CSV CHUNKING OPTIMIZER

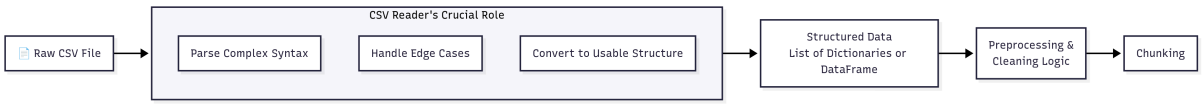
CSV CHUNKING OPTIMIZER DOCUMENT

This document outlines a comprehensive architecture for optimizing CSV processing, chunking, and retrieval for advanced RAG (Retrieval-Augmented Generation) systems. It covers the entire pipeline from initial data ingestion to search and monitoring.

Table of Contents

Each section contains detailed diagrams, explanations, and implementation guidance to build a robust CSV-powered semantic search system.

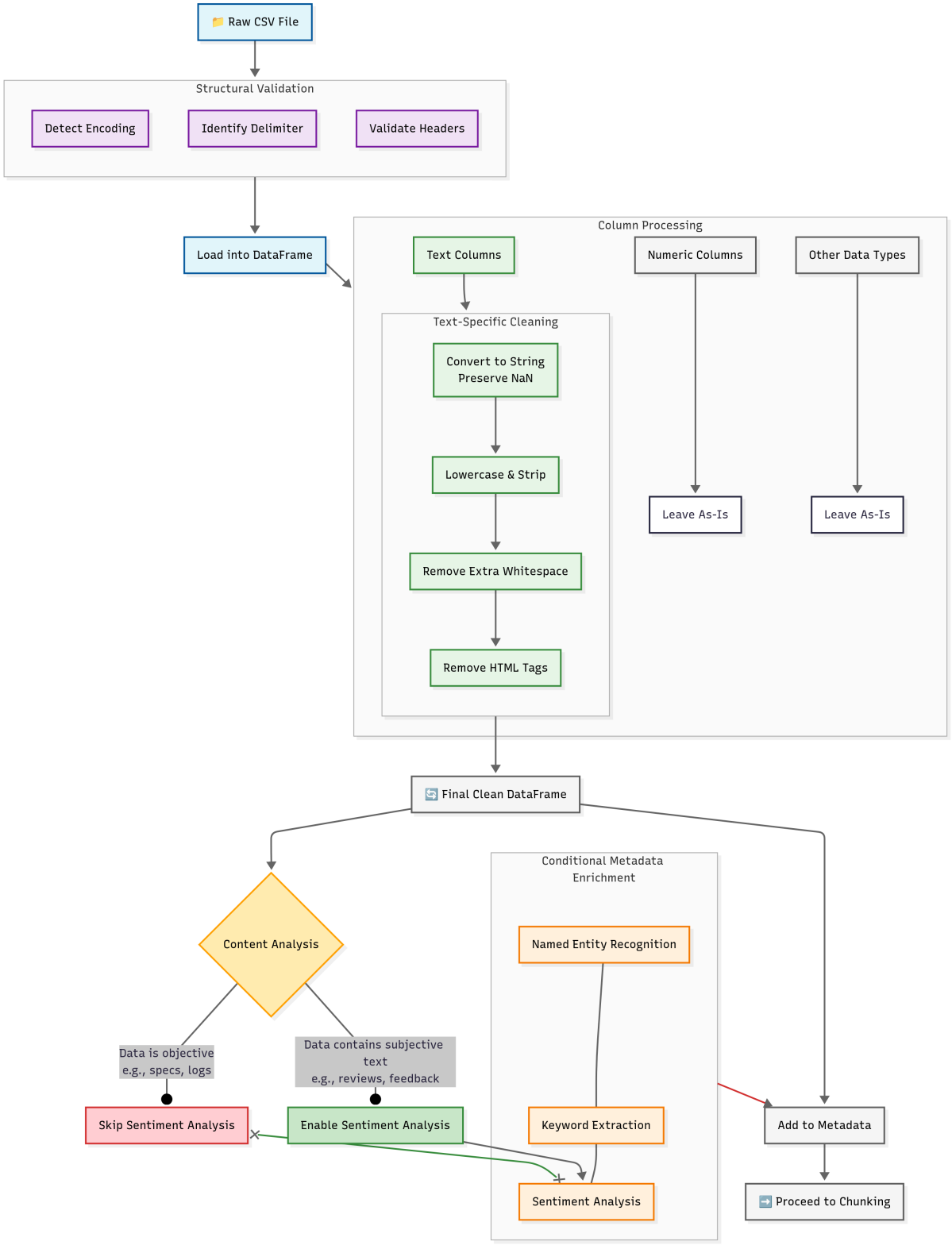
1. CSV Reader



Why csv reader is useful?

Problem	Naive split(',')	Dedicated CSV Reader
Commas in valuese.g., "Smith, John",30,London	['"Smith', ' John"', '30', 'London'] → BROKEN	['Smith, John', '30', 'London'] → CORRECT
Different delimiterse.g., id;name;price (semicolon)	['id;name;price'] → BROKEN	['id', 'name', 'price'] → CORRECT
Line breaks in values(multi-line addresses)	Treats as a new row → BROKEN	Handles as a single value → CORRECT
Quotation marksto escape commas	Doesn't understand quotes → BROKEN	Correctly handles quotes → CORRECT
Headers	You have to manually manage them.	Can automatically detect and use them as dictionary keys.
Data Type Conversion	Everything is a string.	Can help with automatic conversion to numbers, dates, etc

2. Preprocessing Layer



Preprocessing Layer Workflow

1. Structural Validation

Purpose: Ensure CSV files are readable and structured correctly before processing.

- Detect Encoding: Identify character encoding (UTF-8, ASCII) to prevent text corruption
- Identify Delimiter: Determine if file uses commas, semicolons, or tabs

- Validate Headers: Check if first row contains valid column names

Why it's important: This foundational step ensures reliable data loading. If this fails, everything else fails.

2. Column Type Processing & Text Cleaning

Purpose: Clean and standardize text data for semantic search while preserving other data types.

- Column Type Processing:
  - Text Columns: Sent through text cleaning pipeline for embedding
  - Numeric & Other Columns: Preserved as metadata (e.g., price: 499, rating: 4.5)
- Text-Specific Cleaning:
  - Convert to String, Preserve NaN: Maintain text format while respecting missing values
  - Lowercase & Strip: Standardize text and remove extra spaces
  - Remove Extra Whitespace: Collapse multiple spaces/tabs/newlines
  - Remove HTML Tags: Strip HTML/XML code from exported data

Why it's important: Creates clean, consistent text for AI embedding, leading to higher quality search results.

3. Conditional Sentiment Analysis

Purpose: Intelligently determine whether to perform sentiment analysis based on content type.

- Content Analysis: System analyzes cleaned text using heuristics (e.g., TextBlob)
- Decision Pathways:
  - Enable: For subjective text (reviews, feedback, support tickets)
  - Skip: For objective text (product specs, logs, financial data)

Why it's important: Optimizes processing by applying sentiment analysis only where relevant.

4. Metadata Enrichment

Purpose: Extract additional insights to enhance search functionality.

- Named Entity Recognition (NER): Identify people, organizations, locations, dates
- Keyword Extraction: Pull important keywords/topics from text
- Sentiment Analysis: Apply sentiment scoring when appropriate

Why it's important: Enables powerful hybrid search with semantic search and metadata filtering.

5. Final Output Preparation

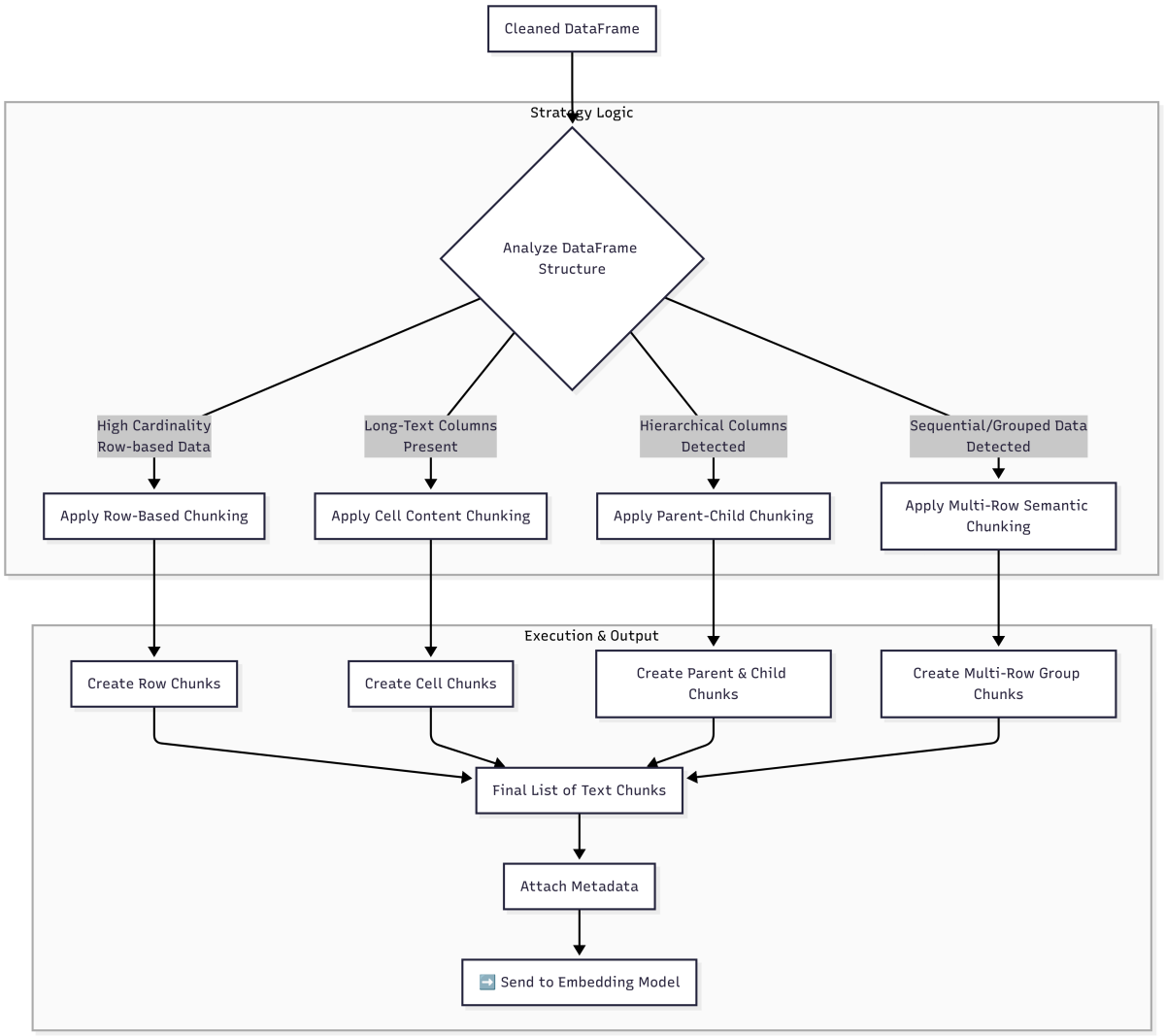
Purpose: Package processed data for the next pipeline stage.

- Add to Metadata: Compile enriched information with original non-text data
- Proceed to Chunking: Send cleaned text and metadata to the chunking stage

Overall Flow Summary

Stage	Input	Output	Key Question
Structural Validation	Raw CSV	Well-formed DataFrame	"Is this file readable?"
Text Cleaning	Raw Text	Clean, Standardized Text	"Is the text ready for the AI model?"
Conditional Analysis	Clean Text	Decision	"Would sentiment analysis add value here?"
Metadata Enrichment	Clean Text	Entities, Keywords, Sentiment	"What extra filters can I add for better search?"
Final Output	Clean Text + Metadata	Data for Chunking	"Is everything packaged for the next step?"

3. Optimized Chunking



Detailed Chunking Strategies

This architecture employs four adaptive chunking strategies that can be applied simultaneously to optimize data retrieval.

1. Row-Based Chunking

- Trigger: Default strategy for most CSVs, ideal for self-contained records
- Process: Treats each row as an independent chunk by combining all column values
- Example: "Product ID 101: Wireless Headphones. Price: $49.99. Category: Audio."
- Advantages: Simple implementation, preserves row-level context
- Limitations: May create overly short or long chunks depending on text columns

2. Cell Content Chunking

- Trigger: Activates for columns with long-form text (descriptions, reviews, comments)
- Process: Splits text-heavy cells into smaller, semantically coherent chunks while retaining row metadata
- Example: Description cell split into multiple chunks about different product features
- Advantages: Improves retrieval of long texts, creates multiple search points per record

- Limitations: Requires complex implementation and careful metadata handling

3. Multi-Row Semantic Chunking

- Trigger: Activates for sequential or grouped data (logs, time-series, conversation threads)
- Process: Groups and chunks consecutive rows sharing common context
- Example: "Error Report for Time: 14:23: Multiple connection timeouts from user IP 192.168.1.5."
- Advantages: Preserves context that spans multiple rows
- Limitations: Requires identifying appropriate grouping keys

4. Parent-Child Hierarchical Chunking

- Trigger: Activates for columns implying hierarchical relationships
- Process: Creates chunks at different levels of granularity (parent overview, child details)
- Example: Parent chunk for "Audio Equipment" category, child chunks for individual products
- Advantages: Enables multi-level search and retrieval
- Limitations: Most complex implementation, requires careful chunk linking

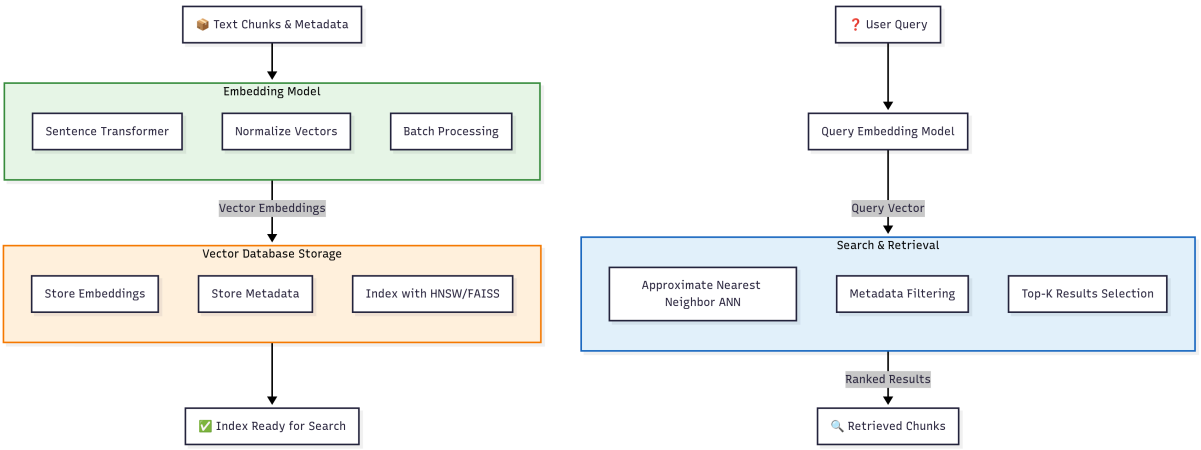
Implementation Architecture

- Dynamic Analysis: System analyzes column types, text lengths, hierarchies, and sequences
- Hybrid Application: Multiple strategies can process the same data simultaneously
- Metadata Tagging: Each chunk includes detailed metadata for traceability and filtering
- Adaptive Triggering: Uses data analysis to determine appropriate chunking strategies

Benefits of This Architecture

- Adaptive: Chooses strategies based on data characteristics, not fixed rules
- Redundant: Creates multiple retrieval paths for the same information
- Comprehensive: Covers all major chunking strategies for structured data
- Production-Ready: Implements advanced RAG techniques for high-accuracy retrieval

4. Intelligence Search Layer (Embedding, Storing, Search & Retrieval)



Detailed Intelligence Search Layer Steps

1. Embedding Generation

What it is: The process of converting text into numerical vectors (arrays of numbers) that represent its semantic meaning.

Core Component: Sentence Transformer Model (e.g., all-MiniLM-L6-v2), a specialized AI model trained to produce vectors where similar sentences are close together in vector space.

Key Processes:

- Batch Processing: Embeds hundreds of chunks simultaneously for efficiency
- Vector Normalization: Scales all vectors to unit length for accurate cosine similarity search
- Input: List of text chunks (["Wireless headphones...", "Battery life is 30 hours..."])
- Output: List of numerical vectors ([[0.12, -0.45, ..., 0.98], ...])

2. Vector Database Storage & Indexing

What it is: A specialized database designed to store and efficiently query vector embeddings.

Core Component: ChromaDB or FAISS, optimized specifically for fast similarity search.

Key Processes:

- Dual Storage:
- Vector Store: The dense vector embeddings
- Metadata Store: Corresponding metadata for each chunk (chunk_id, source_file, etc.)
- Indexing: Creates HNSW or IVF index with "shortcuts" to find approximate nearest neighbors quickly
- Input: List of vectors + their metadata
- Output: A persisted, indexed vector store ready for querying

3. Query Processing & Search

What it is: The real-time process of answering a user's query by finding the most relevant chunks.

How it Works:

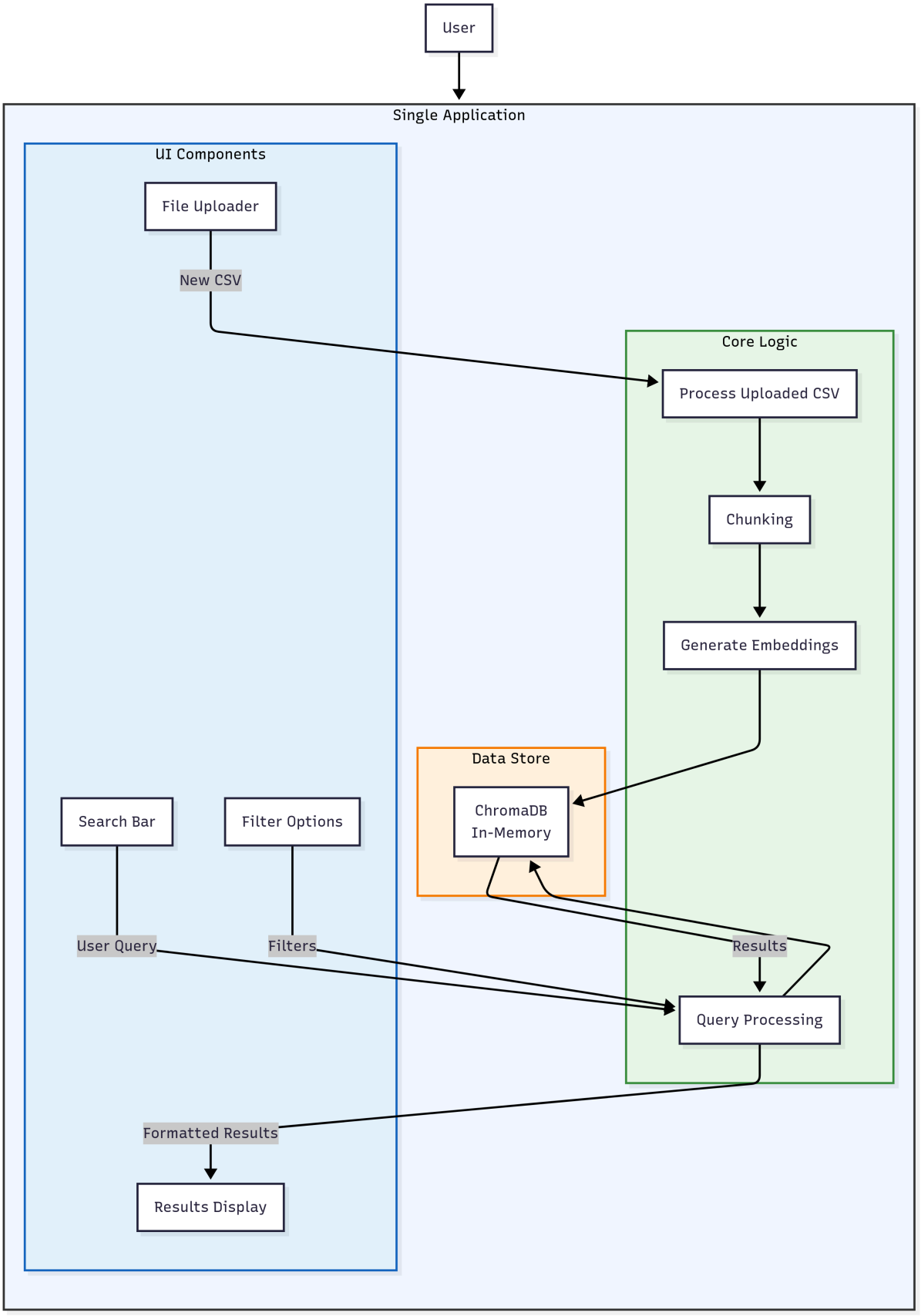
- Query Embedding: Converts the user's natural language query into a vector using the same model
- Similarity Search: Performs Approximate Nearest Neighbor search through the index
- Metadata Filtering: Enables hybrid search (e.g., vector_search("headphones") + filter(metadata["price"] < 100))
- Top-K Results: Retrieves the top K most similar chunks and their metadata
- Input: User query string
- Output: Ranked list of semantically relevant text chunks and their sources

Why This Architecture is Intelligent & Dynamic

- Semantic Understanding: Finds relevant content based on meaning, not just keywords
- Blazing Fast Search: ANN index allows for sub-second retrieval from millions of chunks
- Hybrid Search Capability: Combines vector search with metadata filtering
- Domain Adaptable: Transformer model can be swapped or fine-tuned for specific domains

This Intelligence Layer is the core of your RAG system, bridging the gap between human language and machine-readable data for powerful semantic search.

5.Interface Layer Architecture (Web & API)



Detailed Interface Layer Architecture Steps

1. The User

This is the person interacting with the application through a web browser. They have two main jobs:

- Provide Data: Upload a CSV file to build the knowledge base.
- Ask Questions: Enter a search query to find information within that data.

2. The Streamlit Web App (The Complete System)

This box contains the entire application, built as a single Streamlit script. It's logically divided into three interconnected layers that work together.

3. The Three Core Layers Inside the App

A. UI Layer (The Face)

This is what the user sees and interacts with—the website itself. It contains all the visual elements:

- A file uploader for accepting CSV files.
- A search bar for typing queries.
- Filter options (e.g., dropdowns, sliders) to refine searches.
- A results area that displays the answers in a clean, formatted way.

B. Application Logic Layer (The Brain)

This is where all the heavy thinking and processing happens. It has four key jobs:

- Process Data: It takes the uploaded CSV file, reads it, and cleans it up.
- Chunk Data: It breaks down the text from the CSV into smaller, logical pieces (e.g., by paragraph or section). This is crucial for finding precise answers.
- Create Understanding: This is the magic step. It uses an AI model to convert each chunk of text into a numerical representation (called an embedding) that captures its semantic meaning. Words and sentences with similar meanings have similar numerical representations.
- Handle Queries: When a user searches, it takes their query, understands its meaning (by also converting it to an embedding), and figures out what to ask the database.

C. Data Store Layer (The Memory)

This layer is responsible for storage and retrieval.

- It uses ChromaDB, a specialized database designed for this purpose.
- It stores the numerical embeddings (from the Brain) along with the original text chunks.
- Its primary job is to perform similarity search: it finds the text chunks whose numerical embeddings are most similar to the numerical representation of the user's query. It doesn't just match keywords; it matches concepts.

4. How the Data Flows: Two Key Processes

Process 1: Ingesting New Data (Building the Knowledge Base)

This happens when a user uploads a CSV file.

- The file moves from the UI to the Application Logic.
- The Logic layer processes the file, chunks the text, and generates embeddings (numerical representations) for each chunk.
- These embeddings are sent to the Data Store (ChromaDB) to be saved for future searches.

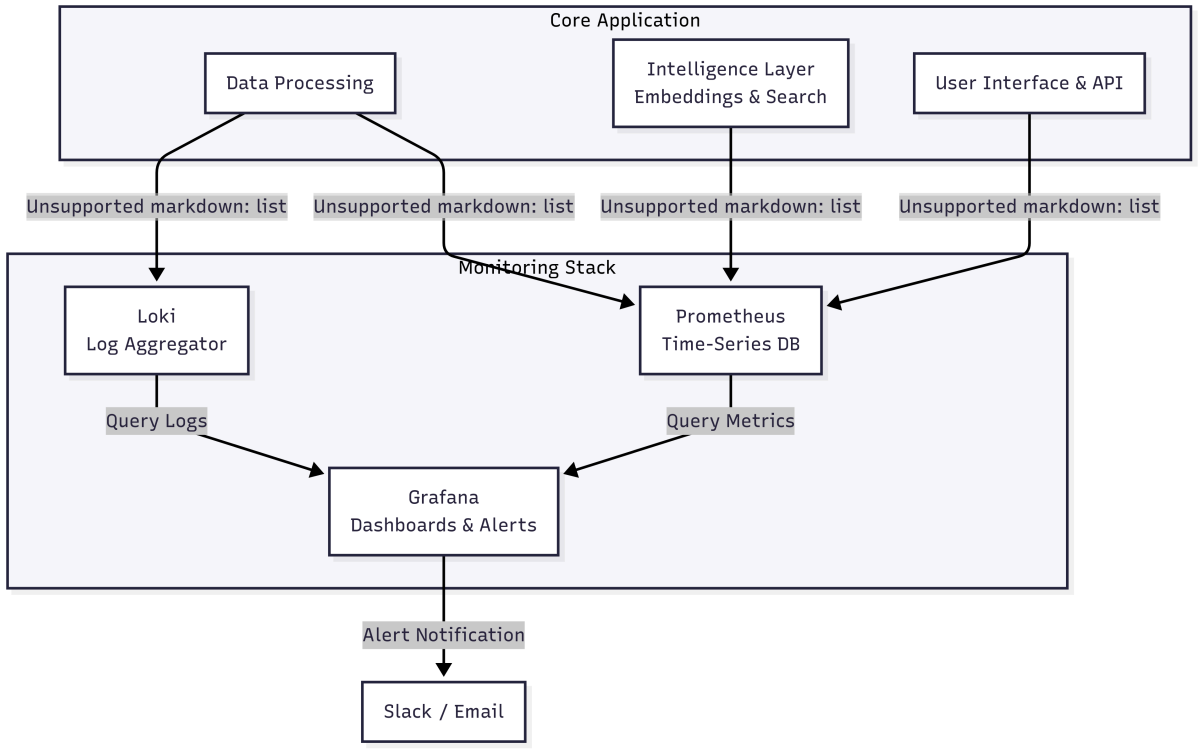
Process 2: Answering a Query (Searching the Knowledge Base)

This happens when a user types a question.

- The query moves from the UI to the Application Logic.
- The Logic layer processes the query, converting it into an embedding.
- The Logic asks the Data Store: "Find the text chunks that are conceptually closest to this query embedding."
- The Data Store performs the ultra-fast similarity search and returns the most relevant results.

- These results are sent back to the Logic layer, which formats them.
- Finally, the formatted answers are displayed to the user in the UI.

6. Monitoring Layer



Detailed Monitoring Layer Steps

1. Data Quality Monitoring

Goal: Ensure the uploaded CSV files are valid and useful.

How: Add checks during the Data Cleaning stage.

Metrics (to Prometheus):

- csv_rows_processed_total (Counter) - How many rows were ingested.
- csv_columns_total (Gauge) - Number of columns detected.
- csv_null_values_total (Counter) - Count of empty cells.

Logs (to Loki):

- Log a warning for schema changes (e.g., "Expected 5 columns, got 7").
- Log an error for completely unparseable files.

Simple Alert:

- Alert IF: csv_null_values_total > 100 - Too many missing values.

2. Chunk Validation

Goal: Verify your text chunks are a good size for accurate search.

How: Analyze chunks after the Chunking stage.

Metrics (to Prometheus):

- chunks_created_total (Counter) - Total chunks generated.
- chunk_size_chars (Histogram) - Distribution of chunk sizes in characters. This is the most important metric for chunking.

Logs (to Loki):

- Periodically log a few sample chunks (as DEBUG level). This lets you manually check what the chunks actually look like.

Simple Alert:

- Alert IF: chunk_size_chars > 2000 - Chunks are too large, which will hurt search relevance.

3. Performance Metrics

Goal: Ensure the system is fast and responsive.

How: Time key operations in the Embedding and Search layers.

Metrics (to Prometheus):

- embedding_duration_seconds (Histogram) - Time to generate a single embedding.
- query_duration_seconds (Histogram) - End-to-end time to process a user query.
- similarity_search_duration_seconds (Histogram) - Time just for the vector database search.

Simple Alert:

- Alert IF: query_duration_seconds > 5.0 - User queries are too slow.

4. System Monitoring

Goal: Track overall health and usage.

How: Instrument the User Interface and API.

Metrics (to Prometheus):

- user_queries_total (Counter) - Total number of searches. Track this to see if anyone is using the app!
- active_users (Gauge) - Number of active users in a time window.
- http_requests_total (Counter) - Count of API requests by status code (200, 400, 500).

Logs (to Loki):

- Log all API requests (method, path, status code, response time). Great for debugging.

Simple Alert:

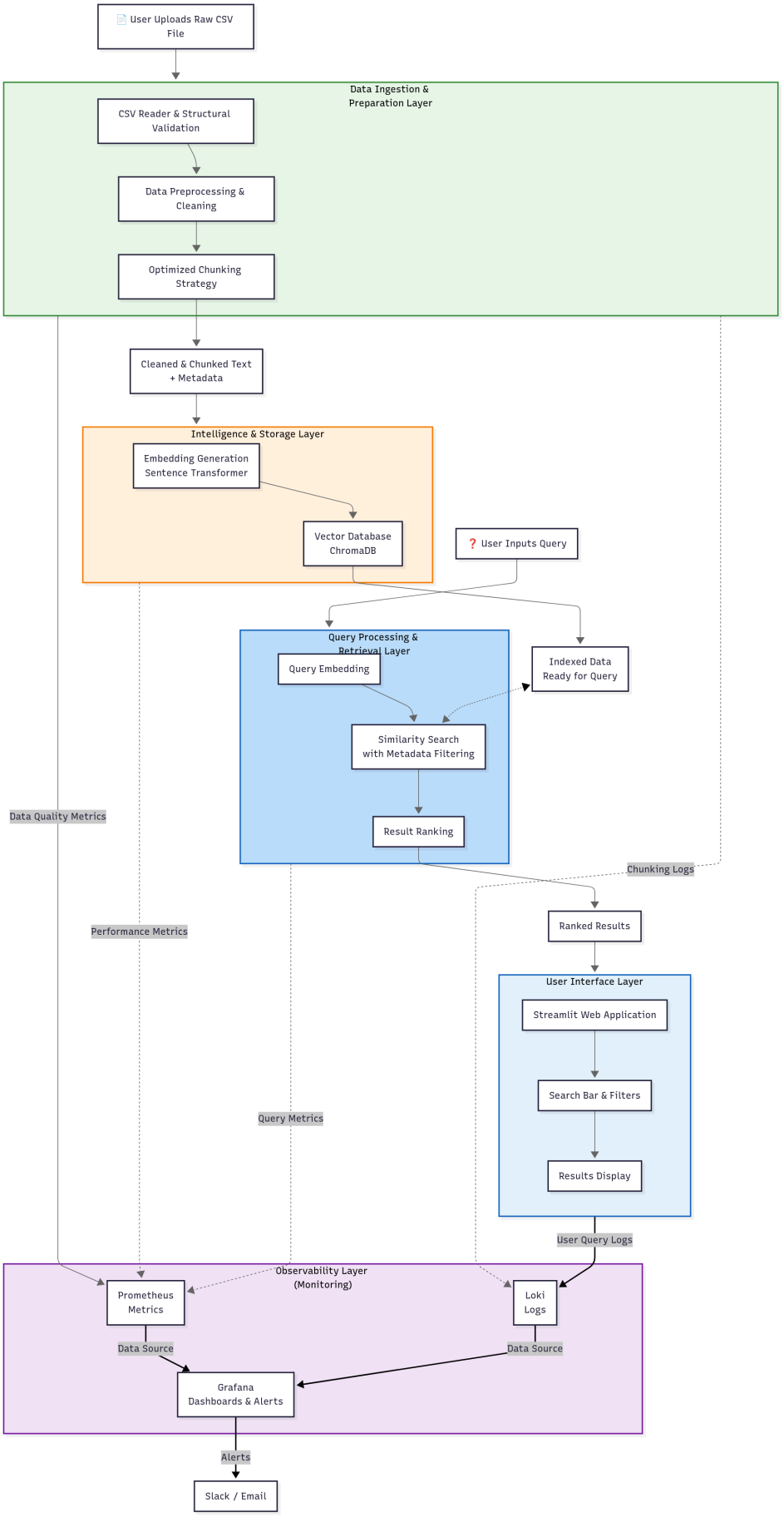
- Alert IF: up == 0 - The application is down.
- Alert IF: rate(http_requests_total{code="500"}[5m]) > 1 - There are server errors.

Why This is Efficient and Useful

- Lightweight: Prometheus and Loki are built for this scale and are easy to run.
- Unified Dashboard: Grafana pulls everything together into a single glass pane. You can create one dashboard with a section for each of these four areas.
- Actionable Alerts: The alerts are focused on real problems that you can actually fix (e.g., "chunks are too big," "the app is down," "queries are slow").
- Debugging Power: When an alert fires, you can jump from Grafana to Loki to see the detailed logs from that exact time period, making root cause analysis fast.

This setup provides a powerful foundation for maintaining reliability and performance without requiring a dedicated operations team.

7. Layer By Layer Workflow


📄 User Uploads Raw CSV File
Data Ingestion & Preparation Layer
CSV Reader & Structural Validation
Data Preprocessing & Cleaning
Optimized Chunking Strategy
Cleaned & Chunked Text + Metadata
Intelligence & Storage Layer
Embedding Generation Sentence Transformer
Vector Database ChromaDB
❓ User Inputs Query
Query Processing & Retrieval Layer
Query Embedding
Indexed Data Ready for Query
Similarity Search with Metadata Filtering
Result Ranking
Data Quality Metrics
Chunking Logs
Performance Metrics
Ranked Results
User Interface Layer
Streamlit Web Application
Search Bar & Filters
Results Display
Query Metrics
User Query Logs
Observability Layer (Monitoring)
Prometheus Metrics
Loki Logs
Data Source
Data Source
Grafana Dashboards & Alerts
Alerts
Slack / Email

Detailed Steps

Layer 1: Data Ingestion & Preparation

1. CSV Upload: User provides a raw CSV file through the Streamlit interface.
2. Structural Validation:

- 2.1 Detect Encoding: The system automatically detects the file's character encoding (e.g., UTF-8, Windows-1252) to avoid parsing errors.
- 2.2 Identify Delimiter: It determines if the file uses commas, tabs, or another character to separate values.
- 2.3 Validate Headers: It checks for the presence of headers and ensures the number of columns is consistent throughout the file. [MONITORING: Errors here are logged to Loki].

3. Data Loading: The validated file is loaded into a Pandas DataFrame for processing.
4. Text Cleaning: For each text column, the system applies:

- 4.1 Convert to String: Ensures all values are strings, preserving NaN for missing data.
- 4.2 Lowercase & Strip: Converts text to lowercase and removes leading/trailing whitespace.
- 4.3 Remove Extra Whitespace: Collapses multiple spaces into one.
- 4.4 Remove HTML Tags: Strips any HTML tags present in the text. [MONITORING: A metric for rows_processed is sent to Prometheus].

5. Metadata Enrichment (Conditional):

- 5.1 Decision Point: The system analyzes the content. If the data is subjective (e.g., reviews), it enables enrichment.
- 5.2 Enrichment: Runs NLP pipelines to add metadata like sentiment score, extracted keywords, or named entities (people, places). This significantly enhances future filtering.

6. Chunking Strategy:

- 6.1 Analysis: The system analyzes the DataFrame's structure to choose the best method.
- 6.2 Execution: It applies the chosen strategy (row-based, cell-based, etc.).

7. Final Output: A list of text chunks is produced, each with attached metadata (original row data + any new enrichments). [MONITORING: A histogram of chunk_sizes is sent to Prometheus].

Layer 2: Intelligence & Storage

1. Embedding Generation:

- 1.1 Model Load: Loads the pre-trained Sentence Transformer model (e.g., all-MiniLM-L6-v2).
- 1.2 Vector Conversion: Processes the list of text chunks in batches, converting each into a high-dimensional vector (e.g., 384 dimensions). [MONITORING: The time taken embedding_duration_seconds is recorded].
- 1.3 Normalization: Vectors are normalized to unit length, which is critical for using cosine similarity as the distance metric.

2. Vector Database Storage:

- 2.1 Connection: Establishes a connection to the ChromaDB instance.
- 2.2 Collection Management: Creates or retrieves a "collection" (like a table) named documents.
- 2.3 Storage: Stores the normalized vectors in the collection. Each vector entry is linked to its original text chunk and all metadata as a JSON-like object.
- 2.4 Indexing: ChromaDB automatically builds an HNSW index on the vectors, optimizing them for extremely fast approximate nearest neighbor search.

Layer 3: Query Processing & Retrieval

1. Query Input: User types a natural language question and applies optional filters (e.g., "date > 2020").

2. Query Embedding: The user's query string is passed to the same embedding model, converting it into a vector. [MONITORING: The query text is anonymized and logged to Loki for debugging].
3. Search Formulation: The system formulates a query for the VectorDB: "Find the top 5 vectors closest to this query vector, but only where the source_file is 'X' and sentiment is 'positive'".
4. Execution: The VectorDB performs the ANN search on its indexed vectors. This is incredibly fast, even for large datasets.
5. Result Retrieval: The DB returns the IDs of the most similar vectors, their similarity scores, and their associated metadata.
6. Fetch Text: The system uses the IDs to retrieve the original text chunks from the database.
7. Ranking: The final list of (text, score, metadata) tuples is sorted by the similarity score (highest to lowest). [MONITORING: The total query_duration_seconds is measured and sent to Prometheus].

Layer 4: User Interface

1. Formatting: The ranked results are formatted for readability.
2. Highlighting: Key terms or matches can be highlighted to help the user see why a result was relevant.
3. Rendering: Streamlit displays the results in a clean, web-based interface, often showing the source metadata to provide context.
4. Completion: The user reads the results, completing the cycle.

Layer 5: Observability (Continuous)

This layer operates continuously and in parallel:

1. Metrics Flow: Prometheus scrapes the /metrics endpoint from the app every 15 seconds, collecting all numeric performance data.
2. Logs Flow: A Promtail agent tails the application's log files and ships them to Loki.
3. Dashboarding: Grafana queries both data sources to populate live dashboards showing system health, performance trends, and data quality.
4. Alerting: Grafana continuously evaluates alert rules. If a metric breaches a threshold (e.g., latency > 2s, error rate > 1%), it sends immediate notifications to Slack or Email, enabling proactive incident response.

8. Tech Stack

Comprehensive Open-Source Tech Stack for CSV Semantic Search Pipeline

This table lists potential technologies for each component of your architecture, with a primary recommendation and alternatives for consideration.

Layer / Component	Primary Recommendation (& `pip install` command)	Alternative Options (& `pip install` command)	Notes / Why Choose
Core Framework	**LangChain** (`pip install langchain`)	**LlamaIndex** (`pip install llama-index`)	**LangChain** is more flexible for buildin complex, custom pipelines like yours. **LlamaIndex** is more focused a streamlined for RAG.
CSV Reader & Data Manipulation	**Pandas** (`pip install pandas`)	**Polars** (`pip install polars`)	**Pandas** is the industry standard with immense community support. **Polars** much faster for large datasets and ha clean API.
Text Cleaning & Preprocessing	**BeautifulSoup4** (`pip install beautifulsoup4`) for HTML stripping.**NumPy** (`pip install numpy`) for NaN handling.	**lxml** (`pip install lxml`) for faster HTML/XML parsing.	Built-in string methods are used for lowercasing and stripping. Pandas handles `NaN` values. BeautifulSoup i robust for messy HTML.

Layer / Component	Primary Recommendation (& `pip install` command)	Alternative Options (& `pip install` command)	Notes / Why Choose
NLP (NER, Sentiment, Keywords)	**spaCy** (`pip install spacy`) + model (`python -m spacy download en_core_web_sm`)**TextBlob** (`pip install textblob`) for quick sentiment.	**NLTK** (`pip install nltk`)**Transformers** (`pip install transformers`) for advanced models.	**spaCy** is production-ready, fast, and accurate for NER. **TextBlob** is simpler basic sentiment heuristic. For advanc sentiment, use a `transformers` model.
Text Splitting (Chunking)	**LangChain Text Splitters** (comes with `langchain`)	**Tokenizers** from `transformers` or `tiktoken` (`pip install tiktoken`) for token-based splitting.	LangChain's `RecursiveCharacterTextSpli` the default and works well. For precis chunk sizes for a specific model, use tokenizer.
Embedding Model	**Sentence Transformers** (`pip install sentence-transformers`)	**OpenAI Embeddings** (`pip install openai`)**Hugging Face Transformers** (`pip install transformers`)	**Sentence Transformers** is the best o source choice. Models like `all-MiniLM-v2` are fast and effective. OpenAI is n open-source and has cost.
Vector Database (Client)	**Chroma** (`pip install chromadb`)	**FAISS** (`pip install faiss-cpu`) // for CPU(`pip install faiss-gpu`) // for GPU**Weaviate Client** (`pip install weaviate-client`)	**Chroma** is simple, pure-Python, and persists easily. **FAISS** is a library, not database; great for static indexes. **Weaviate** is a full-fledged DI needs a server.
Application Framework	**Streamlit** (`pip install streamlit`)	**Gradio** (`pip install gradio`)**FastAPI** (`pip install fastapi`) + **UVicorn** (`pip install uvicorn`)	**Streamlit** is perfect for quick, data-ce UIs. **Gradio** is great for demoing ML models. **FastAPI** is for building a robu REST API.
Monitoring - Metrics Client	**Prometheus Client** (`pip install prometheus-client`)	-	The standard for exposing metrics in format for Prometheus to scrape.
Monitoring - Logging	**Structlog** (`pip install structlog`) or built-in **logging** module	-	Use to generate structured logs in JS format for Loki.
Monitoring - Dashboard	**Grafana** (External Tool, not a pip package)	-	The industry standard for visualizing metrics and logs. Must be installed separately (e.g., Docker, apt-get).
Monitoring - Log Aggregator	**Loki** (External Tool, not a pip package)	-	Must be installed separately. The `promtail` agent is needed to ship to Loki.

9. Sprint Goal: Build a functional MVP that can ingest a CSV, process it, and allow for semantic search via a simple web interface.

Week 1: Foundation & Core Processing
Focus: Setting up the environment and building the data processing and chunking pipeline.

Day	Task	Description	Deliverable
Day 1	Project Setup & Core Dependencies	Initialize project repository (e.g., Git). Create `requirements.txt` with libraries (LangChain, Pandas, Sentence-Transformers, ChromaDB, Streamlit). Set up virtual environment and install packages.	Working Python environment with dependencies installed. GitHub/GitLab repository.
Day 2	Robust CSV Reader & Validation	Implement CSV reading using Pandas. Add validation: auto-detect encoding, delimiter, validate headers. Handle errors gracefully.	Python function that returns cleaned DataFrame or useful error message.
Day 3	Text Preprocessing Pipeline	Build text cleaning functions: lowercase, strip whitespace, remove HTML tags. Separate text columns (to be cleaned) from numeric/categorical (stored as metadata).	Module with preprocessing functions returning cleaned DataFrame.
Day 4	Implement Row-Based Chunking	Develop chunking strategy: convert each row into formatted text (e.g., `"ColumnName: Value, OtherColumn: Value"`).	Function that outputs list of text chunks from DataFrame.

Day	Task	Description	Deliverable
Day 5	Implement Cell-Based Chunking	Use LangChain's `RecursiveCharacterTextSplitter` for long text columns (e.g., descriptions). Preserve row metadata for each new chunk.	Function to generate chunks from specific long-text columns.

Week 2: Intelligence, Search & User Interface Development

Day	Task	Description	Deliverable
Day 6	Embedding Generation & Storage	Initialize Sentence Transformer model. Create function to take list of text chunks, generate embeddings in batches, and store them in ChromaDB collection with metadata.	Script that processes chunks and populates ChromaDB vector store.
Day 7	Search & Retrieval Logic	Build core search function: (1) take user query, (2) generate embedding, (3) query ChromaDB for similar vectors, (4) return top-K results with text + metadata.	Function `search_query(query: str, filters: dict)` returning ranked results.
Day 8	Streamlit UI – Data Ingestion	Build UI components: file uploader + processing button. Trigger backend pipeline (Days 2–5) and store resulting vectors in ChromaDB (Day 6).	Web app where user uploads CSV and sees **"Processing Successful"** message.
Day 9	Streamlit UI – Search Interface	Build search bar + results display in Streamlit. Connect UI to search function (Day 7). Display results neatly, showing source text and metadata.	Web app where user asks a question and gets semantic search results.
Day 10	Debugging, Polish & Stretch Goals	Test pipeline with multiple CSVs. Fix bugs. Add loading indicators. Stretch goals: filter sidebar, better error messages, basic README for project.	Robust functional MVP + plan for future improvements.