KMRL Document Graph — Deep Design

> A deep, practical graph-first design for the KMRL Document Library: nodes, edges, ingestion, retrieval, ranking, snippet-extraction, provenance, security, and an incident walkthrough (engine failure).

---

1. Purpose & design goals

Goal: when an incident or query occurs, automatically find the *exact paragraphs / snippets / records* (and the people who need them) from an ocean of legacy and live documents — across email, SharePoint, Maximo exports, WhatsApp PDFs, scans, and cloud links — with strong provenance and role-aware routing.

Design principles: - **Graph-first**: explicit relationships are first-class (asset → vendor → PO → job card → incident → people). The graph is the connective tissue that tells us *where* to look. - **Hybrid retrieval**: combine graph traversal + keyword search + vector semantic search + reranking for precision and recall. - **Chunk-aware**: treat retrievable units as semantically meaningful chunks (sections, table rows, SOP steps), not fixed-size token windows. - **Provenance & audit**: every snippet must point to original doc, page/section, timestamp, uploader, and checksum. - **Multilingual & multimodal**: support English + Malayalam, OCRed scans, and tables/images. - **Human-in-loop**: always allow verification for high-risk decisions (safety/regulatory).

---

2. High-level architecture (textual)

1. **Connectors**: Email IMAP watcher, SharePoint API, Maximo export pipeline, WhatsApp export parser, SFTP for scanned PDFs, Google Drive crawler.
2. **Pre-processing**: OCR, language detection, layout parsing (headings/tables/figures), chunking, metadata extraction.
3. **NLP Enrichment**: NER (asset IDs, part numbers, station names), classification (doc type), key-value extraction (invoices/POs), date extraction.
4. **Storage**:
5. Raw files in object store (S3-like)
6. Graph DB (Neo4j / JanusGraph) for entities + relationships
7. Text search index (Elasticsearch/OpenSearch)
8. Vector DB (Weaviate / Milvus / Pinecone) for semantic chunks
9. Audit log store
10. **Retrieval Engine**: multi-stage retrieval + reranker + snippet extractor + summarizer (LLM optional)
11. **Workflow & Notification**: rule engine + graph-driven role resolution + notification layer (app push / WhatsApp Business / email / SMS)
12. **UI**: incident page with prioritized docs, snippets, provenance, assign/ack buttons, and 'open original doc'.

---

3. Graph data model (nodes & properties)

Core node types (with suggested properties):

- `Document`
- `doc_id`, `title`, `source`, `file_url`, `file_type`, `language`, `created_at`, `uploaded_by`, `pages`, `checksum`, `version`
- `DocumentChunk` (a retrievable unit)
- `chunk_id`, `doc_id`, `page_range`, `section_title`, `text`, `embedding_id`, `char_count`, `is_table`, `table_metadata`, `language`
- `Incident`
- `incident_id`, `timestamp`, `station`, `train_id`, `severity`, `reported_by`, `status`, `tags`
- `Asset` / `Train`
- `asset_id`, `asset_type` (e.g., traction_motor), `train_id`, `depot`, `current_status`
- `Part`
- `part_id`, `part_number`, `description`, `vendor_id`, `current_stock`
- `Vendor`
- `vendor_id`, `name`, `contact_info`, `contract_reference`
- `Person`
- `person_id`, `name`, `roles` (array), `department`, `contact`
- `JobCard`
- `jobcard_id`, `asset_id`, `issue`, `performed_by`, `start_time`, `end_time`, `status`
- `PurchaseOrder` / `Invoice`
- `po_id` / `invoice_id`, `part_id`, `amount`, `date`, `status`
- `SOP` / `Policy` / `RegulatoryDirective`
- `policy_id`, `title`, `effective_date`, `jurisdiction`
- `SystemSource` (represents where doc originally came from)

- `source_id`, `type` (email, sharepoint, maximo, whatsapp), `details`
- `Location` (Station, Depot)
- `location_id`, `name`, `lat`, `lon`, `type`
- `Project` (for expansion documents)
- `project_id`, `name`, `phase`
- `AuditLog` (immutable event entries)
- `event_id`, `actor`, `action`, `target_node`, `timestamp`, `metadata`

Example node instance (DocumentChunk)

```
{
  "chunk_id": "doc123#p45-47#section-repair-guidelines",
  "doc_id": "doc123",
  "page_range": [45,47],
  "section_title": "Traction Motor - Emergency Restart Procedure",
  "text": "If traction motor overheats, follow steps...",
  "embedding_id": "vec789",
  "is_table": false
}
```

---

4. Edge types (relationship semantics)

Edges capture orthogonal relationships and should be typed & directional when meaningful. Examples:

- `(:Document)-[:mentions]->(:Asset)` — doc mentions asset
- `(:DocumentChunk)-[:part_of]->(:Document)`
- `(:Person)-[:authored]->(:Document)`
- `(:Incident)-[:affects]->(:Asset)`
- `(:Incident)-[:related_doc]->(:Document)` — manually linked
- `(:Asset)-[:has_part]->(:Part)`
- `(:Part)-[:supplied_by]->(:Vendor)`
- `(:JobCard)-[:created_for]->(:Asset)`
- `(:PurchaseOrder)-[:orders]->(:Part)`
- `(:Document)-[:supersedes]->(:Document)`
- `(:Document)-[:references]->(:RegulatoryDirective)`
- `(:Person)-[:role_in]->(:Department)`

Edge properties matter: `confidence`, `extracted_by`, `created_at`, `source_id`. For example, `Document-[:mentions {confidence:0.81, extracted_by:'ner-v2'}]->Asset`.

---

5. Chunking strategy (making long docs retrievable)

Why chunk?: retrieving 1–2 relevant paragraphs from a 100-page doc requires mapping the document to semantically coherent retrieval units.

Chunking rules (practical): 1. **Structure-aware chunking**: parse document into headings/sections using layout analysis. Each section becomes a chunk if ≥N characters. 2. **Page-aware chunking**: if headings are missing, use page boundaries but combine adjacent pages if they contain a continuous section header. 3. **Table-aware chunking**: extract tables and create chunk-per-table + chunk-per-row for invoices/POs (table rows as structured records). 4. **SOP-step chunking**: SOPs often list steps — each step can be a chunk (so step-3 can be directly returned). 5. **OCR confidence thresholding**: if OCR confidence low for a chunk, tag it and optionally surface for human verification. 6. **Chunk metadata**: page_range, section_title, language, doc_type, named_entities, embedding_id, is_table.

Chunk size: adaptively 200–2000 tokens depending on structure. The goal is semantic coherence, not uniform size.

---

6. Ingestion pipeline (detailed)

1. **Connectors** receive files or pointers.
2. **Normalization**: convert to standard PDF/HTML/plaintext; store original in object store with checksum.
3. **Layout & OCR**: run OCR (for scans/WhatsApp images) and layout parser (to detect headings/tables/figures).
4. **Chunking**: apply the chunking rules; produce `DocumentChunk` objects.
5. **Extraction**: NER (assets/parts/stations/persons), KVP extraction for invoices/POs, detect language (Malayalam/English) — use Indic models for Malayalam.
6. **Graph mapping**: create or update nodes: `Document` node, `DocumentChunk` nodes, `mentions` edges, `authored` edges, etc. If entities exist (asset123), link chunks/docs to them.
7. **Indexing**: push chunk text + metadata to Elasticsearch for keyword; create embeddings and store in Vector DB referencing the `chunk_id`.
8. **Audit**: log ingestion event into `AuditLog`.

---

7. Retrieval pipeline (query handling)

Overall idea: multi-stage retrieval that uses graph traversal first to identify context, then text+vector searches to fetch chunks, then rerank + extract snippet.

Step A — Query interpretation

- **Intent detection**: is the user asking for `SOP`, `history`, `vendor status`, `part invoice`?
- **Entity extraction**: station name, train id, asset id, time window.
- **Priority**: severity (emergency vs info) and role of requester (controller vs manager).

Step B — Graph-based context expansion

- Start from extracted entities (e.g., `Asset: traction_motor#T123`, `Station: X`) and **traverse neighbors** up to a small depth (recommended 1–3), collecting:
- Related `Documents` and `DocumentChunks`
- Related `JobCards`, `PurchaseOrders`, `Vendors`
- People who have role connections to this asset or station

Graph traversal is **low-latency** if precomputed neighbor caches exist for high-frequency assets.

Step C — Candidate generation (hybrid)

- **Graph candidates**: document chunks directly linked to the visited nodes.
- **Keyword candidates**: run BM25/ES query scoped by metadata (station, asset_id, doc_type, date range) to return top-N chunks.
- **Semantic candidates**: run vector similarity against the user's query embedding to find semantically similar chunks across corpus.

Merge candidate sets (dedupe by `chunk_id`).

Step D — Reranking

Compute final score per chunk using weighted model:

```
final_score = α * semantic_score + β * bm25_score + γ * graph_score + δ *
recency_boost + ε * doc_type_priority
```

- `graph_score` measures closeness in graph (e.g., 1 / (1 + distance)) and edge confidence.
- `doc_type_priority` boosts SOPs/regulatory for safety queries.
- Set weights (α..ε) tuned via small validation set (MVP) then refined.

Step E — Snippet extraction & summarization

- Use extractive snippet selection: pick the sub-sentence span in the chunk that maximizes overlap with query tokens and named entities.
- Optionally use an LLM to produce a 1–3 line actionable summary (RAG style), but always include the original snippet + link.

Step F — Role-specific packaging

- For each recipient role (Maintenance/Procurement/Safety/R&D) produce a package: summary + top 1–3 chunks + provenance + quick actions (assign, open jobcard, request PO, escalate).

---

8. Graph traversal & scoring details

Traversal strategy: - Use **bidirectional BFS** when searching for related entities between two nodes (e.g., incident → vendor). For neighborhood discovery, use normal BFS with depth-limits. - Prioritize **typed edges** (e.g., `has_part`, `supplied_by`) over loose `mentions` edges.

Graph score computation example: - If chunk linked directly to queried asset: `graph_score = 1.0` - If chunk linked via one intermediate (jobcard): `graph_score = 0.7` - If linked by mention only: `graph_score = 0.3` - Use edge `confidence` multipliers and `edge_age` decay (older connections can get small penalty).

Confidence aggregation: combine extractor confidence (NER), OCR confidence, and edge confidence to provide an overall provenance confidence.

---

9. Example: engine failure incident — end-to-end traversal

Scenario: Incident reported: "Engine failure (traction motor) at Station X at 13:00; reported by StationControllerA."

1) Query parsing: detect entity `station: X`, `asset_type: traction_motor`, `time: 2025-09-15T13:00`.

2) Graph seed nodes: find `Location: Station X`, assets assigned to trains serving X (traverse `Location<-serves-Train->has_asset->Asset`), select `Asset` nodes with type traction_motor.

3) Graph traversal: from each candidate `Asset` find: - `JobCards` where `status` = closed or open in last 12 months - `Documents` linked to those jobcards (inspection reports) - `Parts` that belong to the asset - `PurchaseOrders` and `Invoices` for those parts - `Vendors` supplying those parts

4) Candidate chunks: gather chunks from: - SOPs for traction motor emergency procedures - Most recent jobcard text chunks for that asset - Incident reports for similar failure keywords (last 2 years) - Invoice rows for part numbers matching the asset's parts

5) Ranking: rerank using combined score (semantic + bm25 + graph proximity + recency)

6) Output: for Maintenance: top chunk = SOP step with exact step number + link to full SOP (page 45–47); for Procurement: top chunk = invoice row + PO link; for R&D: top chunks = list of previous incidents + aggregated frequency stats.

7) Notifications: send packages to `MaintenanceOnCall`, `ProcurementLead`, `SafetyOfficer` with tailored content.

Cypher sketch (Neo4j) to find docs linked to asset (simplified):

```
MATCH (a:Asset {asset_id: 'T123'})<-[:has_part]-(p:Part)
OPTIONAL MATCH (jc:JobCard)-[:created_for]->(a)
OPTIONAL MATCH (d:Document)-[:mentions]->(a)
RETURN a, collect(distinct jc) as jobcards, collect(distinct d) as documents
LIMIT 200;
```

---

10. Integrating full-text + vector search with graph

Common pattern: 1. Graph traversal returns entity-scoped filters (asset ids, part ids, doc types, date ranges). Use these as metadata filters in Elasticsearch and vector search. 2. Run ES query with `must` clauses on metadata (station, asset_id) and `should` on free text. Pull top ~50 chunks. 3. Run vector similarity on the user query to pull top ~50 semantic chunks globally. 4. Merge & dedupe candidate chunk IDs and compute rerank scores.

This reduces false positives from purely semantic or purely keyword-based retrieval.

---

11. LLM fine-tune & role in the system

Use-cases for LLMs: - Short summarization (1–3 lines) of multi-chunk evidence (RAG) - Extractive Q/A over one chunk to find exact procedure line - Reranking candidate chunks with learned relevance model - Mapping free-language queries into structured queries/entities

Fine-tuning on past data: - Create a labeled dataset: (query, relevant_chunk_ids, role, gold-summary) - Fine-tune a small-to-medium LLM for reranking and summarization. Advantage: better domain-adapted prioritization. Risk: hallucinations → always present original snippet and confidence.

Alternative: use open-source rerankers (cross-encoders) for reranking and a small LLM for summarization with RAG and provenance.

---

12. Live ingestion & continuous graph growth

- **Streaming ingestion** via Kafka: connectors emit ingestion events into topics; processors parse and update graph & indexes.
- **Near-real-time updates:** new documents create chunks, embeddings, graph edges; triggers notify watchers if they exist for a related open incident.
- **Backfill pass:** offline job that processes historical corpus and populates graph; store `ingestion_batch` id to allow safe re-processing.

Idempotency: use file checksum + source_id to avoid duplicates; maintain `Document.source_id` mapping.

---

13. Security, RBAC & compliance

- **Attribute-based access control (ABAC)** on nodes: roles, departments, clearance levels. Enforce at query time (filter out nodes user can't see).
- **Field-level masking** for PII in public roles.
- **Immutable audit logs** for any read of regulatory documents (for compliance tracking).
- **Encryption** at rest and in transit; key management.

---

14. Monitoring & metrics

- **Precision@k** for retrieval (manual QA labeling)
- **Time-to-first-useful-snippet** (MTTR improvement)
- **Ingestion lag** (time from doc creation to searchable)
- **Coverage** (percent of incidents for which system found a relevant doc)
- **Number of escalations avoided** (operational metric)

---

15. Scalability & operational choices

- **Graph DB choices**: Neo4j (developer-friendly, great query language + visualization), JanusGraph (scales with Cassandra/HBase), Amazon Neptune (managed). For prototype Neo4j is easiest.
- **Vector DB**: Milvus/Weaviate for OSS; Pinecone for managed.
- **Search**: Elasticsearch or OpenSearch.
- **Event bus**: Kafka or RabbitMQ.
- **Storage**: S3-compatible object store for raw files.

Scaling tips: - Denormalize frequently-needed neighbor lists (materialized views). - Precompute embeddings for chunks. - Shard vector DB by namespace (department/year) if needed.

---

16. MVP blueprint (priorities)

Goal: demonstrate incident→people→docs flow with precise snippets and traceability.

MVP scope: 1. Backfill 200 sample documents (mix of PDFs, scanned images, jobcards, SOPs) 2. Implement connectors for email + folder share (manual upload UI) 3. Implement chunking + ES + Milvus + small Neo4j instance 4. Implement query pipeline (graph seed → hybrid search → rerank → snippet) and simple UI 5. Implement notifications via email/console

Success criteria: controllers test with 10 incidents and find top-1 snippet relevant ≥ 80% of the time.

---

17. Risks & mitigations

- **OCR / Indic language errors**: use specialized OCR engines, human verification for critical docs.
- **Hallucination by LLM**: never present LLM-only statements without original snippet and link.
- **Data duplication & drift**: use checksums & versioning; periodic re-ingestion for schema drift.
- **Complex graph growth**: limit traversal depth and precompute high-degree neighbor caches.

---

18. Appendix: sample Cypher snippets

Find doc chunks mentioning an asset and return with page ranges:

```
MATCH (a:Asset {asset_id:'T123'})
MATCH (c:DocumentChunk)-[:part_of]->(d:Document)
```

```
WHERE (d)-[:mentions]->(a)
RETURN d.title, c.chunk_id, c.section_title, c.page_range LIMIT 50;
```

Get recent jobcards & linked docs for an asset in last 1 year:

```
MATCH (a:Asset {asset_id:'T123'})<-[:created_for]-(jc:JobCard)
WHERE jc.start_time >= date() - duration({days:365})
OPTIONAL MATCH (jc)-[:references]->(d:Document)
RETURN jc.jobcard_id, jc.issue, collect(d.title) as documents
ORDER BY jc.start_time DESC LIMIT 50;
```

Traverse incident->assets->vendors (2 hops):

```
MATCH (i:Incident {incident_id:'INC-2025-001'})-[:affects]->(a:Asset)-
[:has_part]->(p:Part)-[:supplied_by]->(v:Vendor)
RETURN i.incident_id, a.asset_id, p.part_number, v.name LIMIT 100;
```

---

19. Next steps (recommended)

1. Build a small prototype: 200 sample docs, Neo4j + ES + Milvus, ingestion + chunker.
2. Create a small labeled validation set of queries & gold chunks to tune weights.
3. Implement role-specific packaging and a minimal incident UI.

If you want, I can now: - produce **detailed Cypher queries** for the core traversals used in the engine-failure example, or - write **pseudocode for the ingestion pipeline** (connectors → chunker → graph updates → indexing), or - draft a **simple incident UI wireframe** and API contract to power it.

Pick one and I'll generate it next.

End of document.