INVOICE PAYMENT DASHBOARD 2025

Project Documentation and Quality Assurance Report

by

Manohar Duggempudi

Table of Contents

1. Introduction

The Invoice Payment Dashboard 2025 project was initiated to provide an intuitive and intelligent platform for invoice and payment tracking. Given a raw dataset of around 5700 invoice payment records, the primary aim was to clean and structure this data carefully, build a robust backend API, and develop a dynamic and responsive frontend dashboard. Throughout the development, strong emphasis was placed on ensuring real-world applicability, precision in data handling, and user-centric visualizations. The project demanded careful preprocessing of messy financial records, thoughtful UI/UX planning, and strategic backend API design.

Folder Structure:

```
Invoice-Payment-Dashboard-2025/
├── backend/
│   └──         app.py
│   └──         cleaned_data.json
├── frontend/
│   ├── pages/
│   │   ├──     App.jsx
│   │   └──     InvoiceDetails.jsx
│   ├── components/
│   │   ├──     Charts and Widgets (multiple jsx files)
│   ├── styles/
│   │   └── (CSS files)
│   └──         main.jsx
└── data_parsing/
        └──     data_parsing.ipynb
```

Project Links:

- Live Frontend: *https://invoice-dashboard-2025.vercel.app/*
- Live Backend API: *https://invoice-dashboard-2025.onrender.com/*
    - Clients List: *https://invoice-dashboard-2025.onrender.com/api/invoices*
    - Client Invoices: *https://invoice-dashboard-2025.onrender.com/api/invoices/client%202*
    - Invoice Group ID: *https://invoice-dashboard-2025.onrender.com/api/invoice-details/Client2_1*
- GitHub Link: *https://github.com/Manohar-DLMR/Invoice-Dashboard-2025*

2. Pre-processing

2.1. Initial Exploration

The raw dataset containing about 5700 invoice payment records was initially loaded into a Pandas DataFrame. Basic operations were performed to inspect the structure, data types, and missing values.

Observations included:

- Invoice References varied significantly in structure.
- Some clients had multiple invoices under the same reference, invoice amount, invoice date, etc.

This exploration confirmed that deep preprocessing was required to standardize and clean the dataset.

2.2. Cleaning Invoice References

One of the most challenging aspects was handling the Invoice Reference field. Based on the samples provided (e.g., 2016/2999v1, RV 4426, 2019-4755A, AJE151204), references were highly inconsistent.

Strategy and Logic:

- Extract the correct year: If the year format was wrong or missing, it was retrieved from the "Date Invoiced" field.
- Remove unwanted characters: Suffixes like "v1", "Rev1", letters were removed if unnecessary.
- Standardize the format: Final cleaned format was YYYY-ReferenceNumber.

Code Function: extract_clean_ref_from_string

- Uses regular expressions to extract digits and ignore any other characters until a space or '/' or '-' is encountered.
- Matches the correct year with the invoice year.
- Picks appropriate reference numbers following the year.
- Handles fallback cases smartly.

Example Corrections:

Raw Invoice Reference	**Cleaned Reference**	**Reason**
2016/2999v1	2016-2999	Unwanted v1 removed

RV 4426	2019-4426	Year missing, extracted from Invoice Date
2019-4755A	2019-4755	Trailing letter A removed
2018/4471Rev1	2018-4471	Rev1 suffix removed
AJE151204	2015-1204	AJE was removed and 15 is ignored since it is equal to YY(2015 or 15)

This logic ensured all Invoice References followed a consistent and professional format.

2.3. Deep Parsing Logic for AJE151204

In the case of reference AJE151204, it did not follow any standard YYYY-Ref format. Here:

- The extract_clean_ref_from_string function extracted digit chunks from the Invoice Reference.
- Although the full year 2015 was known from the Invoice Date, the Invoice Reference string did not start with 2015.
- However, the logic also allowed matching based on the last two digits of the year (YY).
- In this case, "15" matched the first two digits of "151204", so "1204" was extracted as the clean Reference Number.
- Thus, the final Clean Invoice Reference became 2015-1204, aligning both the year and a correct reference number.
- This ensured references like AJE151204 (without separators) were still parsed consistently without losing meaning.

This fallback logic was critical to handle legacy references or handwritten inputs found in real-world invoice systems.

2.4. Client ID and Name Mapping

Clients initially had numeric IDs. To make API results more readable and user-friendly, a Client Mapping strategy was implemented:

- Unique Client IDs were sorted.
- Mapped to "Client 1", "Client 2", "Client 3", etc.
- Example: Client ID 1 is mapped to "Client 1”, Client ID 260 is mapped to “Client 260”.

2.5. Data Sorting for Cumulative Calculations

Before performing cumulative sums, data was sorted by:

- Client ID, Clean Invoice Ref, Invoice Amount, Date Invoiced, Date Paid

Sorting ensures that when calculating running totals (Cumulative Paid), payments are processed in logical sequence.

2.6. Cumulative Paid and Pending Amount Calculation

Payments were often made in parts. To track real payment status:

- Cumulative sum of Paid Amount was computed per Invoice.
- Pending Amount = Invoice Amount - Cumulative Paid.

Example: A client has made following payments for an invoice amount $29407.37

Paid Amount	**Cumulative Amount**	**Pending Amount**
14000	14000	15407.37
15407	29407	0.37 Rounded to 0

Threshold of $1 was set to consider minor over/under balances as Paid.

2.7. Duplicate Payments Detection

Sometimes, exact duplicate payments were recorded.

Logic:

- Duplicates identified by matching: Client ID, Clean Invoice Ref, Invoice Amount, Paid Amount, Date Invoiced, Date Paid.
- The first occurrence remained normal.
- Subsequent instances marked as Possible Duplicate.

2.8. Late Payment Identification

Late payments were flagged where:

- (Date Paid - Date Invoiced) > 30 days.
- The threshold of 30 days was chosen because "Net 30" payment terms are a global standard in invoicing.
- Payments made within 30 days are considered on time, while payments exceeding this period generally indicate a delay.
- This alignment with real-world business practices ensures that the late payment detection remains practical and relevant.

This flag was added for each invoice line item.

2.9. Invoice Group ID Generation

To make it easier to retrieve a full invoice history for a client:

- Grouped by Client, Invoice Ref, Amount, Date.
- Assigned an Invoice Group ID like Client1_1, Client1_2.

This ID is used by the frontend to show complete payment breakdown.

2.10. Saving Cleaned Data

Final cleaned data was:

- Saved to cleaned_data.json for API serving.
- (Optional) Also available as cleaned_data.csv.

All necessary columns were organized carefully for consistency and ease of frontend/backend integration.

2.11. Why Python and Jupyter Notebook Were Chosen

Python was chosen because:

- Powerful libraries like Pandas and NumPy simplify data manipulation.
- Easy handling of missing values, text cleaning, grouping.
- Regex and datetime operations are natural and straightforward.

Pandas DataFrames were the perfect data structure:

- Row-based and column-based transformations are efficient.
- Built-in groupby(), aggregation, sorting functions made processing intuitive.

Jupyter Notebook was chosen for parsing because:

- Immediate visualization of data allowed fast debugging.
- Step-by-step execution helped validate each cleaning stage.
- Easy to iterate corrections without full re-execution.

This setup ensured a fast, error-free, and professional preprocessing stage before moving to API and UI development.

3. Development Ideas

The idea was to carefully design the UI prioritizing usability, scalability, and responsiveness. The goal was to allow users to intuitively search, filter, and analyze client payment behaviors, even when dealing with large datasets.

3.1. Checkboxes for Filtering

Checkboxes were added to allow users to filter invoices based on Payment Status — Paid, Due, or Overpaid — enhancing user control without typing complex queries.

3.2. Client Selection and Records Display

A dropdown toggle was introduced along with a dynamic search box, allowing users to easily select a client either by browsing or typing. On the Dashboard, each record in the table represents a unique combination of Client Name, Invoice Reference Number, Invoice Amount, and Date Invoiced.

- If the same Invoice Reference had different Invoice Amounts (even if same or different dates), they were treated as separate records.
- If the same Invoice Reference had same Invoice Amount and same Date Invoiced, it was treated as a single record, and individual payments toward that invoice were shown on the Invoice Details page.
- This grouping ensured that partial payments, full payments, and overpayments were appropriately separated.

3.3. Advanced Filtering after Client Selection

Once a client is selected, an additional Filter Button appears, allowing users to apply more specific filters:

- Invoice Reference (text search)
- Invoice Date Range (start and end dates)
- Invoice Amount Range (min and max amounts)
- Payment Status (Paid, Due, Overpaid) via multi-select checkboxes This additional filtering happens completely on the frontend side without needing to make extra backend API calls, ensuring a faster and smoother user experience.

3.4. Pagination for Invoice Records

After selecting a client, the invoice list is paginated at the bottom of the table.

- The number of records displayed per page dynamically adjusts based on the available screen height to maintain usability across all devices.
- Previous and Next buttons allow easy navigation across pages.

- Whenever the user selects a new client or applies filters for a selected client, pagination resets automatically to Page 1 to correctly display updated results.

3.5. Visualizations

- Credit Score Gauge: Calculates client creditworthiness based on their average days to pay. Clients with faster payments have higher scores, while frequent late payments reduce the score.
- Average Days to Pay Trend Line Chart: Tracks how the client's average payment speed changes across months or years.
- Adaptive Invoice Revenue Bar Chart: Shows total invoiced revenue, dynamically aggregating by Month, Quarter, or Year depending on how many records are present.
  - If there are too many invoices in a year, it automatically adjusts aggregation to avoid chart clutter.
  - If very few data points exist, fallback messages are shown appropriately (e.g., "Insufficient Data" in line charts).
- Invoice Payment Completion Circle: Displays the payment completion percentage for each invoice record.
- Invoice Payment Timeline: Visual timeline showing the sequence of payments made toward a selected invoice.

3.6. Responsiveness and Adaptability

The entire UI was built to be mobile-first responsive, dynamically adjusting layouts for mobiles, tablets, laptops, and desktops. Pagination per page was dynamically adjusted based on screen height to optimize visibility and usability.

4. Implementation

The implementation phase focuses on translating the planned ideas into a clean, scalable, and responsive application using modern web technologies. Both the backend and frontend were structured to support fast, reliable data handling and a smooth user experience.

4.1. Backend Implementation

The backend was developed using Flask (Python) and designed to serve lightweight, JSON-based APIs to power the frontend dashboard.

- API Structure:
    - /api/clients:
        - Returns a list of all available clients.
        - Clients are labeled sequentially as "Client 1", "Client 2", etc., based on their original Client IDs.
    - /api/invoices/{client}:
        - Returns the list of invoices grouped by (Client Name, Clean Invoice Reference, Invoice Amount, Date Invoiced).
        - Supports filtering at the client level based on Payment Status (Paid, Due, Overpaid).
    - /api/invoice-details/{group_id}:
        - Returns detailed payment history for a particular invoice, identified by the Invoice Group ID (e.g., Client2_1).
- Backend Logic Highlights:
    - The dataset was pre-cleaned and standardized using a Data Parsing Notebook (data_parsing.ipynb), which:
        - Extracted clean invoice references even from messy formats.
        - Calculated cumulative payments and pending balances.
        - Detected duplicate payments.
        - Dynamically assigned invoice statuses (Paid, Due, Overpaid).
    - Cleaned data was stored in a cleaned_data.json file, eliminating database dependency and optimizing read performance for this challenge.

4.2. Frontend Implementation

The frontend was built using React.js and was structured into clearly separated pages and reusable components to ensure maintainability and scalability.

- Page Structure:
  - Dashboard (App.jsx): The main landing page where users select clients, view grouped invoices, apply filters, and navigate through paginated records.
  - Invoice Details (InvoiceDetails.jsx): Dedicated page displaying the full payment history and visualizations related to a selected invoice record.
- API Integration:
  - The frontend communicates with the backend using the native JavaScript fetch() API to retrieve invoice data and payment details.
  - After selecting a client, the corresponding invoices are retrieved dynamically from /api/invoices/{client}.
  - Detailed payment information for a specific invoice is fetched from /api/invoice-details/{group_id}.
- State Management:
  - React States were used to handle selected clients, applied filters, and current pagination page.
  - When users navigate from the Dashboard to the Invoice Details page and return, the previously selected filters and pagination state are memorized and restored seamlessly.
  - This ensured smooth user experience without forcing users to reselect their filters or scroll back manually after each navigation.
- Component Modularity:
  - Reusable and modular components were developed for each key functionality:
    - CustomDropdown.jsx (Client Selection with Search)
    - CreditScoreGauge.jsx (Credit Score Visualization)
    - InvoicePaymentCompletionCircle.jsx (Payment Completion Percentage)
    - InvoicePaymentTimeline.jsx (Zigzag Timeline for Payments)
    - AdaptiveInvoiceBarChart.jsx (Revenue Aggregation Chart)
    - AvgDaysToPayTrendLine.jsx (Average Days to Pay Trends)
    - LoadingOrError.jsx (Handles loading indicators and error messages during API calls, improving user feedback during network delays or failures.)

- Handling of responsiveness:
  - CSS Flexbox and Grid layouts were used to ensure responsiveness across devices.
  - Pagination dynamically adjusted based on detected screen height to optimize visibility without overwhelming smaller screens.
- Lazy Loading:
  - Visualization-heavy components like bar charts, trend lines, credit score gauge, and invoice timeline were lazy-loaded using React.lazy() and <Suspense>. This reduced the initial JS bundle size and improved perceived performance.
- Error Boundaries:
  - Widgets were wrapped in a reusable ErrorBoundary component to isolate rendering failures without crashing the entire page.
- Improved Loading UX:
  - A reusable <Spinner /> component was added as a Suspense fallback to indicate loading state for lazy-loaded charts.
- Routing Fix for Vercel:
  - A vercel.json file was added to enable proper routing for React Router, fixing 404 errors on page refresh in deployed environments.
- UptimeRobot for Render Cold Start:
  - A 5-minute HTTP ping monitor was added via UptimeRobot to minimize Render's cold-start delay and ensure backend availability.

5. Testing

Given the strong emphasis on usability, scalability, and data accuracy during development, the testing phase was focused not just on feature validation but on ensuring the application behaved consistently under realistic conditions. Rather than treating testing as a checklist exercise, it was approached from a user's perspective — ensuring that every action felt natural, responsive, and intuitive.

5.1. Real-world Simulation

- Split payments, late payments, overpayments, and duplicate invoices were simulated using the parsed dataset.

- Filters were applied in various combinations to validate dynamic updates without needing page reloads.
- Memory of filters and pagination across navigation events was rigorously tested to ensure users never lost context while exploring invoice details.

5.2. Backend Response Integrity

- APIs were tested against both valid and invalid requests to confirm accurate data retrieval, proper error handling, and graceful fallbacks without breaking the frontend flow.

5.3. Frontend Experience Validation

- Responsiveness across different screen sizes was verified.
- Visualizations were tested for proper adaptability (Monthly vs Quarterly vs Yearly aggregation) depending on invoice volume.
- Loading indicators and error messages were tested under slow network conditions to ensure users always received appropriate feedback during API interactions.

5.4. Performance and Edge Case Handling

- Extreme conditions, such as clients with hundreds of invoices or very sparse histories, were tested to ensure pagination, filtering, and visualizations remained efficient and user-friendly.
- Fallback behaviors (like "Insufficient Data" messages for trend charts) were validated for clients with limited histories.

Ultimately, testing was treated not just as a validation step but as an extension of the application's core philosophy — ensuring quality, intuitiveness, and resilience at every touchpoint.

6. Summary

- The Invoice Payment Dashboard 2025 project successfully transformed raw, unstructured invoice payment data into a clean, intelligent, and user-centric dashboard for meaningful analysis. Through careful preprocessing, strategic backend API development, and modular frontend design, the platform effectively simulates real-world client payment behaviors.
- The project addressed key challenges, including standardizing messy invoice references, grouping client records accurately, maintaining integrity across partial payments, and ensuring full responsiveness across devices. Cumulative calculations and payment behavior visualizations were implemented thoughtfully to handle a variety of real-world edge cases.
- Throughout development, conscious priority was given to quality, accuracy, and usability, ensuring that each feature delivered a seamless and intuitive user experience rather than just basic functionality.
- Overall, the project reflects a strong focus on blending technical precision, practical applicability, and a commitment to user-centered design.