LLM Data Preparation Pipeline: Design and Implementation

Nicoló Dalmasso

nicolo.dalmasso1@gmail.com

November 21, 2025

Abstract

This report presents a containerized, end-to-end data preparation pipeline for large language model training. The four-stage sequential architecture implements streaming input, batch processing, deterministic sharding, and file consolidation. Key features include comprehensive quality metrics, PII detection and removal, language filtering, and deduplication.

1 Introduction

LLM performance critically depends on training data quality. Raw text contains duplicates, PII, non-target languages, and noise that degrade model performance. This pipeline addresses these challenges through systematic four-stage processing, transforming raw JSONL into clean, tokenized sequences.

The design prioritizes simplicity over parallelism for debuggability, employs streaming architecture for memory efficiency, and implements fail-fast validation with comprehensive error tracking. Core objectives are quality (noise/duplicate removal), safety (PII elimination), appropriate language coverage, efficient processing, and full observability.

2 Pipeline Architecture

2.1 Four-Stage Design

Stage 1 – Light Filtering: Streaming input with empty text detection and JSON validation. Uses generator pattern to avoid memory overhead.

Stage 2 – Heavy Processing: Batch processing (1,000 records) implements language detection (`langdetect`, first 500 chars), regex-based PII removal (emails → `[EMAIL]`, phones → `[PHONE]`), tokenization (`tiktoken cl100k_base`), hash-based deduplication (O(1) lookup), and length filtering (10-1000 tokens). This stage produces both tokenized sequences and cleaned text outputs.

Stage 3 – Shuffle & Shard: Deterministic shuffle (seed=42) and export to 5,000-record JSONL shards for distributed training compatibility.

Stage 4 – Consolidation: Merges all shard files into unified JSONL outputs: `all_processed_text.jsonl` (submission format with {`"text": "..."`}) and `all_processed.jsonl` (tokenized sequences). This stage enables centralized analysis and simplified model ingestion.

2.2 Containerization

Docker container (`python:3.10-slim`) provides frozen dependencies, cross-environment portability, and isolation from system-level conflicts.

3 Evaluation Metrics

INPUT Inspection: Pre-processing analysis measures language distribution (coverage), PII detection rates (safety baseline), and confidence scores (quality proxy).

OUTPUT Inspection: Post-processing validates deduplication effectiveness, token length distributions, drop reasons breakdown, PII removal ($< 1\%$ target), vocabulary richness and lexical diversity (quality), cosine similarity (linguistic diversity), and throughput metrics.

4 Results

The pipeline processed a total of 282,269 records, retaining 229,715 (81.4%) after quality filtering. Table 1 summarizes the key metrics evaluated before and after processing. Details are provided below.

Metric	Input	Output
Total Records	282,269	229,715
Duplicates	940	0
PII Hit Rate	1.9%	0.0%
Avg Token Length	—	222
Lexical Diversity	—	0.732

Table 1: Quality metrics: input and output datasets.

Language distribution: Language detection confidence measures how reliably the language classifier identifies English text. The dataset shows a high mean confidence of 0.985, indicating strong language consistency (Figure 1).

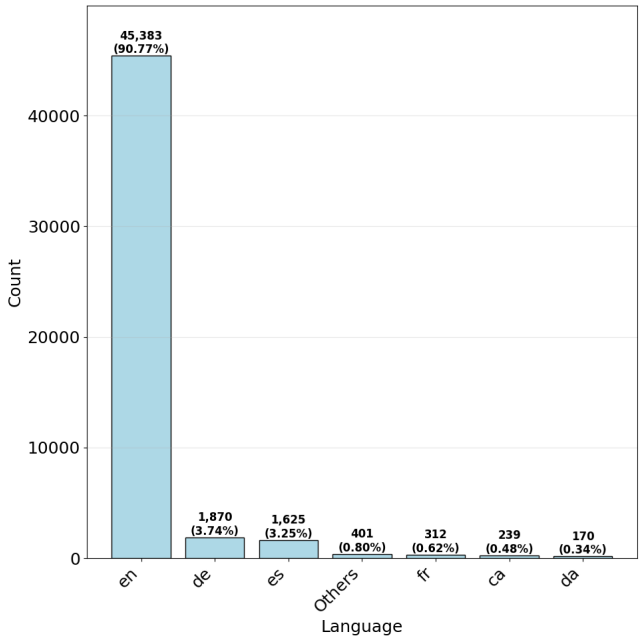

Figure 1: Distribution of detected languages in the raw input corpus.

PII detection: PII detection identifies sensitive personal information that must be removed for compliance and safety. The input contained 1.9% PII-positive texts, and post-processing reduced this to zero (Figure 2).

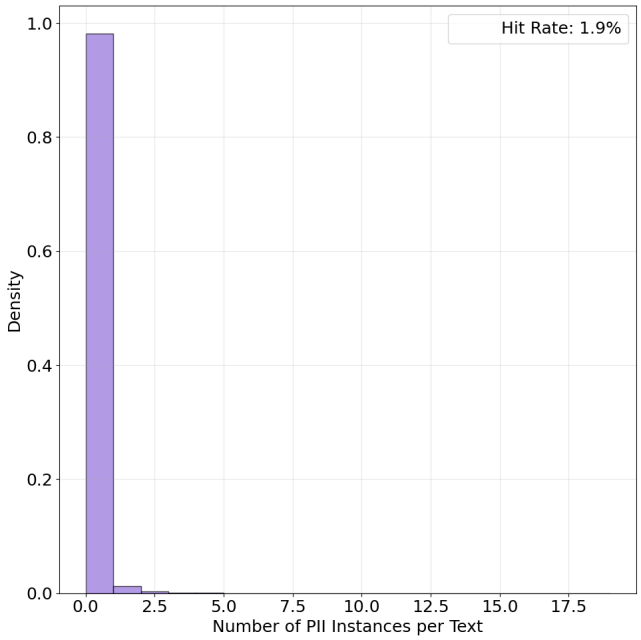

Figure 2: Histogram of PII instances per text in the input dataset, showing the density distribution of detected patterns.

Token distribution: Token length measures text size and helps enforce bounds that prevent extremely short or long sequences. The processed corpus follows a log-normal distribution with a mean of 222 tokens, within the enforced 10–1000 range (Figure 3).

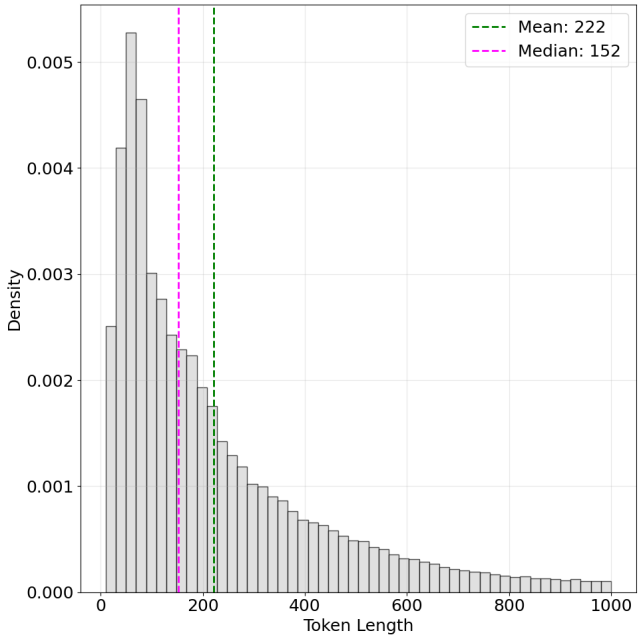

Figure 3: Normalized density distribution of token sequence lengths in the processed output dataset.

Drop reasons: Drop reasons quantify which filters

remove data and help understand quality bottlenecks. Main removals were non-English content (47.8%), length filtering (25.9%), PII (24.5%), and duplicates (1.8%) (Figure 4).

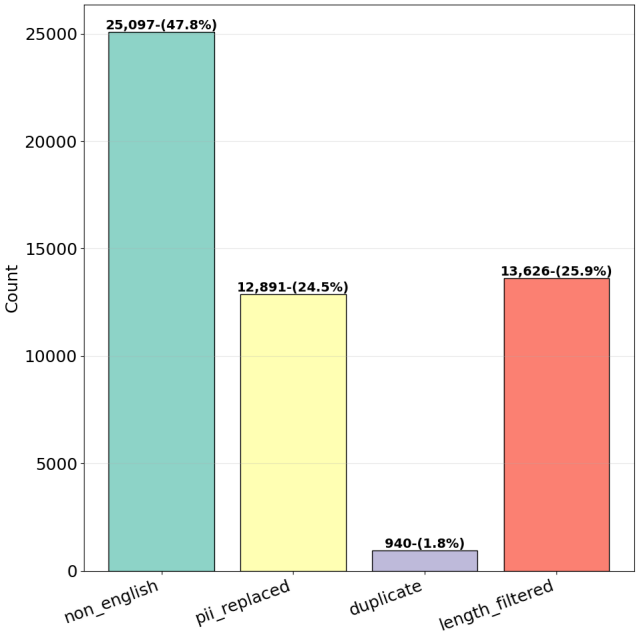

Figure 4: Categorical breakdown of filtering criteria showing the distribution of records removed by each processing stage.

Quality metrics: Lexical diversity estimates vocabulary richness, with higher values indicating more varied language. The dataset shows a mean lexical diversity of 0.732 (Figure 5). Cosine similarity measures redundancy by quantifying how similar texts are to one another. The mean similarity of 0.216 confirms low redundancy and high content variety (Figure 6).

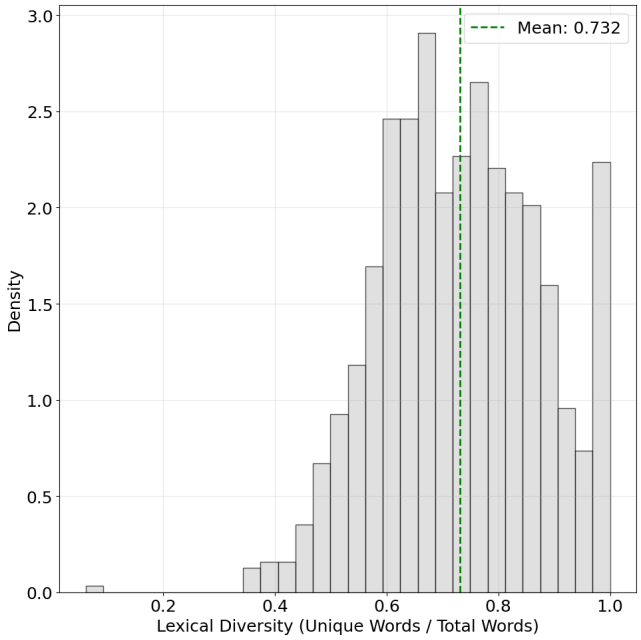

Figure 5: Distribution of lexical diversity scores across output texts, computed as the ratio of unique to total words.

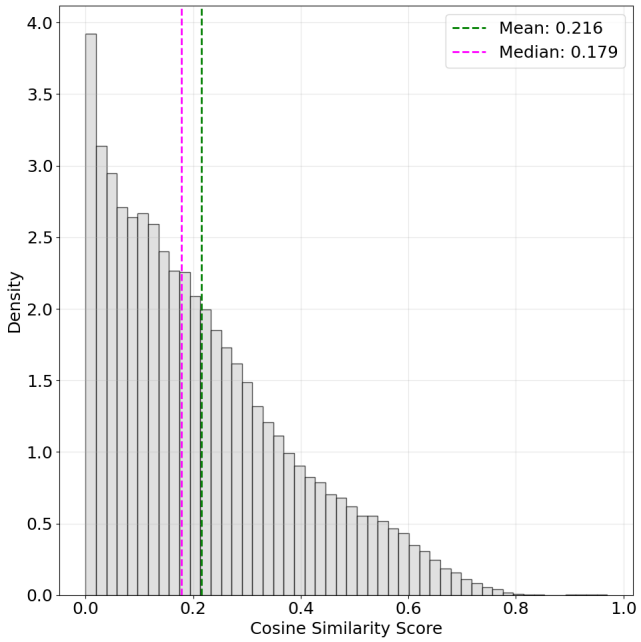

Figure 6: Pairwise cosine similarity distribution for a sample of tokenized sequences, measuring content overlap.

5 Scalability & Improvements

The following scalability considerations outline industry-standard strategies for large-scale text-processing pipelines. These approaches go beyond the scope of the current implementation but represent realistic directions for extending the system to production-level or multi-million–record workloads.

5.1 Current Limitations

The current pipeline processes data sequentially, which limits throughput when scaling to tens or hundreds of millions of records. In-batch deduplication may miss duplicates spread across different batches. Regex-based PII detection has limited recall, especially for unstructured formats. Additionally, sampling only the first 500 characters may misclassify code-switched or multilingual texts.

5.2 Parallelization & Performance

Distributed Framework: For large-scale deployments, migrating to a distributed framework such as Apache Spark or Ray can provide horizontal scaling and improved fault-tolerance. Spark offers mature ETL capabilities, while Ray provides flexible integration with machine-learning workloads. Useful optimizations include hash-based partitioning for deterministic data distribution, broadcast variables for shared resources (e.g., regex patterns, models), and checkpointing for failure recovery.

Multi-threading: On a single machine, batch operations can be parallelized using Python's `multiprocessing` or `concurrent.futures`, enabling significant speedups on multi-core systems.

GPU Acceleration: Tokenization and language detection can be offloaded to GPUs using frameworks such as RAPIDS cuDF, which may provide large performance gains for compute-heavy operations.

Caching Strategy: Using a cache layer (e.g., Redis or Memcached) for deduplication hash lookups can improve performance and reduce repeated computation across batches.

5.3 Scalability Improvements

Advanced Deduplication: Techniques such as MinHash LSH enable near-duplicate detection in sub-linear time, while distributed Bloom filters can support efficient exact deduplication across nodes.

Small-File Mitigation: As datasets grow, storing data in large shards (100K–1M records) and using columnar formats like Parquet or ORC helps reduce filesystem overhead. Keeping file counts manageable (e.g., below 10,000) is important for efficient metadata handling at scale.

Memory-Bounded Shuffle: For datasets exceeding RAM capacity, external sorting with disk-backed merging or reservoir sampling helps maintain performance without requiring large in-memory buffers.

Streaming Pipeline: Frameworks such as Apache Beam or Flink can replace batch processing with streaming architectures for continuous ingestion and near real-time processing.

5.4 Data Quality Enhancements

Future iterations could incorporate additional quality filters commonly used in large text-corpora construction, such as perplexity-based filtering (e.g., with GPT-2) to detect noisy or templated content; toxicity detection using models like Detoxify or the Perspective API; boilerplate removal with tools such as `trafilatura` or `jusText`; and transformer-based NER systems for more comprehensive PII detection, though these methods come with substantially higher compute costs.

5.5 Monitoring & Fault Tolerance

A production-grade system requires real-time metrics (Prometheus + Grafana), distributed tracing (OpenTelemetry), automated quality alerts, and resource monitoring. Typical failure modes include out-of-memory events during large writes, corrupted or malformed input files, straggler tasks during parallel execution, and non-determinism arising from uncontrolled random seeds.