LLM Project Data Ingestion: Processing Steps (Data Engineering View)

This guide explains the processing steps to take raw data (docs, tickets, PDFs, web pages, chats, code, database rows) and turn it into something an LLM system can safely and reliably ingest.

It covers two common scenarios:

- **RAG ingestion** (Retrieval-Augmented Generation): you index knowledge so the LLM can retrieve it at question time.
- **Training / fine-tuning ingestion**: you prepare datasets so the model can learn from them.

Most production LLM projects do both: RAG for up-to-date knowledge, and fine-tuning for behavior/style.

---

0) Define the ingestion goal (this changes everything)

Before you write pipelines, answer these three questions:

1. **What is the LLM supposed to do?**

- answer questions from internal docs?
- summarize tickets?
- generate SQL?
- draft emails in a company style?

1. **Where should the truth come from?**

- from retrieved docs (RAG)
- from model weights (training)
- from both

1. **What is the safety / compliance boundary?**

- PII? secrets? regulated docs?
- retention requirements?

- “who can see what” (ACLs) must be preserved end-to-end.

If you skip this step, you’ll build a beautiful ingestion pipeline that indexes the wrong thing perfectly.

---

1) Source inventory + access + contracts

What you do

- Make a list of sources (examples):

- Confluence / Notion

- Google Drive / SharePoint

- Slack / Teams

- Jira / Zendesk

- Git repos

- Databases / data warehouse

- Websites / knowledge bases

- For each source define:

- **ownership** (who approves ingestion)

- **legal/compliance** constraints

- **expected update frequency**

- **schema / document model**

- **ACL model** (who can access)

Outputs

- A source catalog (even a simple table) and a clear data contract.

---

2) Extraction (getting raw content out)

Common extraction patterns

- **APIs**: best for structured and incremental sync.
- **Web crawling**: for public/internal sites without APIs (be careful).
- **File sync**: PDFs, docs, HTML exports.
- **CDC / change logs**: for databases.

Key data-engineering concerns

- **Incremental ingestion**: don't re-process everything every day.
- **Idempotency**: re-running a job should not create duplicates.
- **Backfills**: support historical re-ingestion when logic changes.

Outputs

- Raw snapshots stored in durable storage (object storage is typical).
- Metadata per item: `source`, `source_id`, `timestamp`, `version`, `hash`, `owner`, `acl`.

---

3) Parsing + normalization (turning "formats" into text + structure)

Raw data comes in messy containers:

- PDFs (text + layout)
- Word/Google docs
- HTML pages
- emails
- chat threads
- images (scanned PDFs)

What you do

- Convert each item into a **canonical document structure**.

Suggested canonical shape:

- `doc_id`
- `source`
- `title`
- `body_text`
- `sections` (optional)
- `tables` (optional)
- `code_blocks` (optional)
- `created_at`, `updated_at`
- `authors` (optional)
- `acl` (required for internal systems)
- `source_url`

Parsing tips

- **HTML**: remove nav/ads/footers; keep headings.
- **PDF**: preserve reading order; watch out for two-column layouts.
- **Scanned docs**: OCR (quality varies; store OCR confidence).
- **Tables**: either keep as markdown or extract to structured rows.

Output

- Normalized documents (text + metadata) stored in a “silver” layer.

---

4) Cleaning (make text useful, not pretty)

Cleaning is not about making text “nice.” It is about making it:

- searchable
- chunkable

- safe
- consistent

Typical cleaning steps

- Remove boilerplate repeated on every page (headers/footers).
- Normalize whitespace, encoding, and weird characters.
- Fix broken hyphenation from PDFs (e.g., `infor- mation`).
- Remove or tag auto-generated content.

Output

- Cleaned `body_text` plus a log of what was removed.

---

5) Deduplication (the silent killer of RAG quality)

Duplicates waste tokens and ruin retrieval.

What you do

- **Exact dedup**: hash exact text.
- **Near-dup dedup**: detect similar documents/sections (common with mirrored pages, copied docs, repeated policy templates).

Output

- A stable mapping: `doc_id -> canonical_doc_id`.

---

6) Safety: PII/Secrets redaction + policy filtering

For most companies, this is non-negotiable.

What you do

- Detect and remove/mask:

- PII (emails, phone numbers, addresses, national IDs)
- secrets (API keys, tokens, passwords)
- highly sensitive info (depending on policy)

- Optionally classify documents into tiers (public/internal/confidential).

Outputs

- Redacted text (or masked spans) + an audit trail.
- A decision: some documents are **excluded** from indexing/training.

---

7) Language detection + routing

If you have multiple languages, you may want:

- per-language indexes
- language-specific tokenizers
- translation pipelines (only if allowed)

Output

- `language` field per document/chunk.

---

8) Chunking (split documents into model-sized pieces)

Chunking is one of the biggest levers for RAG quality.

Why chunk?

LLMs have context limits. Retrieval works better when the returned snippet is focused.

Chunking principles

- Keep semantic boundaries (headings/sections).
- Avoid splitting mid-sentence when possible.

- Include enough context to be meaningful.

Common strategies

- **Fixed-size tokens** (simple, but may break structure).
- **Section-based** (use headings as chunk boundaries).
- **Hybrid** (section-based then sub-chunk long sections).

Recommended chunk metadata

For each chunk:

- `chunk_id`
- `doc_id`
- `chunk_index`
- `text`
- `title/section_path` (e.g., `Security > Access Control`)
- `source_url`
- `acl`
- `updated_at`

---

9) Enrichment (metadata that boosts retrieval)

Enrichment is everything you wish you had during an incident.

Examples:

- tags / taxonomy (team, product, system)
- document type (runbook, ADR, policy, RCA, ticket)
- owner/team
- freshness score
- “authoritative source” flag

Why it matters:

- you can filter retrieval (only runbooks)
- you can boost results (prefer RCAs for incident questions)
- you can avoid outdated docs (freshness)

---

10) Embeddings (turn text into vectors)

For RAG, you typically create an embedding per chunk.

What you do

- Pick an embedding model.
- Generate vectors for each chunk.
- Store vectors with chunk metadata.

Practical tips

- **Version embeddings**: if you change models, keep both versions until you reindex.
- Track:

- embedding model name/version

- dimensionality

- normalization settings

---

11) Indexing (how the system will retrieve later)

RAG often uses a mix of retrieval methods:

- **Vector search** (semantic similarity)
- **Keyword search** (exact matches)
- **Hybrid search** (often best)

What you store in the index

- chunk text
- embeddings
- metadata filters
- ACL attributes

ACL is critical

If a user can’t access a doc in the source system, the LLM must not retrieve it.

That means:

- store ACLs with chunks
- enforce ACL filtering at retrieval time

---

12) Quality checks (don’t ship a blind index)

Data quality checks

- empty documents / empty chunks
- duplicate rate
- language distribution
- chunk size distribution
- OCR error rate
- PII/secrets detection rate

Retrieval quality checks

Build a small “golden” evaluation set:

- 30–200 real questions
- expected sources (which docs should answer them)

Measure:

- does retrieval return the right chunks?
- do answers cite the right sources?

---

13) Observability + lineage

You need to answer:

- “Where did this answer come from?”
- “Which docs got indexed?”
- “When did we last sync source X?”

Track:

- pipeline runs
- document versions
- chunk counts
- index versions
- embedding versions

---

14) Incremental updates (the real production problem)

LLM ingestion is not a one-time batch.

Common patterns

- **Change-based reprocessing**: only re-chunk and re-embed changed docs.
- **Reindex by version**: keep index v1 running while building v2.
- **Tombstones**: handle deletions correctly.

---

15) Training / fine-tuning ingestion (extra steps)

If you’re preparing data for training or fine-tuning, add these steps:

A) Dataset design

- Decide what behavior you want:

- domain vocabulary?

- writing style?

- better instruction following?

- Choose format:

- instruction-response pairs

- conversations

- preference data (chosen vs rejected)

B) Labeling + alignment

- human review guidelines
- quality scoring
- remove ambiguous or low-quality examples

C) Train/val/test splits (avoid leakage)

- Split by entity/time so you don’t leak future info.
- Keep a clean held-out test set.

D) Tokenization + packing

- Measure token counts.
- Pack sequences to reduce wasted padding (important at scale).

E) Decontamination (especially for evaluation)

- Ensure eval questions aren’t in training data.

Reference architecture (one simple mental model)

Think in layers:

- **Bronze**: raw extraction snapshots
- **Silver**: normalized + cleaned documents
- **Gold**:

- RAG: chunks + embeddings + indexes

- Training: curated datasets + labels + splits

Minimal checklist (if you're in a hurry)

For a first production RAG ingestion:

- source catalog + ACL mapping
- robust parsing (PDF/HTML)
- boilerplate removal
- dedup
- PII/secrets filtering
- chunking strategy
- embeddings + vector index
- evaluation set + basic retrieval metrics
- incremental sync + deletion handling

Suggested tooling (examples)

Orchestration:

- Airflow, Dagster

Processing:

- Spark, Beam

Document parsing:

- Apache Tika, Unstructured, docling

Search:

- Elasticsearch/OpenSearch (keyword/hybrid)

Vector DBs:

- (many options) pick based on scale, filtering, ops maturity

Storage:

- object storage for raw + normalized layers

Closing

In LLM projects, ingestion is not a side quest. It decides whether your system is:

- helpful or hallucinating
- safe or leaking
- fast or expensive

If you want, tell me your data sources (Confluence? PDFs? Jira? DB?) and whether you're building RAG, fine-tuning, or both — and I'll tailor this pipeline to your exact setup.