Name: Priyadharshan

**RAG AGENT**

* Retrieval Augmented Generation
* ETA – Estimated Time of Arrival

**1. Collect Data**

* Gather logistics-related documents: shipment records, warehouse SOPs, delivery partner manuals, GPS/IoT data, customer FAQs.

**2. Preprocess & Chunk**

* Break reports and manuals into smaller chunks (e.g., per section or per route info).

**3. Convert to Embeddings**

* Turn each chunk into embeddings (numerical meaning representation).  
   *Example:* “Average truck speed in Chennai route is 40 km/h” → stored as a vector.

**4. Store in a Vector Database**

* Save embeddings in Chroma, FAISS, or Pinecone for semantic search.

**5. Retrieval Step**

* When someone asks: “Why are deliveries late in Monsoon season?”,
* The system searches DB → pulls chunks mentioning “rain”, “traffic jams”, “port delays”.

**6. Augment the Prompt**

* Combine retrieved logistics info + the user’s query.

***Example:***

Context: Deliveries on Chennai–Bangalore route slow down in monsoon due to flooding and heavy traffic.

**Question: Why is ETA longer in monsoon season?**

LLM → “ETAs are longer due to weather-related traffic delays and reduced truck speeds.”

**7. Add Agentic Layer**

* **The agent can use different tools:**
  + Retriever → find logistics SOPs & reports
  + Calculator → compute ETA, average delivery time
  + API/Web → fetch live traffic/weather data

 Ask: “What is ETA for a truck from Chennai to Coimbatore at 60 km/h?” → agent uses **calculator tool**.

 Ask: “What causes delays in port shipments?” → agent uses **retriever (docs)**.

 Ask: “What’s today’s traffic status?” → agent uses **web/live API tool**.

**CHUNKING:**

**What**

Chunking = cutting big documents into small pieces (chunks) before indexing/searching.

**Why**

* LLMs & vector search work best on short, focused text.
* Small chunks = better matches and less hallucination.
* Overlap keeps context that would otherwise get cut.

**How (simple rule of thumb)**

* Size: 300–800 tokens (≈ 400–1200 chars)
* Overlap: 50–150 chars between chunks
* Keep metadata: source, title, page/section → for citations

Mini example (feel it once)

DS18B20 is a 1-Wire temperature sensor (−55°C to +125°C) needing a pull-up on data.

DHT22 measures temperature + humidity (±0.5°C, ±2–5% RH).

On ESP32 use OneWire for DS18B20 and DHT library for DHT22. Long cables need proper pull-up/shielding.

**Split into ~120-char chunks with 30 overlap:**

* **Chunk 1**  
  DS18B20 is a 1-Wire temperature sensor (−55°C to +125°C) needing a pull-up on data. DHT22 measures temperature +
* **Chunk 2**  
  + humidity (±0.5°C, ±2–5% RH). On ESP32 use OneWire for DS18B20 and DHT library for DHT22. Long cables
* **Chunk 3**  
  bles need proper pull-up/shielding.

**CHUNKING — TYPES, WHEN TO USE, LOGISTICS EXAMPLES**

**A) Fixed-size (chars/tokens) with overlap**

* **What:** cut every N tokens (e.g., 800) with 100–150 token overlap.
* **Pros:** simple, fast, consistent.
* **Cons:** can split mid-sentence/clause.
* **Use when:** quick prototype; mixed prose.
* **Logistics ex:** split a 30-page **operations SOP** into uniform chunks.

**B) Sentence/paragraph boundary chunking**

* **What:** split on sentence/paragraph, then merge until size limit.
* **Pros:** semantic integrity; fewer “half ideas”.
* **Cons:** uneven sizes; still may be large.
* **Use when:** manuals, FAQs, knowledge bases.
* **Logistics ex:** **returns policy** or **warehouse safety** doc per paragraph.

**C) Heading/section (clause-aware) chunking**

* **What:** split on headings like 1., 1.1, Clause:, Penalty.
* **Pros:** perfect for **contracts**; keeps legal meaning intact.
* **Cons:** needs pattern rules for messy PDFs.
* **Use when:** **carrier contracts**, **SLAs**, **Incoterms** summaries.
* **Logistics ex:** “Late delivery penalties”, “Liability”, “Force majeure” as distinct chunks.

**D) Recursive/hierarchical splitter (best general)**

* **What:** try big (by heading) → if too big, split by paragraph → then sentence → then fixed size.
* **Pros:** balanced, robust to messy docs.
* **Cons:** slightly more engineering.
* **Use when:** most real projects; mixed quality PDFs.

**E) Structure-aware (tables, forms, code)**

* **What:** detect **tables** or key-value **form fields**; chunk by **row** or **logical block**.
* **Pros:** preserves relations; great for extraction.
* **Cons:** needs table/form detection (OCR/coords).
* **Use when:** **invoices**, **bills of lading**, **HS code tables**.
* **Logistics ex:** one chunk = “Invoice 1034: buyer, seller, due date, line items”.

**F) Sliding window / n-gram “passage” chunks**

* **What:** overlapping windows slide through sentences.
* **Pros:** maximizes recall for short queries.
* **Cons:** more storage; duplicates.
* **Use when:** search must not miss tiny facts (e.g., HS codes).

**EMBEDDING:**

**1. What is Embedding?**

In simple words:

* Embedding = converting **text into numbers (vectors)** so that a computer/LLM can understand and compare meanings.
* These numbers (vectors) live in a **vector database** where "similar meanings" are stored close together.

**🔹 2. Why Embedding in Your Topic?**

Let’s say you have **100 DHL contracts (PDFs)**.  
Now you want to ask:  
*“What’s the late delivery penalty in DHL’s June 2025 contract?”*

* The system can’t directly read PDF text as words.
* So first, we **chunk** it (e.g., 200 words with overlap).
* Then we **embed each chunk** → convert into vectors.
* When you query, *“late delivery penalty”*, the query is also embedded into a vector.
* The system searches which chunk vector is **closest in meaning** → retrieves that contract clause.

**3. Types of Embeddings**

There are **different kinds of embeddings** depending on what meaning you want to capture:

**(A) Text Embeddings (most common for RAG)**

* Convert **sentences/paragraphs** into vectors.
* Used for **searching meaning in documents**.
* Example in logistics:
  + *“late delivery penalty”* → becomes a vector.
  + The contract sentence *“If goods are delivered late, 5% of invoice value will be charged”* → becomes another vector.
  + Since both are close in meaning, the system retrieves it.

Models: **OpenAI text-embedding-3-large**, **Sentence-BERT**, **Cohere embeddings**.

**(B) Word Embeddings**

* Each **word** gets a vector.
* Useful if you want to analyze keywords across many contracts.
* Example:
  + “Penalty” and “Fine” will be **close in vector space**.
  + So the system can recognize they mean the same thing, even if different contracts use different words.

Older but classic models: **Word2Vec, GloVe**.

**(C) Document Embeddings**

* Create **one embedding for the whole document**.
* Faster, but less detailed (you lose fine-grained info).
* Example:
  + You embed an **entire DHL contract** as one vector.
  + You can then quickly compare **which contracts are similar** (e.g., DHL vs FedEx).
* But → not good for pinpointing *exact clauses*, which is why **chunk + text embedding** is better for your RAG project.

**(D) Multimodal Embeddings (advanced)**

* Convert not just text but **text + images/tables** into embeddings.
* Very useful in logistics → because many invoices/customs forms have **tables, scanned stamps, logos**.
* Example:
  + A scanned **invoice PDF** is converted into embeddings of text + image fields.
  + Then you can query: *“Find all invoices pending from June 2025”* → and it works even if they were scanned copies.

Models: **OpenAI CLIP, LLaVA, Gemini multimodal**.

**PROBLEM STATEMENT:**

Unstructured Documents & Contracts Problem:

Logistics is filled with invoices, customs forms, bills of lading, and carrier contracts — often in PDF/scan form. Impact: Manual lookup is slow, and mistakes cost money (penalties, wrong customs declaration). Solution with RAG: Chunk contracts/invoices → store embeddings → agent answers queries like: “What’s the late delivery penalty in this DHL contract?” “Which invoices are pending payment for June 2025?”

**1. Why Hybrid Chunking?**

Different documents have different **structures**:

* **Contracts** → long text, divided into clauses, headings, sub-sections.
* **Invoices/Customs Forms** → structured data (tables, key-value pairs).

If you use **only clause-aware chunking**, you’ll lose table meaning.  
If you use **only structure-aware (table parsing)**, you’ll lose contract clause details.

So, hybrid chunking = **best of both worlds**.

**🔹 2. Hybrid Chunking Workflow**

Here’s how you could design it:

**(A) Pre-processing Stage**

1. Upload PDF (contract, invoice, bill of lading).
2. Detect the **document type** (contract vs invoice).
   * Use a classifier (simple ML model or rules like keyword “Invoice”, “Bill of Lading”).

**(B) Chunking Stage**

1. **Clause-Aware Chunking (Contracts)**
   * Split based on **headings** (like “Section 1: Payment Terms”, “Section 2: Delivery Timeline”).
   * Maintain **clause boundaries** (so you don’t break a sentence like “Penalty of 5% per week”).

Example chunk:

Section 3: Penalties

If delivery is delayed, a penalty of 5% of invoice value per week applies.

1. **Structure-Aware Chunking (Invoices/Tables)**
   * Parse tables into **row-wise or key-value chunks**.
   * Example invoice:
   * Item | Quantity | Price
   * Laptop | 10 | $500

Becomes → embeddings like:

* + “Laptop, quantity 10, price $500”

**(C) Embedding Stage**

* For **text chunks** → use a **text embedding model** (e.g., OpenAI text-embedding-3-large).
* For **tables/numeric data** →
  + Option 1: Flatten into natural language (“Invoice #1234: pending payment of $5,000”).
  + Option 2: Use a **multimodal embedding model** (like OpenAI CLIP or Gemini) if scanned with tables/images.

**(D) Store in Vector DB**

* Store **both embeddings** together in the same vector DB.
* Add **metadata tags** like:
  + doc\_type=contract or doc\_type=invoice
  + chunk\_type=clause or chunk\_type=table

**(E) Retrieval Stage**

When you query → the system:

1. Embeds your question.
2. Searches both **clause chunks** and **table chunks**.
3. Returns the most relevant ones.

**🔹 3. Example in Your Case**

**Query:**  
 “What’s the late delivery penalty in the DHL June 2025 contract?”

* System searches **clause-aware chunks** → retrieves penalty clause.

**Query:**  
“Which invoices are pending payment for June 2025?”

* System searches **structure-aware chunks** → retrieves invoice table rows with pending status.

**4. Is It Possible?**

Yes, many advanced RAG pipelines already do this in research & startups.  
This is called **“multi-strategy chunking”** or **“hybrid chunking + embedding”**.  
Some even call it **“structured RAG”**.

Tools that support this:

* **LangChain** (Python framework) → lets you define custom chunkers.
* **LlamaIndex** → has strong support for structured docs (tables, SQL, JSON).

**PARCEL LIFE CYCLE**

* **Order and Booking:** The Customer books the product through online and this order was packed, labelled and hand overed to the courier partner
* **First Mile Pickup:** The Courier partner collects the parcel from the senders location and shift it to the nearest hub.
* **Local Sorting:** The orders are shorted based on the delivery PIN code and the address. Here the product is scanned and the customer got the update. Then it was sent to the **Regional Distribution Centre.**
* **Transportation:** The product is transported to the central hub which is the long journey via flight or train or truck.
* **Nearest Hub Arrival:** Once the product got arrived the product is again scanned for status updating. It gets sorted by the delivery zones.
* **Distribution:** once sorted the parcel is assigned for the individual delivery agent and the product is delivered.
* **Reverse Logistics:** If the product is returned the process will happened in the reverse order.

**API (Application Programming Interface)**

A set of rules that allows two software systems to talk to each other and share data or services.

**API Key**

A unique secret code used to identify and authenticate the user or application that is calling an API.