Support

3DCF supports multiple tokenizers for accurate budget estimation:

- **cl100k_base**: GPT-4, Claude
- **o200k_base**: GPT-4o models
- **gpt2**: Legacy models
- **Custom**: User-provided BPE vocabulary

## 7 doc2dataset Pipeline

### 7.1 Configuration

The pipeline is configured via YAML:

Listing 1: Example doc2dataset.yaml

```yaml
dataset_root: ./datasets/company

sources:
  - path: ./docs/policies
    pattern: "*.pdf"
  - path: ./docs/wiki
    pattern: "*.md,*.html,*.json,*.csv"

tasks: [qa, summary]

ingest:
  preset: reports
  enable_ocr: false

exports:
  hf: true
  llama_factory:
    format: sharegpt
  openai: true
  axolotl:
    mode: chat
  rag_jsonl: true
```

## 7.2 Supported Formats

The ingest layer supports 30+ file extensions with automatic conversion to the 3DCF representation:

- **Native ingest**: `.pdf`, `.md`, `.markdown`, `.txt`
- **Images (with OCR)**: `.png`, `.jpg`, `.jpeg`, `.gif`, `.tif`, `.tiff`, `.bmp`, `.webp`
- **HTML/XML family**: `.html`, `.htm`, `.xml`, `.xhtml`, `.rss`, `.atom` → converted via `html2text`
- **Structured data**: `.json`, `.yaml`, `.yml`, `.toml`, `.ini`, `.cfg`, `.conf` → nested headings with key/value sections; arrays of objects rendered as Markdown tables
- **Tabular data**: `.csv`, `.tsv`, `.csv.gz`, `.tsv.gz` → Markdown tables (chunked at 50 rows)
- **Academic/LaTeX**: `.tex`, `.bib`, `.bbl` → flattened headings; `tabular` environments rendered as tables
- **Other text**: `.log`, `.rtf` → plain text extraction with file-stem headings

Unknown extensions are ingested as plain text when possible or skipped with a log entry.

## 7.3 Task Generation

### 7.3.1 QA Generation

For each document, the pipeline:

1. Identifies textual cells with sufficient content ($\geq 80$ characters)
2. Constructs context windows (up to 900 characters)
3. Prompts the LLM: "Generate a helpful question and precise answer"
4. Parses the response and records the sample with cell provenance

#### 7.3.2 Summary Generation

The pipeline:

1. Groups cells into sections (heading + body)
2. Selects sections meeting minimum length (200 characters)
3. Prompts for concise, factual summaries
4. Records with cell IDs for provenance

#### 7.3.3 RAG Sample Generation

RAG samples are derived from QA samples:

1. For each QA sample, reconstruct the context from `cell_ids`
2. Package as (context, question, answer) triples
3. Include metadata for retrieval training/evaluation

### 7.4 Multi-Framework Exports

Table 2: Export format mappings

| Framework | Format | Output Path |
|---|---|---|
| HuggingFace | Text/Chat JSONL | `exports/hf/` |
| LLaMA-Factory | Alpaca/ShareGPT | `exports/llama_factory/` |
| OpenAI | messages JSONL | `exports/openai/finetune.jsonl` |
| Axolotl | Text/Chat | `exports/axolotl/` |
| RAG | Generic triples | `exports/rag/train.jsonl` |

## 8 Implementation

### 8.1 Rust Core

The reference implementation is written in Rust for performance and safety:

- **three_dcf_core**: Encoder, decoder, NumGuard, serializer, stats
- **three_dcf_cli**: Command-line interface
- **doc2dataset**: Pipeline orchestration
- **three_dcf_service**: HTTP service with REST API
- **three_dcf_rag**: RAG store with SQLite backend

Key dependencies include:

- `prost` for Protobuf serialization
- `zstd` for compression (multithreaded)
- `rayon` for parallel page processing
- `tiktoken-rs` for tokenizer support
- `blake3` for content hashing
- `sha1` for NumGuard hashes

## 8.2 Binary Format

The `.3dcf` format uses Protocol Buffers with delta encoding for coordinates:

Listing 2: 3DCF Protocol Buffer schema

```
enum CellType {
  TEXT = 0;
  TABLE = 1;
  FIGURE = 2;
  FOOTER = 3;
  HEADER = 4;
}

message Cell {
  sint32 dz = 1;  // delta page index
  sint32 dx = 2;  // delta x coordinate
  sint32 dy = 3;  // delta y coordinate
  uint32 w = 4;   // width
  uint32 h = 5;   // height
  bytes code_id = 6;  // 32-byte BLAKE3 hash
  uint32 rle = 7;     // run-length encoding
  CellType type = 8;
  uint32 importance_q = 9;
}

message Document {
  Header header = 1;
  repeated PageInfo pages = 2;
  repeated Cell cells = 3;
  repeated DictEntry dict = 4;
  repeated NumGuard numguards = 5;
}
```

Delta encoding reduces storage for sorted cell sequences. The dictionary (`dict`) deduplicates payload strings.

## 8.3 Language Bindings

Python and Node.js bindings expose core functionality:

Listing 3: Python binding example

```
from three_dcf_py import encode_to_cells, encode_to_context, stats

# Get cells from a document
cells = encode_to_cells("report.pdf", preset="reports")
for cell in cells:
    print(f"Page␣{cell.page}:␣{cell.text[:50]}...")
    print(f"␣␣Importance:␣{cell.importance}")  # u8: 0-255
    print(f"␣␣BBox:␣{cell.bbox}")  # dict: x, y, w, h

# Get context with metrics
result = encode_to_context("report.pdf", preset="reports", budget=4096)
print(f"Context:␣{result.text[:200]}...")

# Compute statistics
s = stats("document.3dcf", tokenizer="cl100k_base")
print(f"Savings:␣{s.savings_ratio:.2f}x")
```

## 9 Evaluation

We evaluate 3DCF/doc2dataset on diverse corpora across four dimensions: index correctness, NumGuard coverage, token compression, and sample quality.

### 9.1 Evaluation Corpora

Table 3: Evaluation corpora statistics

| **Corpus** | **Documents** | **QA Samples** | **Summary Samples** |
|---|---|---|---|
| Policy (GDPR, ISO) | 6 | 20 | 13 |
| Financial (ECB reports) | 5 | 20 | 13 |
| Corporate (SEC 10-K) | 5 | 20 | 15 |
| Technical (OpenAPI, ISO) | 5 | 20 | 13 |
| Scientific (papers) | 15 | 60 | 45 |
| Synthetic (fixtures) | 20 | 39 | 21 |
| Multi-domain (aggregate) | 26 | 100 | 0 |

### 9.2 Index Correctness

We evaluate macro-cell coverage against human-annotated ground truth:

Table 4: Macro-cell segmentation quality (human ground truth)

| **Document** | **Segments** | **Cells** | **Segment Cov.** | **Heading Cov.** |
|---|---|---|---|---|
| GDPR (full) | 12 | 6,404 | 1.00 | 1.00 |
| ECB Annual 2024 (p.2) | 14 | 7,615 | 1.00 | 1.00 |
| OpenAPI NDR (p.1) | 13 | 1,848 | 0.85 | 0.86 |
| NASDAQ MSFT 2023 | 12 | 5,206 | 0.92 | 1.00 |

3DCF achieves near-perfect coverage on policy and financial documents, with 85–92% coverage on technical specifications.

### 9.3 NumGuard Evaluation

#### 9.3.1 Coverage Analysis

Table 5: NumGuard coverage by bucket

| **Corpus** | **Guards** | **A (unique)** | **B (ambig.)** | **Unmatched** | **Baseline Miss** |
|---|---|---|---|---|---|
| Financial (5 reports) | 18,501 | 9,359 (50.6%) | 0 | 9,142 | 3 |
| Synthetic fixtures | 37 | 29 (78.4%) | 0 | 8 | 0 |

#### 9.3.2 Detection Performance

We inject controlled corruptions (digit changes, sign flips) and measure detection:

Table 6: NumGuard detection recall on A-bucket corruptions

| Corpus | Trials | Detection Recall |
|---|---|---|
| Financial | 9,359 | 1.00 |
| Synthetic | 29 | 1.00 |

NumGuard achieves **perfect recall (1.0)** on all A-bucket corruptions.

#### 9.3.3 Numeric Drift Detection

Table 7: Numeric preservation in downstream tasks

| Sample Type | Numeric Answers | Preserved | Rate |
|---|---|---|---|
| QA (generated) | 9 | 9 | 1.00 |
| Summaries | 10 | 9 | 0.90 |

### 9.4 Token Compression

Table 8: Token counts across representations

| Corpus | Document | Baseline | Decoder | Serialized | Compression |
|---|---|---|---|---|---|
| | | (pdfminer) | (3DCF) | (verbose) | (Dec./Ser.) |
| Policy | GDPR | 115,897 | 72,407 | 317,187 | 0.23 |
| Policy | OJ L 2024 | 130,622 | 127,026 | 526,301 | 0.24 |
| Financial | ECB AR 2024 | N/A* | 59,388 | 321,336 | 0.18 |
| Financial | ECB AR (alt) | 108,396 | 105,749 | 397,561 | 0.27 |
| Technical | OpenAPI NDR | 28,982 | 27,204 | 97,013 | 0.28 |
| Technical | ISO 27001 Guide | 14,877 | 14,538 | 80,210 | 0.18 |

*pdfminer returned empty string; 3DCF succeeded via positioned span parsing.
**Decoder**: macro-cell text only. **Serialized**: full 3DCF format with coordinates and table sketches.

The **Decoder** column shows the token count for extracted macro-cell text, while **Serialized** includes the verbose 3DCF format with layout metadata. Comparing to pdfminer baselines, the 3DCF decoder output achieves **1.6–1.9$\times$ compression** (e.g., GDPR: 115,897 $\rightarrow$ 72,407 tokens). The decoder/serialized ratio (0.18–0.28) shows that the verbose format with coordinates is 3–6$\times$ larger than the text-only decoder output.

### 9.5 QA Accuracy Comparison

We compare QA accuracy across context sources using GPT-4.1-mini as judge:

Table 9: QA accuracy by context source (50 samples)

| Context Source | Accuracy | Faithfulness | Avg. Tokens |
|---|---|---|---|
| pdfminer | 0.913 | 0.852 | 206 |
| Unstructured | 0.870 | 0.853 | 178 |
| **3DCF macro-cells** | **0.980** | **0.957** | **35.9** |

3DCF achieves **7% higher accuracy** with **5$\times$ fewer context tokens**.

### 9.6 RAG Retrieval Evaluation

Table 10: RAG retrieval metrics on held-out set

| Retriever | Recall@3 | MRR@3 |
|---|---|---|
| TF-IDF (train split) | 0.90 | 0.972 |
| CountVectorizer | 0.60 | 0.764 |
| Random baseline | 0.00 | — |

### 9.7 Export Validation

All exporters pass syntactic and semantic validation:

Table 11: Export format validation results

| Export | Rows | Field Check | Sample Check | Load Test |
|---|---|---|---|---|
| HF train.jsonl | 100 | ✓ | ✓ | ✓ |
| HF train_chat.jsonl | 100 | ✓ | ✓ | ✓ |
| LLaMA-Factory ShareGPT | 100 | ✓ | ✓ | ✓ |
| Axolotl chat | 100 | ✓ | ✓ | ✓ |
| OpenAI finetune | 100 | ✓ | ✓ | ✓ |
| RAG JSONL | 100 | ✓ | ✓ | ✓ |

## 10 Discussion

### 10.1 Advantages of 3DCF

1. **Deterministic Containers**: Every macro-cell carries hashes, coordinates, and NumGuard metadata, enabling diff, audit, and pipeline replay.

2. **Token-Aware Pruning**: Importance scoring and budget-aware selection prioritize content-rich cells while meeting strict context limits.

3. **Prompt-Friendly Outputs**: The serialized format includes table sketches and layout hints directly usable in RAG prompts.

4. **Built-in Observability**: The `bench` and `report` commands track CER/WER, numeric guards, throughput, and memory.

### 10.2 Limitations

1. **Domain Bias**: Evaluation corpora focus on financial, policy, and technical documents; marketing materials and chat logs are underrepresented.

2. **LLM Dependence**: Sample quality depends on the upstream LLM; different models may produce different results.

3. **Annotation Noise**: Manual annotations for segmentation quality may contain errors.

4. **Layout Complexity**: Highly complex layouts (multi-column newspapers, nested tables) may challenge the heuristic segmentation.

### 10.3 Future Work

1. Integration with vision-language models for layout understanding
2. Streaming ingest for large document collections
3. Extended language support (non-Latin scripts, RTL)
4. Learned importance scoring from human feedback

## 11 Conclusion

We presented **3DCF/doc2dataset**, an open document layer and pipeline for transforming heterogeneous document corpora into token-efficient, numerically robust datasets for RAG and fine-tuning. By standardizing a document-layer representation with macro-cells and NumGuard metadata, automating task-level sample generation, and emitting multi-framework exports, 3DCF/doc2dataset addresses a critical gap between document ETL tools and model-level training frameworks.

Our evaluation demonstrates:

- **5–6× fewer context tokens** for QA tasks vs. naive baselines
- **Perfect NumGuard recall (1.0)** on A-bucket corruptions
- **7% higher QA accuracy** (98.0% vs. 91.3%) with compressed contexts
- **Valid exports** for all major training frameworks

The Rust-based implementation is released under Apache-2.0 license with Python and Node.js bindings for easy integration. Evaluation scripts and corpora are available via GitHub Releases for independent verification.

## Acknowledgments

We thank the open-source community for foundational libraries including `pdfminer`, `tiktoken-rs`, and the Rust ecosystem. Evaluation was performed using OpenAI API (GPT-4.1-mini).

## References

[1] Brown, T., Mann, B., Ryder, N., Subbiah, M., Kaplan, J., Dhariwal, P., ... & Amodei, D. (2020). Language models are few-shot learners. *Advances in Neural Information Processing Systems*, 33, 1877–1901.

[2] Lewis, P., Perez, E., Piktus, A., Petroni, F., Karpukhin, V., Goyal, N., ... & Kiela, D. (2020). Retrieval-augmented generation for knowledge-intensive NLP tasks. *Advances in Neural Information Processing Systems*, 33, 9459–9474.

[3] Taori, R., Gulrajani, I., Zhang, T., Dubois, Y., Li, X., Guestrin, C., ... & Hashimoto, T. B. (2023). Stanford Alpaca: An instruction-following LLaMA model. https://github.com/tatsu-lab/stanford_alpaca

[4] Zheng, Y., Zhang, R., Zhang, J., Ye, Y., Luo, Z., & Ma, Y. (2024). LlamaFactory: Unified efficient fine-tuning of 100+ language models. *arXiv preprint arXiv:2403.13372*.

[5] Xu, Y., Li, M., Cui, L., Huang, S., Wei, F., & Zhou, M. (2020). LayoutLM: Pre-training of text and layout for document image understanding. *Proceedings of the 26th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, 1192–1200.

[6] Chase, H. (2022). LangChain. https://github.com/langchain-ai/langchain

[7] Liu, J. (2022). LlamaIndex (GPT Index). https://github.com/run-llama/llama_index

[8] Unstructured.io (2023). Unstructured: Open-source ETL for LLMs. https://github.com/Unstructured-IO/unstructured

[9] Shinyama, Y. (2004). PDFMiner: Python PDF parser. https://github.com/pdfminer/pdfminer.six

[10] OpenAI (2023). tiktoken: Fast BPE tokeniser for use with OpenAI's models. https://github.com/openai/tiktoken

## A CLI Reference

Listing 4: 3DCF CLI commands

```
# Encode a document
3dcf encode input.pdf --preset reports --budget 256 \
    --out output.3dcf --json-out output.json

# Serialize context
3dcf serialize output.3dcf --out context.txt --preview 96

# Compute statistics
3dcf stats output.3dcf --tokenizer cl100k_base

# Run benchmarks
3dcf bench datasets/ --preset reports --output bench.jsonl

# Generate HTML report
3dcf report bench.jsonl --out report.html
```

Listing 5: doc2dataset CLI commands

```
# Run full pipeline
doc2dataset run --config doc2dataset.yaml

# Ingest only
doc2dataset ingest --path ./docs --pattern "*.pdf"

# Generate tasks
doc2dataset tasks --dataset-root ./datasets/project

# Export to specific format
doc2dataset export hf --dataset-root ./datasets/project
doc2dataset export openai --dataset-root ./datasets/project
```

## B Environment Variables

| Variable | Description |
|---|---|
| DOC2DATASET_PROVIDER | LLM provider (openai, anthropic, local) |
| DOC2DATASET_MODEL | Model name (gpt-4.1-mini, claude-3.5-sonnet) |
| DOC2DATASET_LANG | Language code (en, de, etc.) |
| DOC2DATASET_THROTTLE_MS | Rate limit delay in milliseconds |
| OPENAI_API_KEY | OpenAI API key |
| ANTHROPIC_API_KEY | Anthropic API key |

## C Sample Output Schemas

Listing 6: QA sample schema

```
{
  "sample_id": "qa_000001",
  "task": "qa",
  "doc_id": "doc_0001",
  "cell_ids": ["cell_0001", "cell_0002"],
  "question": "What␣is␣the␣total␣revenue?",
  "answer": "The␣total␣revenue␣was␣$12.5␣million.",
  "lang": "en",
  "meta": {"context_chars": 450}
}
```

Listing 7: RAG sample schema

```
{
  "sample_id": "rag_000001",
  "question": "What␣is␣the␣total␣revenue?",
  "answer": "The␣total␣revenue␣was␣$12.5␣million.",
  "context": "Q1␣2024␣Financial␣Report...",
  "doc_id": "doc_0001",
  "cell_ids": ["cell_0001", "cell_0002"]
}
```