# Milestone 4:  Polishing & Integration

## 1. Introduction

The primary objective of Milestone 4 is to transition the Receipt & Invoice Digitizer from a functional prototype to a robust, production-grade application. While previous milestones established the core pipeline—ingesting images, extracting text via OCR, and storing raw data—this phase addresses the critical challenges of **data integrity**, **user accessibility**, and **system scalability**.

In this phase, we move beyond simple data capture to implement a "Hybrid Extraction Engine" that combines the flexibility of Large Language Models (LLMs) with the precision of rule-based templates. Concurrently, the user interface is upgraded to support granular data retrieval, and the backend database interactions are refactored to minimize latency and memory overhead.

## 2. Key Architectural Enhancements

### 2.1. High-Fidelity Extraction via Template Parsing

Standard OCR and LLM-based parsing are inherently probabilistic; they "guess" the content based on patterns. While effective for general text, this approach struggles with the rigid mathematical precision required for financial auditing.

To resolve this, Milestone 4 introduces a **Template-Based Parsing Layer**:

- **Deterministic Standardization:** We implement a logic layer that recognizes recurring vendors (e.g., "Walmart," "Coffee House") and normalizes their naming conventions, eliminating data fragmentation caused by OCR variations (e.g., "Wal-mart" vs. "Walmart Inc").
- **Logic-Based Gap Filling:** By storing known metadata (such as local tax rates for specific vendors), the system can now mathematically reconstruct missing data points—such as calculating a missing subtotal based on the total and tax rate—rather than relying solely on visual extraction.
- **Validation Scoring:** A comparison algorithm now runs parallel to the extraction, flagging low-confidence data before it enters the database.

### 2.2. Dynamic Data Visualization (Search & Filtering)

A financial database is only as valuable as the insights it can generate. Previously, the application provided a static, lifetime view of all transactions, which obscured short-term trends and specific anomalies.

We have implemented an **Interactive Analytics Engine** that allows users to slice data dimensions:

- **Temporal Filtering:** Users can now isolate spending behaviors within custom date ranges (e.g., "Fiscal Q1" or "Last 30 Days").
- **Categorical Drill-Down:** The dashboard now supports multi-select filtering, allowing users to analyze specific cost centers (e.g., "Travel" vs. "Office Supplies") independently.
- **Reactive UI:** The integration ensures that all Key Performance Indicators (KPIs) and visualization charts update in real-time as filters are applied, providing immediate feedback for budget analysis.

### 2.3. Backend Optimization and Query Refactoring

As the dataset grows from hundreds to thousands of records, client-side processing (loading all data into Python memory) becomes a performance bottleneck. Milestone 4 shifts the "heavy lifting" from the application layer to the database layer.

- **Server-Side Aggregation:** We have replaced full-table data fetches with optimized SQL queries (using SUM, COUNT, AVG, and GROUP BY). This drastically reduces the data transfer payload between the disk and the application.
- **Database Indexing:** B-Tree indexes have been applied to high-traffic columns (date, merchant, category). This changes the time complexity of search operations from $O(N)$ (scanning every row) to $O(\log N)$ (binary search), ensuring instant retrieval times regardless of database size.

## 3. Scope of Milestone 4

## 3.1. Advanced Data Extraction & Validation (The Logic Layer)

- **Template-Based Parsing:** Implementation of a deterministic logic layer to handle specific, high-frequency vendors (e.g., "Coffee House," "Walmart").
- **Data Normalization:** Automatic standardization of vendor names (mapping aliases like "Coffee House Inc." to "Coffee House").
- **Mathematical Gap-Filling:** Logic to calculate missing financial fields (e.g., inferring Tax or Subtotal) using known tax rates stored in the template system.
- **Confidence Scoring:** Visual indicators comparing "AI-extracted" data vs. "Template-corrected" data.

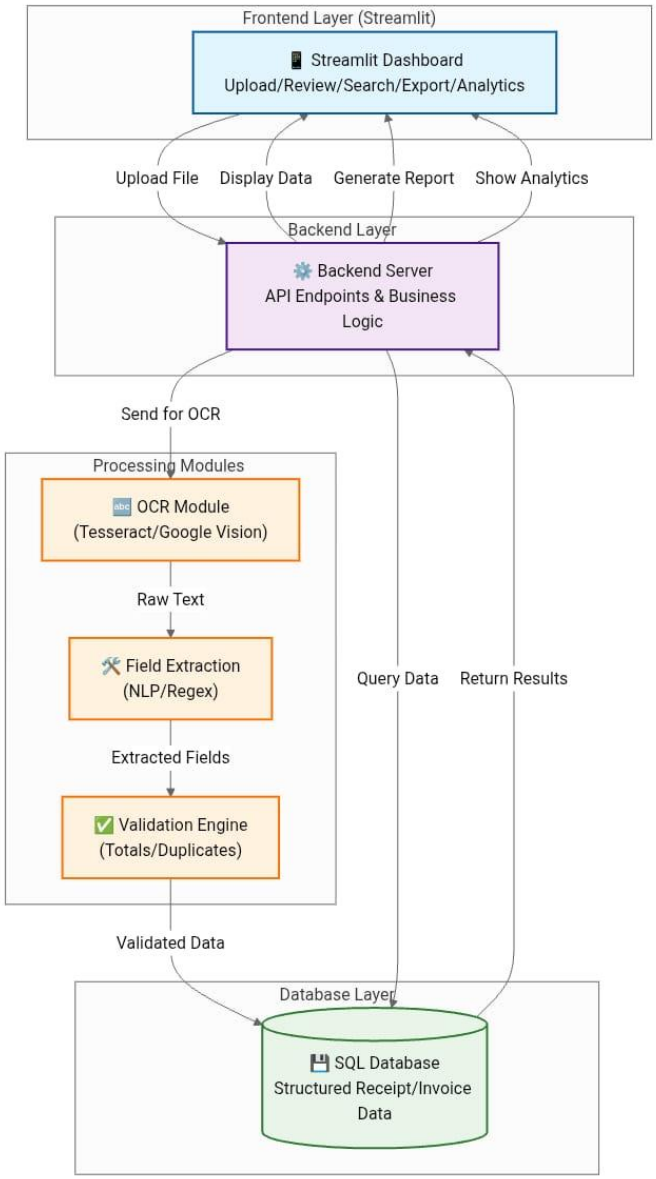
## 3.2. Interactive Analytics Dashboard (The UI Layer)

- **Dynamic Filtering:** Implementation of sidebar controls allowing users to filter analytics by:
    - **Date Range:** (Start Date to End Date).
    - **Category:** Multi-select options (e.g., Food, Transport, Utilities).
- **Reactive Visualizations:** All charts (Pie, Bar, Line) and KPIs (Total Spend, Avg Receipt) must update instantly based on the selected filters.
- **Global Search:** A text-based search bar to query the database by Invoice Number, Vendor Name, or Item Name.

## 3.3. Backend Optimization (The Performance Layer)

- **Database Indexing:** Implementation of SQLite B-Tree indexes on high-traffic columns (date, merchant, category) to reduce query time.
- **Server-Side Aggregation:** Refactoring analytics queries to perform calculations (SUM, AVG, COUNT) within the SQL engine rather than in Python memory.

## 4.System Architecture Diagram




## 5. Improve extraction accuracy with template-based parsing

### 5.1 Objective overview

In the earlier milestones, the application relied solely on **Optical Character Recognition (OCR)** and **Large Language Models (LLMs)** to read receipts. While this approach is flexible (it can read *any* receipt), it suffers from two critical flaws in a financial context:

1. **Inconsistency (The "Naming" Problem):**
    a. An AI might read a receipt from "Starbucks" as *"Starbucks Coffee"* on Monday and *"Starbucks Store #234"* on Tuesday.
    b. **Result:** Analytics are fragmented. You cannot accurately track total spending because the system treats them as two different merchants.
2. **Hallucination & Omission (The "Math" Problem):**
    a. If a receipt is slightly blurry, OCR often fails to read the small "Tax" line.
    b. Generic AI models often "guess" or simply return $0.00 for tax, breaking the mathematical validity of the record ($Subtotal + Tax \neq Total$).

**The system now operates in a Hybrid Mode. It uses the AI to "read" the text, but uses the Template Engine to "correct" it.**

### 5.2. Vendor Standardization (Normalization)

The system checks the extracted merchant name against a VENDOR_TEMPLATES dictionary (a master list of known vendors).

- **Scenario:** OCR extracts *"The Coffee House Inc."*
- **Action:** The Template Engine recognizes this as an alias for *"Coffee House"*.
- **Result:** The database saves it strictly as **"Coffee House"**. This ensures that when you filter by "Coffee House," *every* transaction appears, regardless of how the receipt was printed.

### 5.3. Gap-Filling (Tax Logic)

For known vendors, we store metadata like their standard tax rate (e.g., 8.25%).

- **Scenario:** A receipt has a total of $10.82, but the tax line is faded and unreadable. The AI returns Tax: $0.00.
- **Action:** The Template Engine detects "Coffee House" and knows the tax rate is 8.25%. It works backward:
    - $Total / 1.0825 = Subtotal$
    - $Total - Subtotal = Tax$
- **Result:** The system *mathematically reconstructs* the missing tax ($0.82) and saves the correct data, even though the image was unreadable.

### 5.4. Confidence Scoring

The system now assigns a "Quality Score" to every extraction.

- **Standard Parsing:** Score is lower if tax is missing or math doesn't add up.
- **Template Parsing:** Score is boosted (e.g., +25 points) because the vendor was positively identified and validated against a known rule set.

## 6. Add Search/Filter in Dashboard

## 6.1 Objective Overview

The primary goal of this task within Milestone 4 (Polishing & Integration) is to transition the dashboard from a static display of all records to a dynamic, interactive data exploration tool. Initially, the dashboard likely shows a list or table of all digitized receipts. While functional, this approach becomes inefficient as the database grows. The "Add search/filter" feature empowers users to quickly pinpoint specific transactions by querying the database based on defined criteria, thereby enhancing the usability and analytical power of the entire system.

### Functional Requirements

The search and filter functionality is designed to be intuitive and comprehensive. Key filter criteria include:

- **Text-Based Search:** A global search bar allowing users to find receipts by vendor name, or invoice number. This will perform a partial text match against relevant fields in the database.
- **Date Range Filter:** A date picker or input fields enabling users to view receipts within a specific time frame (e.g., all receipts from last month, a specific quarter, or a custom range). This is crucial for period-specific reconciliation and reporting.
- **Numeric Filters:** Input fields to filter records by amount (e.g., greater than $100, less than $50, or between two values). This helps in quickly identifying high-value transactions or spotting anomalies.
- **Categorical Filters:** Dropdown menus populated with distinct values from the database, such as "Vendor" or "Payment Status," allowing for one-click filtering.

## 6.2 Technical Implementation

The implementation will follow a standard pattern for interactive web applications:

1. **Frontend (Streamlit):** We added interactive widgets to the dashboard sidebar or top panel. These widgets (e.g., select box, text input, date slider) will capture the user's filter criteria without reloading the page. An "Apply Filters" or "Search" button will trigger the data refresh.
2. **Backend (Application Logic):** When the user applies filters, the frontend sends the criteria to the backend. The application logic will dynamically construct a parameterized SQL query based on which filters are active. For example, if only a vendor and date range are provided, the query will be:
   sql SELECT * FROM receipts WHERE vendor ILIKE %vendor input% AND receipt date BETWEEN start date AND end date;
   Using parameterized queries is essential for preventing SQL injection vulnerabilities.
3. **Database (SQL):** The SQL database will execute the constructed query and return only the filtered subset of records. This approach ensures that data processing is handled efficiently by the database engine, rather than filtering a large dataset in the application's memory, which would be slow and resource intensive. Appropriate

indexes on frequently filtered columns like vendor, receipt date, and total amount will be created to ensure fast query performance even with thousands of records.

### 6.3 Expected Impact

Implementing this feature transforms the dashboard from a simple log into a powerful management tool. Users can now:

- **Prepare for Audits:** Filter all receipts from a particular vendor for a given fiscal period with a few clicks.
- **Perform Analysis:** Analyse spending trends by filtering for all transactions above a certain threshold or within a specific date range.
- **Improve Efficiency:** Eliminate the need to manually scroll through long lists of records, saving significant time and reducing the potential for human error.

By providing precise control over the displayed data, the search and filter functionality directly contributes to the project's goal of creating a user-friendly and efficient system for managing digitized financial documents.

## 7.Optimize DB queries and reports.

### 7.1 Objective Overview

As our Receipt & Invoice Digitizer stores more receipts and line items, the amount of data in the database increases. In the earlier version, the application loaded large amounts of data and then filtered or calculated results in Python. This worked for small data but became slower as the database grew.

To improve speed and efficiency, we optimized how the application retrieves and processes data from the database.

### 7.2 Why Optimization Was Needed

Before optimization:

- The system loaded all receipts even when only a few were needed.
- Filtering and searching were done after loading data.
- Analytics calculations were repeated every time the dashboard loaded.
- Duplicate detection required checking many records manually.
- Page reloads caused repeated database queries.

This caused slower performance and higher memory usage.

### 7.3 What We Improved

#### 1. Faster Searching with Indexing

We added indexes to frequently searched fields like date, merchant name, category, and invoice number.

This allows the database to find results quickly instead of scanning all records.

#### 2. Filtering Done by the Database

Earlier, the app loaded all data and filtered it later.
Now, filters like date, month, keyword, and category are applied directly in the database.

This means only required data is retrieved.

#### 3. Improved Duplicate Detection

Duplicate receipts are now detected using database filtering instead of checking each record in Python.

This makes duplicate detection much faster.

#### 4. Efficient Data Retrieval Using Joins

Instead of retrieving receipts and line items separately and merging them later, the database now combines related data in one query.

This reduces processing time and memory usage.

#### 5. Faster Reports Using Database Calculations

Totals, category spending, and vendor summaries are now calculated by the database instead of Python.

Databases are optimized for calculations, making reports load faster.

#### 6. Reduced Repeated Queries with Caching

Frequently used data like receipt lists and analytics summaries are temporarily stored.

This prevents repeated database calls when the page reloads.

### 7.4 Results of Optimization

After optimization:

- Search and filtering are much faster.
- Dashboard and reports load quickly.

- Memory usage is reduced.
- Database workload is minimized.
- The system can handle large numbers of receipts efficiently.

## 8. Conclusion

Milestone 4 significantly enhanced the system's accuracy, usability, and performance.

Template-based parsing improved extraction reliability by standardizing vendors and correcting missing financial data.

Advanced search and filtering features enabled users to quickly locate and analyse receipts, improving usability and financial tracking.

Database query optimization, indexing, caching, and pre-aggregated analytics ensured fast performance and scalability for growing datasets.

These enhancements make the Receipt & Invoice Digitizer a **robust, efficient, and production-ready system** suitable for real-world financial record management.