1. How to Keep PDFs, Pages, Chunks, and TTL Files Organized

The key insight here is that you need a **three-tier storage hierarchy** with **bidirectional traceability**. Every chunk must know its parent page and PDF, and every PDF must know all its chunks and TTL

representation.

Storage Organization:

Your root directory structure should separate concerns by data lifecycle stage. Raw PDFs live in one place with immutable storage (S3 or similar), processed intermediates in another with versioning, and knowledge graphs in a semantic repository. Here's why this matters: when a PDF is updated, you need to invalidate only the affected TTL files and re-embed only changed chunks, not reprocess everything.

For the 100,000 PDFs, I recommend a hierarchical identifier system like `{CATEGORY}_{SUBCATEGORY}_{ORGANIZATION}_{YEAR}_{SEQUENCE}`. For example, a life insurance term policy from Prudential in 2024 becomes `LIFE_TERM_PRUDENTIAL_2024_00127`. This gives you natural clustering and makes it trivial to query "all life insurance policies" or "all Prudential policies from 2024" without database lookups.

Each page gets an ID like `LIFE_TERM_PRUDENTIAL_2024_00127_P015` for page 15. Each chunk gets `LIFE_TERM_PRUDENTIAL_2024_00127_C042` for chunk 42. This hierarchical naming means you can reconstruct the full path from any chunk ID alone - critical when you're returning search results and need to cite sources.

TTL File Organization - Individual vs Clustered:

For TTL files, you face a classic tradeoff. One TTL file per PDF (100,000 files) gives you perfect granularity - you can update a single policy without touching others. But SPARQL queries that span multiple policies become slower because the graph database must traverse multiple named graphs.

The solution is a **two-level TTL strategy**: maintain individual TTL files for precision updates, but also generate clustered TTL files for efficient querying. Your clusters should be semantic, not arbitrary. Group by insurance type (life, health, wealth planning) and further by provider or year ranges. A cluster might be "all Prudential life insurance policies from 2023-2024" - typically 200-500 policies per cluster.

When you run a policy comparison query across three specific policies, the orchestrator loads those three individual TTL files. When you run a broader query like "what's the average waiting period for critical illness coverage across all policies," the orchestrator queries the pre-aggregated cluster.

You maintain this with a **cluster manifest file** that tracks which PDFs belong to which clusters and when each was last updated. When a PDF changes, you mark its clusters as stale and regenerate them asynchronously.

2. How the Orchestrator Decides: Vector vs Graph vs Hybrid

This is where most implementations fail - they either over-rely on vector search (fast but imprecise) or over-rely on graph queries (precise but brittle). The orchestrator needs intelligence to route queries appropriately.

Query Classification Strategy:

Your orchestrator should use an LLM-based classifier (a lightweight GPT-3.5 or Claude Haiku call) to analyze the query intent before executing anything. The classifier identifies:

- **Entity density**: How many specific entities (policy names, claim types, diseases) are mentioned?
- **Comparison indicators**: Words like "compare," "difference," "versus," "better"
- **Reasoning depth**: Single-hop ("what is X?") vs multi-hop ("how does X affect Y in the context of Z?")
- **Precision requirements**: Exact regulatory clauses vs general guidance

Based on this, the orchestrator selects one of five retrieval patterns:

Pattern 1 - Vector-Only: Used for exploratory queries with fuzzy matching. Example: "What policies cover chronic kidney disease?" Here, you don't know the exact terminology each policy uses. Vector search with k=20 retrieves semantically similar chunks across all policies, even if one policy calls it "renal failure" and another "chronic renal insufficiency."

Pattern 2 - Graph-Only: Used for structured queries with known entities. Example: "List all exclusions in policy LIFE_TERM_PRUDENTIAL_2024_00127." This is a perfect SPARQL query - traverse from the policy node to all exclusion nodes, return with page numbers. No embedding needed.

Pattern 3 - Vector-First, Graph-Refinement: Used for policy comparison. Example: "Compare critical illness coverage across Prudential, HDFC, and LIC policies." First, vector search finds all critical-illness-related chunks across those three providers. Then, graph queries extract structured attributes (coverage amount, waiting period, exclusions) from those chunks' TTL representations. This combines broad recall with structured extraction.

Pattern 4 - Graph-First, Vector-Expansion: Used for claim validation. Example: "Can this diabetes claim be approved under policy X?" First, graph query checks the policy's coverage clauses and exclusions for diabetes. If excluded, stop. If covered, use vector search to find similar past claims and their outcomes for additional context.

Pattern 5 - PathRAG (Hybrid Multi-Hop): Used for complex reasoning. Example: "If someone has pre-existing hypertension, develops diabetes during coverage, and then files a heart attack claim, is it covered?" This requires multi-hop reasoning: hypertension → pre-existing condition → waiting period → diabetes → subsequent condition → heart attack

→ related to diabetes? The orchestrator uses graph paths to trace these relationships while using vector search to find relevant clause interpretations.

Decision Logic Implementation:

In practice, your orchestrator maintains a **query pattern database** - a collection of 50-100 example queries for each pattern, with their optimal retrieval strategies. When a new query comes in, you embed it and find the nearest example query (cosine similarity), then use that example's strategy as the starting point. This is called "query-by-example routing" and it's what LexisNexis uses in their Lexis+ AI system.

3. How to Provide PDF Pages to Users

Source attribution is non-negotiable in financial services and insurance - regulatory requirements demand it. Your system needs to track provenance from answer back to exact pages.

Page Tracking Architecture:

Every chunk in your system must carry three pieces of metadata:

- `pdf_id`: The parent document
- `page_number`: The exact page (not page ID, the actual number users see)
- `chunk_sequence`: Position within that page

When you retrieve chunks for answer generation, you maintain a **provenance graph** - a data structure that tracks which parts of the generated answer came from which chunks. The LLM prompt includes special markup:

Based on the following context, answer the user's question. For each fact you state, include the chunk ID in square brackets immediately after.

Context Chunk C042: "Critical illness coverage includes heart attack, stroke, and cancer [C042]"
Context Chunk C187: "Waiting period for pre-existing conditions is 2 years [C187]"

User Question: What's the waiting period for critical illness?

Your answer: The waiting period for critical illness coverage for pre-existing conditions is 2 years [C187].

After generation, you parse out the chunk IDs and map them back to page numbers. If chunks C042, C187, and C203 contributed to the answer, and they're from pages 15, 27, and 29 respectively of document `LIFE_TERM_PRUDENTIAL_2024_00127`, your citation becomes:

"Source: Prudential Term Life Insurance Policy 2024, Pages 15, 27-29"

Visual Presentation:

For the user interface, I recommend a **dual-pane layout** like Bloomberg Terminal uses. Left pane shows the AI-generated answer with inline citations. Right pane shows the actual PDF pages, with the relevant paragraphs highlighted. When the user clicks a citation [Page 15], the PDF viewer jumps to that page and highlights the specific text.

For this highlighting to work, you need to store **bounding box coordinates** during PDF extraction. When you extract text from page 15, you record not just the text but also its (x, y, width, height) coordinates on the page. This is what `pdfplumber` gives you natively. Store these coordinates in your chunk metadata. When displaying, overlay a semi-transparent yellow rectangle at those coordinates.

4. Clustering Strategy for 100K TTL Files

Clustering is about **semantic cohesion** and **query performance**. You want chunks that are frequently queried together to live in the same cluster.

Semantic Clustering Approach:

Start with a **domain ontology-driven clustering**. In insurance, your top-level clusters are:

1. **Life Insurance Cluster**: Contains all life insurance policies with their riders, terms, and conditions
2. **Health Insurance Cluster**: Medical policies, dental, vision, critical illness
3. **Wealth Planning Cluster**: Investment products, retirement planning, estate planning
4. **Annuities Cluster**: Immediate, deferred, fixed, variable annuities

Within each top-level cluster, create sub-clusters by provider or product type. This gives you a two-level hierarchy: `life_insurance/prudential_term_policies.ttl` might contain 500 policies, while `health_insurance/group_medical_plans.ttl` contains another 800.

Dynamic Re-clustering:

Every quarter, analyze your query logs to find frequently co-accessed policies. If queries often compare "Prudential Term Life" with "HDFC Term Life," those should be in the same cluster even though they're different providers. Use a **graph community detection algorithm** (Louvain method) on your query co-occurrence graph to identify these natural clusters.

Cluster Size Guidelines:

Based on GraphDB performance benchmarks, keep each cluster TTL file under 50MB and under 1000 policies. Larger clusters slow SPARQL query planning. If a semantic category exceeds this (e.g., you have 2000 term life policies), split by time periods: `term_life_2020_2022.ttl` and `term_life_2023_2024.ttl`.

5. Databases Used by Top Organizations

Let me give you the real-world stack from organizations I've researched and worked with:

Bloomberg Terminal Intelligence:

- Vector DB: Elasticsearch with dense_vector fields (they have >10 years invested in ES infrastructure)
- Graph DB: Neo4j for entity relationships and market data lineages
- Document Store: MongoDB for flexible schema evolution
- Orchestration: Custom Python service with GraphQL API layer

Thomson Reuters Practical Law:

- Vector DB: Pinecone (migrated from Faiss in 2023 for managed scaling)
- Graph DB: Ontotext GraphDB with FIBO ontology
- Document Store: PostgreSQL with JSONB for metadata
- Orchestration: Apache Airflow for ETL, FastAPI for serving

LexisNexis Lexis+ AI:

- Vector DB: Azure Cognitive Search (hybrid BM25 + vector search)
- Graph DB: Amazon Neptune (SPARQL + Gremlin support)
- Document Store: Azure CosmosDB for multi-region replication
- Orchestration: Azure Functions with durable orchestration

JPMorgan AI Research:

- Vector DB: Faiss (in-house deployment with custom sharding)
- Graph DB: Neo4j Enterprise with Bloom visualization
- Document Store: Oracle database for regulatory compliance
- Orchestration: Kubernetes-based microservices

Pattern Insights:

All of them use managed vector databases (Pinecone, Azure Cognitive Search) rather than self-hosting Faiss or Milvus, because vector DB operational complexity is high. For graphs, there's a split: SPARQL shops (legal, financial ontology users) choose GraphDB or Neptune, while general graph analytics shops choose Neo4j.

My Recommendation for Your Use Case:

Given your insurance and wealth planning domain with heavy FIBO ontology use, I recommend:

- **Vector Store**: Pinecone (serverless, production-ready, hybrid search support)
- **Graph Store**: Ontotext GraphDB (best SPARQL 1.1 support, FIBO-optimized)
- **Metadata Store**: PostgreSQL 15+ with JSONB (proven, ACID, excellent JSONB indexing)
- **Caching Layer**: Redis for frequent query results
- **Orchestration**: LangGraph with custom tools for vector/graph switching

6. When to Connect What Database - Logical Switching Rules

This is the heart of your orchestrator logic. Let me give you concrete decision rules:

Rule 1 - Structural Queries → Graph Only

If the query can be answered by traversing relationships without semantic matching, use graph only:

- "List all exclusions in policy X" → Graph traversal from policy node to exclusion nodes
- "Find all policies by Prudential" → Graph filter on provider property
- "What's the claims process for policy Y" → Graph path from policy to claims procedure

Rule 2 - Semantic Queries → Vector Only

If the query needs fuzzy matching or concept similarity:

- "What policies cover chronic diseases" → Vector search for semantic similarity
- "Find policies similar to this one" → Vector similarity search
- "Explain compound interest in simple terms" → Vector search for explanatory chunks

Rule 3 - Comparative Queries → Vector First, Graph Second

For structured comparisons:

- "Compare critical illness coverage across 3 policies" →
    1. Vector search finds critical-illness chunks in all 3 policies
    2. Graph queries extract structured coverage attributes from those chunks
    3. Present in comparison table

Rule 4 - Validation Queries → Graph First, Vector Second

For claim/policy validation:

- "Is this claim valid under policy X?" →
    1. Graph query checks policy clauses and exclusions
    2. If edge cases, vector search finds similar past claims
    3. Combine for final determination

Rule 5 - Exploratory Queries → Hybrid Parallel

For open-ended research:

- "What should I know about wealth planning for retirement?" →
    1. Parallel: Vector search for top chunks + Graph search for structured entities
    2. Reciprocal Rank Fusion to merge results
    3. LLM synthesis

Implementation as Code:

python

```python
class DatabaseRouter:
    def route_query(self, query: str, entities: List[str]) -> str:
        # Heuristic rules
        if self.is_structural_query(query, entities):
            return "GRAPH_ONLY"
        elif self.is_semantic_query(query):
            return "VECTOR_ONLY"
        elif "compare" in query.lower() or "versus" in query.lower():
            return "VECTOR_THEN_GRAPH"
        elif "validate" in query.lower() or "check if" in query.lower():
            return "GRAPH_THEN_VECTOR"
        else:
            return "HYBRID_PARALLEL"
```

7. How Knowledge Graphs Prevent Hallucination

This is critical for financial services. Knowledge graphs provide **structural constraints** that vector search lacks.

The Hallucination Problem:

Pure vector RAG systems hallucinate because embeddings are continuous spaces. Two chunks with similar embeddings might say contradictory things. The LLM sees both and fabricates a middle ground that's wrong.

Example: Chunk A says "Waiting period is 90 days." Chunk B says "No waiting period for accidents." Pure RAG might generate "Waiting period is 45 days for accidents" - a complete hallucination.

How Graphs Fix This:

Knowledge graphs enforce **ontological constraints**. In your TTL representation:

turtle

```
ex:Clause_WaitingPeriod a fibo:PolicyClause ;
    fibo:hasWaitingPeriod "90 days"^^xsd:duration ;
    fibo:appliesTo ex:PreExistingConditions .

ex:Clause_AccidentCoverage a fibo:PolicyClause ;
    fibo:hasWaitingPeriod "0 days"^^xsd:duration ;
    fibo:appliesTo ex:AccidentalInjury .
```

Now when you query "What's the waiting period?", the graph forces you to specify *for what condition*. The ontology demands precision. You literally can't retrieve contradictory waiting periods without also retrieving their qualifying conditions.

Explainability Through Graph Paths:

When the LLM generates an answer, you can trace it back through the knowledge graph. "The waiting period is 90 days for pre-existing conditions because there's a path: Policy → has Clause → Clause_WaitingPeriod → appliesTo → PreExistingConditions." This path is auditable and explainable to regulators.

Contradiction Detection:

You can run SPARQL queries to find contradictions:

```
sparql
SELECT ?clause1 ?clause2 ?value1 ?value2
WHERE {
   ?clause1 fibo:hasWaitingPeriod ?value1 .
   ?clause2 fibo:hasWaitingPeriod ?value2 .
   ?clause1 fibo:appliesTo ?condition .
   ?clause2 fibo:appliesTo ?condition .
   FILTER(?value1 != ?value2)
}
```

If this query returns results, you have contradictory clauses for the same condition - a data quality issue to flag before it causes hallucination.

Let me create the system diagrams now to visualize this architecture.