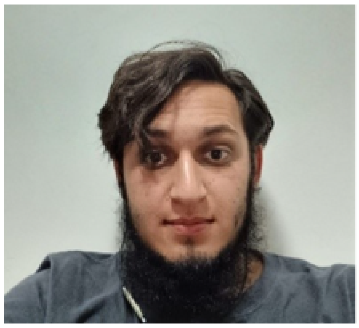
Deliverable 1: Requirements Analysis and Conceptual Design

Project Title: ClearVue Sales Intelligence System

Team: Potatwo

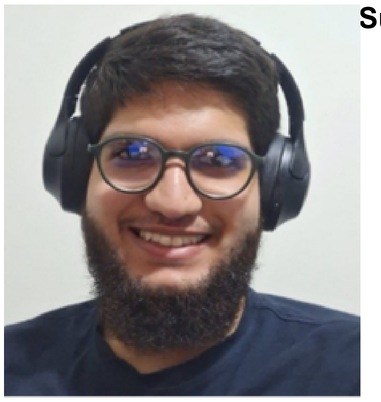
Members:

Picture	Name and Surname	Student Number
	Humaid Ebrahim	**47324732 (Leader)** **47324732@mynwu.ac.za** **072 620 798** **Preliminary Source System Analysis**
	Ruan Thompson	**37613480** **ruan4.t@gmail.com** **078 813 6239** **Hierarchies and Relationships**

	Usaamah Imraan Chothia	47255528 47255528@mynwu.ac.za 063 576 8155 Executive Summary, Requirements Definition Approach
	Raken Belayet	45934843 rakenhossain@gmail.com 074 524 4444 Business Requirements and High-Level Business Objectives
	Janre Oelofse	46678492 46678492@mynwu.ac.za 076 453 9696 Analytical and Information Requirements and Preliminary Success Criteria

Mbuyelo Mboweni	45304718 45304718@mynwu.ac.za 067 007 2746 Conceptual NoSQL Design Of the Proposed Database	

Muhammad Omar	39178544 Muhammadomar241@gmail.com 083 260 3172 Preliminary Source System Analysis	



Sulaiman Sulaman	45427429 sulaimansulaman@gmail.com 071 438 7704 Conceptual NoSQL Design Of the Proposed Database	

Date: 17 October 2025

Module: CMPG321 - Advanced Databases

Table of contents:

1. Executive Summary

ClearVue Ltd. is pursuing to modernize its Business Intelligence (BI) capabilities to help it overcome the current limitations it has in reporting and analytics. The company has a unique fiscal year structure but lacks an effective system for generating a timely and accurate sales report, which comprises of a daily, weekly, monthly, quarterly, and annual periods. This ineffective system hinders strategic decision making.

To combat these challenges the project proposes the design of a NoSQL based Business Intelligence system. This system will provide scalable, flexible, and real-time analytical capabilities. This solution will help leverage a document-oriented data base such as Mongo DB to manage hierarchical and semi structured data and integrate streaming data pipelines for real time insights. Visualization and reporting supported by Business Intelligence dashboard using tools such as Metabase or Tableau.

This system will enhance current sales performance reporting and will also lay the foundation for future supplier focused analytics which will align with ClearVue's strategic direction. By implementing a NoSQL approach, the solution will indicate clear advantages over traditional systems due to the flexibility, scalability and adaptability to the business ever evolving data needs.

2. Requirements Definition Approach

In our approach we will firstly consider the data provided by ClearVue Ltd. By viewing all seventeen file we will be able to view relationships, understand the current system of ClearVue and the hierarchical structure of the company. This will ensure that our analysis is specifically for ClearVue.

We then investigate the required information of the stakeholders in each position of the company. This will help stakeholders use specific data to rectify business failures:

- **Sales Managers** require accurate, timely reporting across daily, weekly, monthly, quarterly, and annual financial periods.
- **Finance Officers** require alignment with ClearVue's unique fiscal year and support for financial aging analysis.
- **Executives** require improved decision-making insights from consolidated reporting and dashboards.
- **Supplier Relationship Managers** require the system to support future supplier-focused analytics.

Furthermore, our secondary research will use NoSQL system for Business Intelligence, in which we will use the data handling to observe where we will be able to correct current problems within the company.

Lastly using AI-tools we can aid our analysis, and it will help us understand problems. The AI-tool will also provide schema suggestions, queries, and documentation. We will use our research knowledge to validate the AI-tools findings, this will ensure accuracy and suitability.

These approaches ensure that the defined requirements are both feasible and aligned with ClearVue's long term objectives.

3. Business Requirements

- BR1: System must support daily, weekly, monthly, quarterly, annual reports to drive performance analysis.
- BR2: Provide supplier-focused analytics for a strategic shift in the future.
- BR3: Support reporting by customer type, region and categories.
- BR4: Integrate streaming data for real-time reporting and transactions.
- BR5: Handle hierarchal and semi-structured data (brands, categories, ranges, styles).
- BR6: Ensure scalability to accommodate growth and evolving data needs.
- BR7: All reports much align with ClearVue's financial calendar.
- BR8: Provide a dash visualizations for executives and managers.
- BR9: Ensure governance (data accuracy, integrity and auditability.)

4. High-Level Business Objectives

- Improve managerial decision-making with timely and accurate reports.
- Future-proof ClearVue’s analytics platform by transitioning from MS Access to a scalable NoSQL solution
- Improve customer and product insights to optimize marketing and sales strategies.
- Support ClearVue’s supplier-focused analytics strategy by enabling supplier performance analytics.
- Enable executives to make faster, data-driven decisions through consolidated BI dashboards.

5. Analytical and Information Requirements

ClearVue needs a set of reports and analyses that will help management, finance, and executives make better decisions. These are the main requirements we identified

- **Daily Sales Report**
  Transactions should be grouped by customer type (retail, wholesale, corporate).
  Why: This shows where the money is coming from on a day to day basis and helps managers pick up on unusual activity or sudden drops in sales.

- **Monthly Product Trends**
  A report showing the top 10 products by revenue in each category and range.
  Why: This helps ClearVue see which products are performing best, plan future stock levels, and guide marketing or promotions.

- **Customer Analysis**
  Reports on customer buying behaviour such as frequency of purchases, total spend, and settlement patterns.
  Why: This makes it easier to see who the loyal and high value customers are, while also identifying customers who might pose credit risks.

- **Supplier Report**
  Purchases per supplier with the total spend.
  Why: This gives visibility into supplier performance and allows for better contract negotiations and planning for supplier related risks.

- **Age Analysis Report**
  Outstanding amounts owed by customers broken down into 30, 60, and 90+ day periods.
  Why: This is essential for finance to manage overdue accounts, make sure payments are followed up, and keep cash flow healthy.

- **Profitability and Margin Analysis**
  Gross profit and net margins by product, category, or customer type.
  Why: This helps management focus on products or areas that generate the most profit and cut back on the ones that don't.

- **Inventory Movement Analysis**
  Tracks inflows and outflows of stock including shrinkage and wastage.
  Why: This helps reduce stock problems such as overstocking, stock-outs, and unnecessary carrying costs.

6. Preliminary Source System Analysis

The preliminary analysis examined the 19 data files supplied by ClearVue Ltd. The data includes include sales transactions, purchases, payments, customers, products, suppliers, and related data. The aim of this step was to uncover the core data structures, relationships, and hierarchies present within the current system.

Table	Purpose	Info
Age Analysis	Tracks how much each customer owes and the age of the debt.	CUSTOMER_NUMBER: The customers id. FIN_PERIOD: financial period. year and month TOTAL_DUE: the total amount owed. AMT Columns: How much the debt is overdue by.
Customer Account Parameters	Shows the type of customer account	CUSTOMER_NUMBER: The customers id. PARAMETER: Type of account
Customer Categories	Lookup table	CCAT_CODE: Numeric code CCAT_DESC: Describes the code
Customer Regions	Region Lookup Table	REGION_CODE: The regions id. REGION_DESC: Locations and account statuses
Customer	Stores basic information for customers	CUSTOMER_NUMBER: The customers id. CCAT_CODE: Links to the Customer category tables. REGION_CODE: Links to the Region Table. REP_CODE: Links to the Representative table. SETTLE_TERMS: Customers settlement code. NORMAL_PAYTERMS: Payment terms in days. DISCOUNT: Discount amount

		CREDIT_LIMIT: Max credit allowed for customer
Payment Header	Customer deposit references	CUSTOMER_NUMBER: Customer who made the deposit. DEPOSIT_REF: deposit reference number
Payment Lines	Detailed description of payments	CUSTOMER_NUMBER: Customer who made the deposit. FIN_PERIOD: Fincancial period which the payment occured. DEPOSIT_DATE: Date of the deposit. DEPOSIT_REF: deposit reference number, links to payment header table. BANK_AMT: Amount deposited. DISCOUNT: The discount applied. TOT_PAYMENT: Rounded BANK_AMT
Product Brands	Product Brand lookup table	PRODBRA_CODE: Numeric code for brand. PRODBRA_DESCR: Brand description
Product Category	Products according to brand and range	PRODCAT_CODE: The product category code. PRODCAT_DESC: Description of the product category. BRAND_CODE: Links to product brand table. PRAN_CODE: Links to the product range table.

Product Range	The different types of products offered	PRAN_CODE: The identifying ID of the range PRAN_DESC: The description of the ranges
Product Styles	Information stored about different styling and information about a product	INVENTORY_CODE: The ID of the product GENDER: The gender the product is for MATERIAL: The material of the product STYLE: The specific style type of the product COLOUR: Colour of the product BRANDING: Product brand QUAL_PROBS: Quality problems
Products	Base information for a product	INVENTORY_CODE: The code for the inventory item PRODCAT_CODE: The product category code LAST_COST: The last cost of the product STOCK_IND: If the current item is being stocked
Purchase Header	The base information of a stock order	SUPPLIER_CODE: The ID code for the stock supplier PURCH_DOC_NO: The ID code for the stock order PURCH_DATE: The date the order was created
Purchases Lines	Details of a purchase and products purchased in a stock order	PURCH_DOC_NO: The ID of the purchase header INVENTORY_CODE: The product ID QUANTITY: The amount of product purchased UNIT_COST_PRICE: The price of a single unit at cost

		TOTAL_LINE_COST: The total cost for the product being ordered.
Representatives	The information stored about each sales representative.	REP_CODE: The unique Identifier for a sales rep REP_DESC: Description of the rep COMM_METHOD: The method used to calculate the commission amount using the commission percentage COMMISSION: The percentage of the commission
Sales Header	It contains all the information of a sale. It is essentially an order description	DOC_NUMBER: The identifier code for the sale TRANSTYPE_CODE: The identification code for the transaction type REP_CODE: The code for the representative who made the sale. CUSTOMER_NUMBER: The code for the customer buying the items TRANS_DATE: The date the transaction occurred FIN_PERIOD: The financial period the transaction took place in.
Sales Line	Stores information about specific sales on products	DOC_CODE: The sales header identifier code INVENTORY_CODE: The code for the product being sold QUANTITY: The amount of product sold UNIT_SELL_PRICE: The selling price of the product TOTAL_LINE_PRICE: The total cost

		LAST_COST: The previous cost of the product
Suppliers	Information stored about suppliers and their credit	SUPPLIER_CODE: The identifier code for the supplier SUPPLIER_DESC: name or short description of the supplier EXCLSV: States if the supplier is exclusive or not. NORMAL_PAYTERMS: The time in days their debt is payable over CREDIT_LIMIT: The amount of credit the supplier offers
Trans Types	The type of transaction occurring	TRANSTYPE_CODE: The identifier for the transaction type TRANSTYPE_DESC: Name of transaction occurring

Hierarchies and Relationships

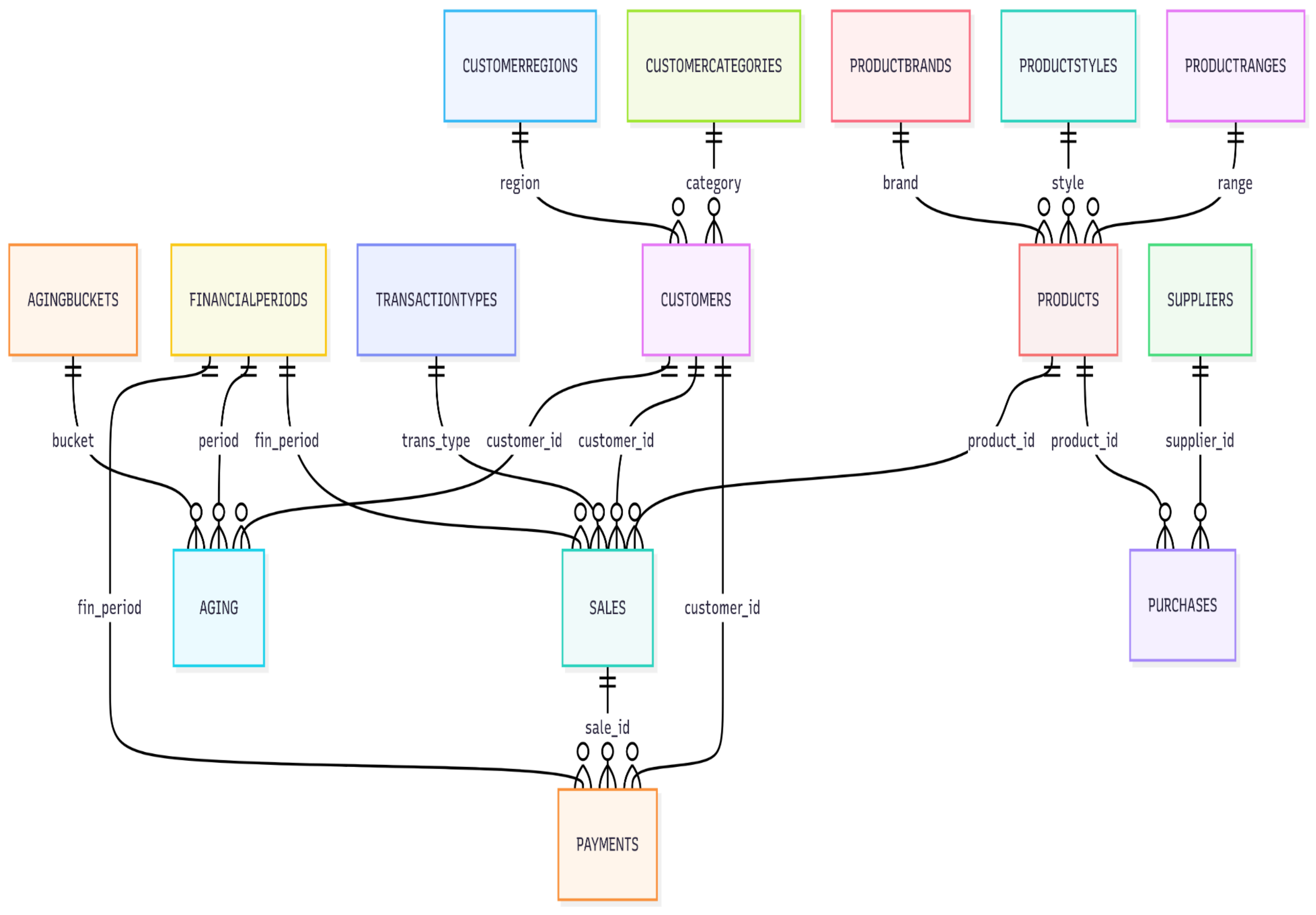
Entity Relationships

Entity/Table	**Relationship identifiers**	**Related To**	**Cardinality**
Customer	CUSTOMER_NUMBER CCAT_CODE, REGION_CODE, REP_CODE	Customer Categories, Customer Regions, Representatives, Sales Header, Payment Header, Age Analysis	1 Customer -> M Sales; 1 Customer -> M Payments; 1 Customer -> M Age Analysis Records
Customer Categories	CCAT_CODE	Customer	

			1 Category -> M Customers
Customer Regions	REGION_CODE	Customer	1 Region -> M Customers
Customer Account Parameters	CUSTOMER_NUMBER + PARAMETER	Customer	1 Customer -> M Account Parameters
Representatives	REP_CODE	Sales Header, Customer	1 Representative -> M Customers; 1 Representative -> M Sales
Sales Header	DOC_CODE CUSTOMER_NUMBER , REP_CODE, TRANSTYPE_CODE, FIN_PERIOD	Customer, Representatives, Trans Types, Sales Line, Time	1 Sale (Header) -> M Sale Lines; 1 Customer -> M Sales
Sales Line	DOC_CODE + Line Number	Sales Header, Products	1 Sale Header -> M Sale Lines; 1 Product -> M Sales Lines
Trans Types	TRANSTYPE_CODE	Sales Header	1 Transaction Type -> M Sales
Payment Header	DEPOSIT_REF CUSTOMER_NUMBER	Customer, Payment Lines	1 Customer -> M Payment Headers; 1 Payment Header -> M Payment Lines
Payment Lines	DEPOSIT_REF + Line Number FIN_PERIOD	Payment Header, Time	1 Payment Header -> M Payment Lines
Age Analysis	CUSTOMER_NUMBER + FIN_PERIOD	Customer, Time	1 Customer -> M Age Analysis Records (one per financial period)
Products	INVENTORY_CODE PRODCAT_CODE	Sales Line, Purchase Line, Product Category, Product Styles	1 Product -> M Sales Lines; 1 Product -> M Purchase Lines
Product Styles	INVENTORY_CODE	Products	1 Style -> M Products (or in some designs, 1 Product -> 1 Style)

Product Category	PRODCAT_CODE BRAND_CODE, PRAN_CODE	Products, Product Brands, Product Range	1 Category -> M Products
Product Brands	BRAND_CODE (PRODBRA_CODE)	Product Category	1 Brand -> M Categories; indirectly 1 Brand -> M Products
Product Range	PRAN_CODE	Product Category	1 Range -> M Categories
Purchase Header	PURCH_DOC_NO SUPPLIER_CODE, FIN_PERIOD	Suppliers, Purchase Lines, Time	1 Purchase Header -> M Purchase Lines; 1 Supplier -> M Purchases
Purchase Lines	PURCH_DOC_NO + Line Number	Purchase Header, Products	1 Purchase Header -> M Purchase Lines; 1 Product -> M Purchase Lines
Suppliers	SUPPLIER_CODE	Purchase Header	1 Supplier -> M Purchase Headers

Simple Schema Diagram:


CUSTOMERREGIONS
CUSTOMERCATEGORIES
PRODUCTBRANDS
PRODUCTSTYLES
PRODUCTRANGES
region
category
brand
style
range
AGINGBUCKETS
FINANCIALPERIODS
TRANSACTIONTYPES
CUSTOMERS
PRODUCTS
SUPPLIERS
bucket
period
fin_period
trans_type
customer_id
customer_id
product_id
product_id
supplier_id
fin_period
AGING
SALES
customer_id
PURCHASES
sale_id
PAYMENTS

Description:

The above Simple Schema Diagram provides a high-level view of the proposed MongoDB NoSQL schema, it highlights the 8 core collections (Customers, Suppliers, Sales, Payments, Purchases, Products Aging and Financial Periods) along with the Reference collections (TransactionTypes, CustomerCategories, CustomerRegions, ProductBrands, ProductRanges, ProductStyles, and AgingBuckets). This shows which entities are embedded (e.g., sales lines inside Sales, purchase lines inside Purchases, weeks inside FinancialPeriods) and which are referenced (e.g., Customers, Products, Suppliers). This balance avoids duplication while ensuring fast queries and scalable BI reporting.

7. Preliminary Success Criteria

To know when this project can be considered complete and successful, the following criteria will be used:

- **SC1:** The system can generate the core sales and financial reports (daily, monthly, supplier, and ageing) according to ClearVue’s financial calendar.
- **SC2:** The database schema supports reporting across different levels such as product hierarchy, customer type, and region.
- **SC3:** Queries should return results quickly (within about 5 seconds) even when working with large amounts of data.
- **SC4:** A BI dashboard is available to visualise key reports such as sales trends, profitability, and ageing.
- **SC5:** Stakeholders (managers, finance, executives) are satisfied with the accuracy and relevance of the reports, with positive feedback after testing.
- **SC6:** The system connects properly to source data and doesn’t require heavy manual work to keep it updated.
- **SC7:** Reports meet audit and governance standards so that data is reliable and traceable.
- **SC8:** Users are trained and able to run their own reports confidently without needing developer support.

If these success criteria are met it means ClearVue’s reporting system will be delivering what the business needs and the project can be marked as complete.

8. CONCEPTUAL NOSQL DESIGN

We propose a document-oriented schema using MongoDB, a NoSQL database better used to handle hierarchical and semi-structured data The main collections include Sales, Customers, Products, Suppliers, Payments, Purchases, FinancialPeriods, and Aging. Hierarchical data such as product categories, ranges, and styles are embedded within the Products collection, while financial weeks are embedded within FinancialPeriods.

In addition to the 8 main collections, we propose 7 reference collections. These are small lookup collections that reduce duplication, enforce consistency, and support BI reporting by standardizing commonly reused values.

NoSQL fits in by allowing embedding and referencing of data without rigid schemas, unlike traditional relational databases. This means ClearVue can model its unique fiscal calendar, store hierarchical product data efficiently and add new business requirements (e.g. supplier analytics) without schema changes.

This model reduces the need for costly joins and data duplication, while supporting ClearVue's requirements for fast, flexible BI reporting. By balancing embedding (for data always queried together) and referencing (for reusable entities such as customers and suppliers), the schema ensures scalability, accuracy, and alignment with ClearVue's unique financial calendar and future supplier-focused analytics strategy.

Core Collections:

Example Sales Collection Document:

```
{  "_id": "SALE567",  "customer_id": "CUST12345",  "rep": { "id": "REP07", "name":
"John Smith", "commission": 10 },  "trans_type": "Tax Invoice",  "trans_date":
"2025-05-01",  "fin_period": "2025-M07",  "lines": [ { "product_id":
"PROD789","quantity": 3,      "unit_price": 900, "total_line_cost": 2700 }, { "product_id":
"PROD456","quantity": 1,      "unit_price": 1200,"total_line_cost": 1200 }
],"total_amount": 3900,"payment_status": "Partially Paid"}
```

Why is it suitable:

The Sales collection is for capturing transactions with embedded line items (products, quantities and prices) and referencing customers and products for scalability. This is because sales documents are always queried with their line details, so embedded storage is more efficient than separate header and line collections. Customers and products are referenced not duplicated so master data can be updated without rewriting history. Storing the financial period in each sale also makes it easy to aggregate by month, quarter or year which is what ClearVue Ltd. need to do for their BI reports according to their own financial calendar. This balance of embedding and referencing gives both query performance and data integrity.

Example Suppliers Collection Document:

```
{  "_id": "SUPP001",  "name": "Global Textiles Ltd",  "contact": { "phone": "+27 11 555
4321",    "email": "info@globaltextiles.co.za", "address": "12 Industrial Rd,
Johannesburg"  },  "exclusive": true,  "normal_payterms": 60,  "credit_limit": 50000,
"status": "Active"}
```

Why is it suitable: The Suppliers collection holds all supplier master data such as names, contact details, credit limits and payment terms in one place. This is because supplier data is reused across multiple contexts (products and purchases) and needs to be updated without duplication. By centralising supplier information and linking it to related purchases and products ClearVue can manage supplier relationships and avoid duplication of data across collections. This also sets the system up for future supplier analytics which is a strategic objective in the project requirements. The design is flexible, reduces maintenance overhead and allows supplier performance to be monitored through BI dashboards.

Example Aging Collection Document:

```
{  "_id": "AGING2025-07-CUST12345",  "customer_id": "CUST12345",  "period":
"2025-M07",  "generated_on": "2025-07-31",  "total_due": 12000,  "amt_current":
5000,  "amt_30_days": 4000,  "amt_60_days": 3000,  "amt_90_days": 0}
```

Why is it suitable:

The Aging collection is good because it gives a customer level view of outstanding balances broken into buckets like current, 30, 60, 90+. These are split according to standard financial practice used in most credit control and accounts receivable system. As appose to embedding aging within individual sales, this way avoids duplication, and aging is a derived financial metric based on multiple sales and payments. By storing each snapshot against a financial period, ClearVue can track historical overdue debt and compare customer behaviour across months or quarters. Customer IDs and financial periods reference ensures strong integration with other collections and allows for accurate, scalable BI reporting for credit risk.

Example FinancialPeriods Collection Document:

```
{  "_id": "2025-M07",  "start_date": "2025-06-28",  "end_date": "2025-07-25",
"fiscal_year": "2025",  "quarter": "Q2",  "weeks": [ { "week_no": 1, "start": "2025-06-28",
"end": "2025-07-04" }, { "week_no": 2, "start": "2025-07-05", "end": "2025-07-11" },    {
"week_no": 3, "start": "2025-07-12", "end": "2025-07-18" }, { "week_no": 4, "start":
"2025-07-19", "end": "2025-07-25" }  ] }
```

Why is it suitable:

The FinancialPeriods collection models ClearVue Ltd’s non-standard fiscal calendar, each document represents a financial month and embeds the corresponding weeks. This is particularly useful as relational “date dimension” tables struggle to model custom periods like ClearVue’s months which run from the last Saturday of one month to the last Friday of the next. By storing each financial period as a document with start and end dates, fiscal year and quarter information, reports can group transactions by business defined periods. Embedding weeks inside each period supports weekly sales and payment rollups without needing extra tables. This way the BI system produces financial reports aligned to ClearVue’s accounting rules.

Example Customer Collection Document:

```
{  "_id": "CUST12345",  "name": "ABC Fashion",  "category": { "id": "CAT01", "desc":
"Speciality Stores" },  "region": { "id": "REG05", "desc": "Cape Town" },  "account": {
"status": "Open","type": "Credit", "credit_limit": 20000, "payment_terms": "30 days" },
"representative": { "id": "REP07", "name": "John Smith" } }
```

Why is it suitable:

The Customer's collection is suitable because it centralizes all master data about clients, including categories, regions, account terms, and sales representatives. By separating customers into their own collection, ClearVue avoids duplication across sales and payments. Embedding account attributes such as credit limits and payment terms ensures quick retrieval of financial details, while referencing customers in sales and payments maintains consistency as the database scales. This design provides ClearVue with a consolidated view of customer performance, supporting both credit management and BI analysis.

Example Product Collection Document:

```
{  "_id": "PROD789", "name": "Air Jordan Leather Sneaker", "brand": { "id": "BR01",
"desc": "Nike" }, "category": { "id": "CAT23", "desc": "Shoes" },  "range": { "id":
"RNG02", "desc": "Running" },  "style": { "gender": "Male", "material": "Leather",
"colour": "Black", "design": "Streetwear" }, "inventory_code": "INV123", "last_cost":
600, "stock_ind": true, "supplier_id": "SUPP001"}
```

Why is it suitable:

The Products collection is good because it’s the central store for all product data, including brand, category, range, style and cost. Products are referenced in both sales and purchases so keeping them in their own collection avoids duplication and keeps everything accurate across transactions. Embedding hierarchical data like brand, category and style makes BI reporting flexible so you can see things like best performing product ranges or top selling categories. Linking products to suppliers keeps traceability without bloating transaction records. This balances efficiency and scalability so product data remains consistent and supports detailed reporting for sales and inventory management.

Example Payments Collection:

```
{  "_id": "PAYM001",  "customer_id": "CUST12345",  "sale_id": "SALE567",
"deposit_ref": "DEP20250501",  "lines": [ { "fin_period": "2025-M07",
"deposit_date": "2025-05-01", "bank_amt": 1500, "discount": 100, "tot_payment": 1400
}  ] }
```

Why is it suitable:

The Payments collection is good because it tracks all money received from customers, including deposits, discounts and partial payments. A single sale can have multiple payments so separating payments into their own collection avoids duplication and allows for installment payments or multiple payment methods. By referencing sales and customers ClearVue keeps consistency across transactions without repeating details. This makes it easy to generate BI reports on cash flow, outstanding balances and payment performance. It also helps with credit management by giving a history of payment behaviour for each customer across different time periods.

Example Purchases Collection:

```
{  "_id": "PURCH333",  "supplier_id": "SUPP001",  "purch_date": "2025-06-15",  "lines":
[  { "product_id": "PROD789", "quantity": 200, "unit_cost_price": 400,
"total_line_cost": 80000 } ] }
```

Why is it suitable:

The Purchases collection is good because it records products bought from suppliers, with line items (product, quantity, unit cost) embedded in each purchase order. This mirrors the Sales collection, so the system is consistent and easy to query. Suppliers and products are referenced not duplicated so it’s scalable and reduces maintenance overhead. Embedding purchased items allows full orders to be retrieved in one query which supports BI reporting on supplier costs, stock movements and purchase history. This allows ClearVue to track expenses, inventory replenishment and generate accurate supplier analytics for decision making.

Reference Collections:

Example TransactionTypes (Sales) Collection:

```
{  "_id": "TRX01",  "code": "TAX_INV",  "description": "Tax Invoice" }
```

Why is it suitable:

TransactionTypes ensures consistent labelling of transactions like Tax Invoices, Credit Notes or Debit Notes. By storing these in a reference collection ClearVue avoids free-text inconsistencies (e.g. “Tax Inv” vs “Invoice”). This design keeps the Sales collection lean and ensures reporting by transaction type is accurate and standardised. It also makes it easier to add new transaction types in the future without changing existing sales records.

Example CustomerCategories Collection:

```
{  "_id": "CAT01",  "description": "Retail" }
```

Why is it suitable:

CustomerCategories classifies customers into groups like Retail, Wholesale or VIP. By storing these in a lookup collection ClearVue avoids repeated text entries across thousands of customers which would otherwise create duplication and inconsistencies. This structure allows ClearVue to segment customers for BI analysis like comparing revenue between wholesale and retail customers. It also allows for scalability by adding new categories without changing the customer collection.

Example CustomerRegions Collection:

```
{  "_id": "REG05",  "description": "Cape Town" }
```

Why is it suitable:

CustomerRegions defines geographical areas like Gauteng, Western Cape or Cape Town. By using a reference instead of storing regions in customers ClearVue prevents inconsistent entries like “CPT” vs “Cape Town”. This ensures clean and reliable reporting by region which is a key requirement for BI dashboards. It also allows for future expansion like adding regional metadata (population, economic zone) without changing customer records.

Example ProductBrands Collection:

```
{  "_id": "BR01",  "description": "Nike" }
```

Why is it suitable:

ProductBrands centralises brand information like Nike, Adidas or Samsung. Without these, brands would be duplicated across thousands of product records making updates and reporting inconsistent. By referencing brands ClearVue can run BI reports on sales performance by brand while keeping product documents lean. This separation also allows for flexibility in updating brand descriptions or grouping brands under larger parent companies if required.

Example ProductRanges Collection:

```
{  "_id": "RNG02",  "description": "Running" }
```

Why is it suitable:

ProductRanges define product groupings like Running, Lifestyle or Premium. Storing ranges separately reduces duplication and provides consistency across the product catalog. This design allows ClearVue to analyse product performance at the range level which is a common requirement in BI reporting. It also allows managers to adjust ranges or add new ones without changing existing product documents.

Example ProductSyles Collection:

```
{  "_id": "STY03",  "gender": "Male",  "material": "Leather",  "colour": "Black",  "design":
"Streetwear" }
```

Why is it suitable:

ProductStyles capture attributes like gender, material, color and design. By separating these attributes ClearVue prevents free-text inconsistencies (e.g. “Blk” vs “Black”) and ensures reporting across style attributes is reliable. This design allows for marketing and trend analysis by tracking performance by material or color across product lines. It also keeps the Products collection lean as products simply reference the standardized style.

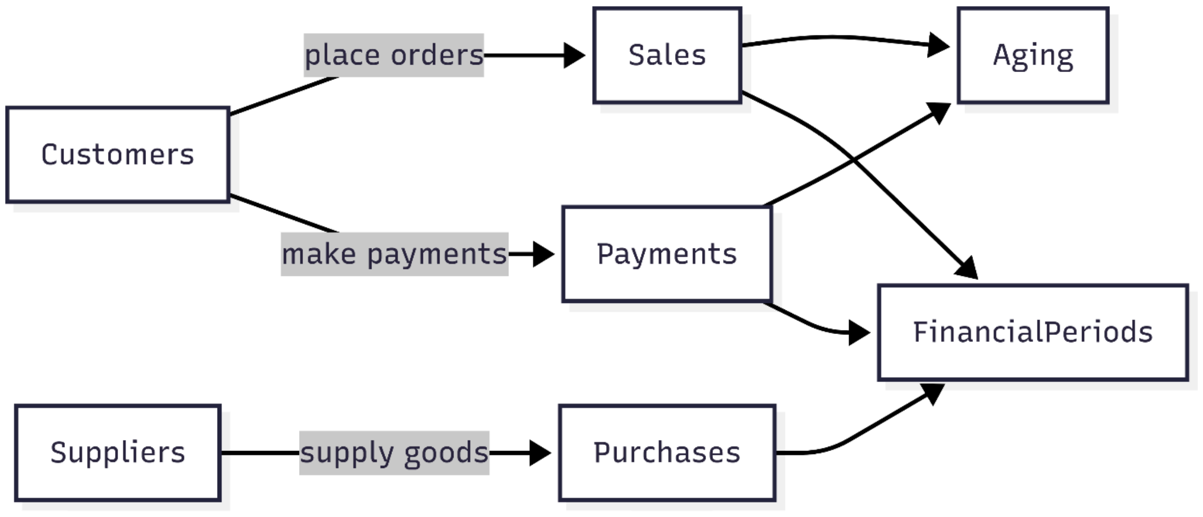
Example AgingBuckets Collection:

```
{  "_id": "BKT01",  "range": "0–30 days",  "description": "Current" }
```

Why is it suitable:

AgingBuckets defines overdue categories like 0–30, 31–60, 61–90 and 90+ days. Standardising these buckets ensures all age calculations across the system use the same definitions. This avoids BI reporting discrepancies and allows for accurate credit risk management. It also allows for future expansion by adding new buckets without changing existing ageing records.

High Level Data Flow Diagram:



Description:

The above High Level Data Flow Diagram describes the flow of business data in the proposed NoSQL BI System for ClearVue. It provides how the business activities map into the NoSQL collections. Customers place orders which go into the Sales collection and pay which goes into the Payments collection. Suppliers supply which goes into the Purchases collection. Sales feed into the Aging collection to calculate outstanding balances, and Sales, Payments and Purchases are linked to the FinancialPeriods collection which ties all transactions to ClearVue's fiscal calendar. This way all business activities – customer sales, payments, supplier purchases and debt aging – are tracked and reported across the required financial periods.

No SQL specifics:

MongoDB is perfect for ClearVue because it supports a document-oriented approach that balances embedding and referencing for efficiency and flexibility. Line items are embedded inside Sales and Purchases since they are always queried together, while Customers, Suppliers and Products are referenced to avoid duplication. Reference collections like TransactionTypes, Regions, Brands and AgingBuckets standardize commonly used values across transactions. To support BI reporting indexes will be created on fields like customer_id, supplier_id, fin_period and product_id so we can do fast lookups and aggregations. We will shard Sales and Payments by financial period or customer to scale, and the Aging collection will capture monthly snapshots of outstanding balances for historical analysis without having to recalculate everything. MongoDB's flexible schema design will also allow us to add new categories, payment types or supplier attributes without having to do a disruptive schema change, so the system will be scalable and future proof.

Sample Queries Examples:

Daily Sales Based on Region:

```
db.Sales.aggregate([  { $match: { trans_date: { $gte: ISODate("2025-07-01"), $lte:
ISODate("2025-07-01") } } },  { $lookup: { from: "Customers", localField:
"customer_id", foreignField: "_id", as: "cust" } },  { $unwind: "$cust" },  {
$group: { _id: "$cust.region.desc", total_sales: { $sum: "$total_amount" } } } ] )
```

The above sample query gets the total sales for a day by **region**. Management can see which regions are performing best daily.

Top 10 Products by Revenue per Category:

```
db.Sales.aggregate([  { $match: { fin_period: "2025-M07" } },  { $unwind: "$lines"
},  { $lookup: { from: "Products", localField: "lines.product_id", foreignField:
"_id", as: "prod" } },  { $unwind: "$prod" },  { $group: { _id:
$prod.category.desc", revenue: { $sum: "$lines.total_line_cost" } } },  { $sort: {
revenue: -1 } },  { $limit: 10 }])
```

The above sample query gets the **top 10 products** by revenue for each category for a given period. Help to see which categories and products are making the most revenue.

Customer Payment History:

```
db.Payments.aggregate([  { $match: { customer_id: "CUST12345" } },
  { $unwind: "$lines" },  { $project: { deposit_date: "$lines.deposit_date",
amount: "$lines.tot_payment", discount: "$lines.discount" } },  { $sort: {
deposit_date: -1 } }]
)
```

The above sample query shows all payments for a customer, including deposit dates, amounts paid, and discounts applied. See a customer's **payment habits** and credit history.

Customer Aging Report:

```
db.Aging.find(  { period: "2025-M07" },  { customer_id: 1, total_due: 1,
amt_current: 1, amt_30_days: 1, amt_60_days: 1, amt_90_days: 1 } )
```

The above sample query gets a list of outstanding balances by customer, broken down into aging buckets (current, 30 days, 60 days, 90+ days) for a given period. Helps finance to track overdue accounts and cash flow.

Suppliers spend report:

```
db.Purchases.aggregate([   { $unwind: "$lines" },  { $group: { _id: "$supplier_id",
total_spent: { $sum: "$lines.total_line_cost" } } },  { $sort: { total_spent: -1 } } ])
```

The above sample query gets the total spend per supplier, so procurement and finance can see supplier costs and negotiate contracts or track supplier performance.

Monthly Sales Trend:

```
db.Sales.aggregate([
  { $group: { _id: "$fin_period", total_sales: { $sum: "$total_amount" } } },
  { $sort: { _id: 1 } }
])
```

The above sample query summarizes sales by month, aligned to ClearVue’s fiscal calendar (last Saturday to last Friday). Help with monthly, quarterly and annual trend analysis for management decisions.

How the source file links to the collections:

Core Collections	Source Files
Customers	Customers.xlsx Customer Account Parameters.xlsx Customer Categories.xlsx Customer Regions.xlsx Representatives.xlsx
Sales	Sales Header.xlsx Sales Line.xlsx Trans Types.xlsx
Payments	Payment Header.xlsx Payment Lines.xlsx
Products	Products.xlsx Product Category.xlsx Product Range.xlsx Product Styles.xlsx Product Brands.xlsx
Suppliers	Suppliers.xlsx
Purchases	Purchase Header.xlsx Purchase Lines.xlsx
Aging	Age Analysis.xlsx
Financial Periods	(Created due to ClearVues fiscal Calendar Rules)

9. AI Usage Log

ChatGPT was used for Relationship tables to refine the wording and to check if there were any repetitive tasks that we could have made better

ChatGPT was used to refine the wording of the **Business Requirements** and **High-Level Business Requirements**. Drafts were made with core ideas (reporting cycles, scalability. Supplier analytics) and ChatGPT were used to improve clarity and suggested a more professional tone such. Furthermore, it pointed out missing requirements such as alignment with ClearVue's calendar, BI dashboard support, and data governance.

ChatGPT was used to research the general Schema to assist with syntax, best practices and structuring examples of the design in MongoDB. The final schema, this includes main core collections as well as reference collections, was submitted to ChatGPT and asked to prompt whether the proposed solution fitted with ClearVue's business requirements. AI's role was limited to just providing examples and explanations, used to further understanding of example code, all final MongoDB NoSQL collections and explanations remain the work of the team members

Preliminary Source System Analysis - Used AI to assist in deciphering spreadsheets.

https://chatgpt.com/share/68b0758f-9920-8003-988c-89199bb9acd2

https://chatgpt.com/share/68b07667-8844-8006-847f-1575eaca091c

https://chatgpt.com/share/68b09c54-2ae8-8009-831e-7e9186aca50b

https://chatgpt.com/share/68b1c1cd-fa98-8007-a276-dda0186015c5

https://chatgpt.com/share/68a77665-9c98-800b-a022-565759a2be33

https://chatgpt.com/share/68b07fc9-31b4-800b-8210-1f90e245b988

10. Bibliography / References

MongoDB. 2024. MongoDB Documentation. https://github.com/mongodb/docs-bi-connector/blob/DOCSP-3279/source/index.txt. https://www.mongodb.com/docs/ Date of access: 22 Aug. 2025.

OpenAI. 2025. ChatGPT. ChatGPT. OpenAI. https://chatgpt.com/ Date of access: 20 Aug. 2025.

DELIVERABLE 2A: Physical Build and Implementation

1. Understanding of ClearVue Ltd.’s needs

1.1 Introduction

ClearVue Ltd.’s. is a retail company that manages and analyses large volumes of sales information. To enhance its competitiveness and data driven nature of the company, the company requires a modern Business Intelligence (BI) solution. This solution increases its decision-making capabilities. The goal of this system is to provide management with accurate and flexible access to sales performance insights. This will allow preparation for the company in the future for supplier focused analytics, with the application of NoSQL based BI approach.

1.2 Business Problem

ClearVue Ltd.’s greatest challenge to date is its inability to analyse and report sales data across various times. The company’s current relational database structure is based on Microsoft Access, which places significant limitations on its ability to generate daily, weekly, monthly, quarterly, and annual reports that align with its unique fiscal year. ClearVue Ltd. operates on a unique financial calendar that runs from the last Saturday of one month to the last Friday of the next. This unconventional reporting cycle creates complications for traditional relational reporting systems. As a result, the company’s system lacks real-time visibility into key performance indicators such as sales trends, customer behaviour, and product performance. An additional complication is that the current systems do not efficiently accommodate hierarchical or semi-structured data such as product categories, brands, and supplier relationships. These restrictions limit the organization’s ability to expand into supplier analytics, a key future business goal. Furthermore, the lack of streaming data integration imposes constraints on timely decision-making, especially in monitoring ongoing payment transactions and sales activities.

1.3 Key Business Requirements

BR1: System must support daily, weekly, monthly, quarterly, and annual reports to drive performance analysis.

BR2: Provide supplier-focused analytics for a strategic shift in the future.

BR3: Support reporting by customer type, region, and categories.

BR4: Integrate streaming data for real-time reporting and transactions.

BR5: Manage hierarchical and semi-structured data (brands, categories, ranges, styles).

BR6: Ensure scalability to accommodate growth and evolving data needs.
BR7: All reports must align with ClearVue's financial calendar.
BR8: Provide dashboard visualizations for executives and managers.
BR9: Ensure data governance, accuracy, integrity, and auditability.

1.4 Underlying Business Goals

The primary objective behind these requirements is to transform ClearVue Ltd.'s reporting environment into a scalable and adaptable Business Intelligence platform. This new system will enable management to make data-driven decisions supported by accurate and timely sales information. Adopting a NoSQL-based approach will improve the system's flexibility, allowing ClearVue Ltd. to integrate new data types and support supplier analytics in the future. Through this transition, the company aims to align with its broader goal of establishing a modern, sustainable, and insight-driven business model.

1.5 Summary

ClearVue Ltd. requires a BI solution that addresses its current reporting inefficiencies while enabling future expansion into supplier analytics. A NoSQL-based BI system provides the ideal foundation for achieving these goals by supporting hierarchical data, real-time reporting, and scalable analytics.

2. Approach and Evaluation of the ETL Process

2.1 Approach

The development of our data management solution followed a structured system design methodology, implemented through four key phases: (1) Analysis, (2) Design, (3) Implementation, and (4) Evaluation. This framework ensured a methodical progression from raw data assessment to a robust, query-ready MongoDB architecture integrated with real-time data streaming and BI reporting.

Phase 1: Analysis

A thorough examination of the nineteen ClearVue data files was the first step in the procedure. High levels of heterogeneity, inconsistency, and irregular formatting including duplicate entries, missing or null values, and disjointed data structures were present in these datasets.

AI was used to uncover dependencies, correlations, and interactions amongst tables in order to speed up comprehension. This served as a guide for defining the cleaning and transformation rules.

This stage involved identifying and classifying data domains for additional modeling, including sales, payments, purchases, customers, products, and representatives. The subsequent phase's conceptual schema design was influenced by the results of this stage.

Phase 2: Design

Based on the analytical findings, a conceptual NoSQL data model was developed to represent the data in a flexible, scalable format compatible with MongoDB, chosen for its adaptability to semi-structured and nested data.

Key design principles included:

- **Embedding:** Line-level details (e.g., sales_lines, payment_lines, and purchase_lines) were embedded within their respective header documents to preserve data hierarchy and enhance query efficiency.

- **Denormalization**: Common entities such as *customers* and *products* were flattened to minimize joins and improve performance.

- **Separation of Independent Entities**: *Suppliers* and *aging data* were maintained in distinct collections due to limited relational dependencies.

- **Analytical Views:** Flattened “view” collections (SALES_VIEW, PURCHASES_VIEW, and PAYMENT_VIEW) were introduced to support BI integration and simplify aggregations.

Throughout the design phase, MongoDB Compass played a crucial role in validating schema integrity and testing aggregation logic. Using MongoDB aggregation pipelines, we verified the correctness of joins, computed fiscal period mappings, and tested transformation logic directly within the Compass environment before deployment.

Phase 3: Implementation

The implementation phase adopted a hybrid Extract–Transform–Load (ETL) and streaming data integration approach.

1. **Extraction & Cleaning**:
   Raw data was processed in Python using pandas to handle null values, remove duplicates, and standardize formats. This step also incorporated AI-driven rules to handle irregularities in naming conventions and incomplete data fields.

2. **Transformation**:
   A key transformation task involved aligning all temporal data with the client’s non-standard financial year, which begins at the end of February. Each fiscal month was defined from the last Saturday of the previous month to the last Friday of the current month, requiring calculated adjustments across all date fields.

3. These transformations were tested and verified in MongoDB Compass using aggregate functions, ensuring fiscal alignment across all collections before final data loading.

4. **Loading & Streaming**:
   Cleaned data was serialized to JSON and streamed into Apache Kafka. Kafka producers transmitted datasets into relevant topics, while Kafka consumers continuously read and loaded records into MongoDB collections. This

asynchronous and fault-tolerant architecture ensured scalability and resilience in data ingestion.

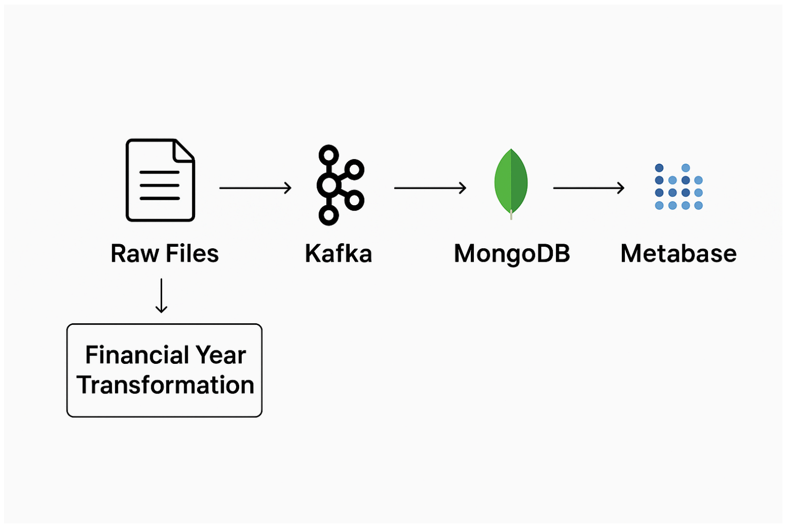
Phase 4: Evaluation and Integration

Once the MongoDB collections were populated, the schema and data structure were validated through Compass visual exploration and aggregate queries to confirm field consistency, data types, and referential alignment.

The final dataset was then integrated into Metabase, a modern BI platform chosen for its native MongoDB support, intuitive dashboard design, and ability to perform real-time analytical queries. Flattened view collections enabled faster aggregation, visual drill-downs, and dynamic filtering directly from Metabase dashboards.

Through this structured system design methodology, the pipeline successfully converted inconsistent and fragmented source data into a highly structured, query-efficient MongoDB repository. The integration of Compass-based aggregation testing and AI-guided cleaning logic ensured data accuracy, fiscal integrity, and optimal readiness for analytical visualization within Metabase.

Below is a visual model of the processes



3. Advantages and Benefits of NoSQL over Relational Systems

NoSQL databases have become a strong alternative to traditional relational database management systems (RDBMS). They are built for situations where data changes quickly or comes in large volumes, offering greater flexibility, scalability, and support for real-time processing. Unlike relational databases that depend on rigid table structures and predefined schemas, NoSQL systems use more adaptable, schema-less designs that can handle diverse and constantly evolving data efficiently (Khan *et al.*, 2023).

4.1 Flexibility for Hierarchical and Semi-Structured Data

Document-based NoSQL databases such as MongoDB can naturally represent hierarchical data by storing it in JSON-like documents. This removes the need for complicated joins or table relationships that are common in relational databases (Amazon Web Services, n.d.). As a result, complex data structures, like customer accounts, orders, and nested product details can be stored in a single record. This flexibility makes NoSQL systems well suited for modern analytical, IoT, and business intelligence environments, where data rarely fits into one fixed format (Khan *et al.*, 2023).

4.2 Scalability and Distributed Architecture

A major advantage of NoSQL is its ability to scale horizontally. Data can be spread across several servers using sharding and replication, improving both system performance and reliability (Anderson & Nicholson, 2022). Relational databases typically scale vertically by upgrading hardware, which quickly becomes expensive and limits capacity. With NoSQL, performance remains consistent as new servers are added, allowing the database to grow naturally with demand, a key reason it is used in large-scale web and enterprise systems (SHIFT ASIA, 2025).

4.3 Schema Flexibility and Rapid Development

Relational databases often slow development because they rely on strict schemas that must be altered manually when data requirements change. NoSQL databases remove that limitation by allowing new data fields or structures to be added without downtime (Atlan, 2024). This schema-less approach supports rapid iteration and agile development, where applications evolve over time. Teams can adjust their data models as features expand without redesigning the entire database (Anderson & Nicholson, 2022).

4.4 Real-Time Data Handling

NoSQL databases also perform better when handling continuous or high-speed data streams. Their distributed architecture supports quick data ingestion and retrieval without locking up the system. In performance testing, a NoSQL-based IoT platform processed incoming data over fifty times faster than a traditional SQL system (Hong *et al.*, 2023). This level of speed and scalability makes NoSQL ideal for real-time monitoring, analytics, and decision-making environments.

4.5 Conclusion

Overall, NoSQL databases provide a more flexible, scalable, and adaptive solution compared to traditional relational systems. They handle hierarchical and semi-structured data with ease, grow seamlessly across multiple servers, and process real-time information efficiently. For organisations that rely on large, complex, or constantly changing datasets, NoSQL delivers the performance and versatility needed in today's data-driven world.

4. NoSQL Solution Description

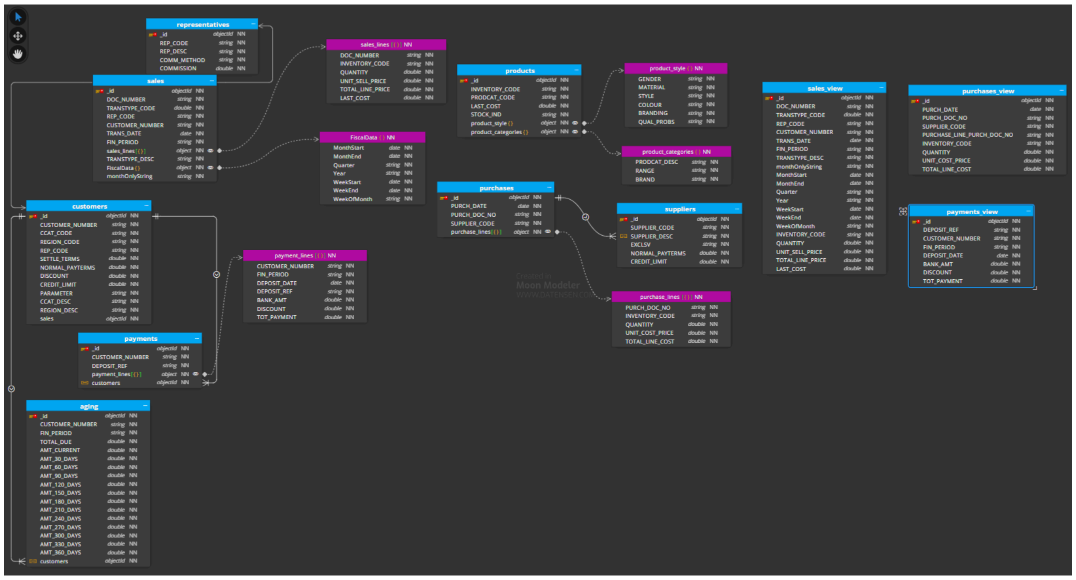
Our MongoDB implementation is designed around eight core collections, each representing a key business domain. To optimize query performance and minimize joins, all related customer-specific data is consolidated within a single Customers collection. Line-level details (e.g., sales lines, purchase lines, and payment lines) are embedded within their corresponding header documents to preserve context and improve read efficiency.

Additionally, we created three view collections: SALES_VIEW, PURCHASES_VIEW, and PAYMENT_VIEW

to support data flattening and simplify lookups for our business intelligence (BI) dashboards.

Each collection is governed by a strict JSON schema to ensure data consistency, enforce data types, and maintain referential integrity across the database.

Logical data model:



5. Implementation Steps

5.1 Overview

The implementation phase is about taking the NoSQL design and physical design and turning it into a working prototype for ClearVue Ltd's BI system. The goal is to automate the ETL pipeline from raw Excel datasets to interactive BI dashboards and have real-time updates through Apache Kafka and MongoDB Atlas.

The pipeline had five stages:

1. Cleaning Excel data
2. Python Script data Cleaning and JSON generation
3. Real-time data streaming via Apache Kafka
4. Data ingestion and aggregation in MongoDB Atlas
5. Visualization and reporting through Metabase dashboards

5.2 Implementation Steps

Cleaning Excel Data

The raw Excel spreadsheets provided by ClearVue Ltd had inconsistencies such as duplicate records, missing customer references and misaligned product categories. The following was needed fix this:

- Python (Pandas) scripts to remove duplicates and placeholder values (e.g. 599000, 999999).
- Handled null values or invalid fields,
- Data types standardized across all 19 files to ensure compatibility between related entities such as Customer ,Sales and Payments.

Python Data Processing

Multiple Python scripts automated the data transformation workflow:

- Data ingestion: Reading the cleaned Excel sheets using Pandas.
- Data transformation: Converting tabular data into hierarchical JSON documents matching the MongoDB NoSQL model.
- Financial alignment: Calculating custom financial-period fields (e.g. FIN_MONTH, FIN_YEAR) to match ClearVue's fiscal calendar, last Saturday and last Friday cycle.
- Output preparation: Exporting the final JSON structures for Kafka streaming.

Each dataset (Customers, Products, Sales, Payments, Suppliers, Purchases, Representatives, Aging) was made to match the same schema and fiscal alignment before being streamed.

Streaming Data Using Apache Kafka

Kafka was the middleware for real-time event propagation:

- Cleaned JSON documents were published to Kafka topics using Python producer scripts.
- Each topic represented a business domain (e.g. sales.realtime, payments.realtime, purchases.realtime).
- Consumer scripts subscribed to these topics to read messages and forward them to MongoDB Atlas.
- A transaction simulator was implemented to mimic live payment and sales updates, to test the pipeline could handle continuous input.MongoDB Atlas Storage and Processing

- Kafka's producer–consumer structure ensured durability, scalability, and real-time synchronization between operational data and BI layers.
- Docker was used to run a local instance of Apache Kafka

MongoDB Atlas Storage and Processing

MongoDB Atlas was the final storage layer:

- Each Kafka consumed dataset was stored in a corresponding MongoDB collection.
- MongoDB Compass was used to explore, flatten and validate nested JSON structures.
- Aggregation Pipelines were used to further nest and denormalise data and create period-to-date fields matching ClearVue's fiscal calendar.
- Indexed fields (Customer_ID, Product_ID, Supplier_ID) were added to speed up BI queries.

This produced a centralised, cloud hosted repository of clean and analytics ready data.

BI Dashboard Creation

The final analytical layer of the BI pipeline was built using Metabase instead of Power BI - the major departure from the original plan.

- First off data from MongoDB Atlas was connected to Metabase using the MongoDB connector
- Before visualisation some Python scripts were used to tidy and flatten nested documents
- The dashboards shows off KPIs such as total sales, product performance, and customer payment status
- To make it really usable, filters for different financial periods and date pickers were added to mirror ClearVue's bespoke fiscal structure.

This ended up with interactive, browser-based dashboards that gave management the ability to monitor things in real-time.

5.3 Tools and Technologies

Component	Tool/Technology	Purpose
Data Cleaning	Python (Pandas, NumPy)	Remove inconsistencies

		and duplicates
Data Transformation	Python (json, datetime modules)	Convert cleaned Excel to JSON
Real-Time Streaming	Apache Kafka	Stream data between producers and consumers
Data Storage	MongoDB Atlas	Store, index, and aggregate data
Data Exploration	MongoDB Compass	Flatten and validate JSON structures
Visualization	Metabase	Create dashboards and BI visualizations

5.4 Testing and Validation

Before deploying anything every single Python cleaning script was rigorously tested with sample Excel data to make sure all the data was being field-checked correctly.

Kafka producer and consumer scripts were given a good going over using simulated transaction streams to make sure they were doing their job properly.

The aggregated data from MongoDB was cross-checked with manually calculated totals from Excel to make sure everything was adding up correctly

Metabase visualisations were also double-checked to make sure they were showing the right fiscal months.

5.5 Outcome

The end result was that the BI pipeline achieved the following:

- It automated raw Excel data all the way through to Kafka, MongoDB and then Metabase.
- Clean, validated data got piped in structured to ClearVue's fiscal calendar
- Real-time event streaming was also verified using simulated transactions
- In the end we ended up with a scalable, cloud-based BI system that could handle all the analytical and reporting needs of ClearVue.

6. Critical Success Factors

Critical Success Factors for the ClearVue Project

1. **Data Accuracy and Integrity**
   Ensuring all extracted, transformed, and loaded data is consistent, validated, and free from duplication or loss during ETL processes.

2. **Efficient ETL Pipeline Design**
   The ETL process must be automated, optimized, and capable of handling data from multiple sources without performance bottlenecks.

3. **Scalable and Flexible Data Model**
   The MongoDB schema design (embedding vs referencing) must support scalability, easy updates, and efficient query performance for BI analysis.

4. **Effective Integration with BI Tools**
   Seamless connectivity and performance between MongoDB and Metabase are essential for real-time dashboarding and visualization.

5. **User-Centric Dashboard Design**
   Dashboards should be intuitive, visually clear, and aligned with key decision-making needs allowing stakeholders to easily interpret metrics.

6. **Performance Optimization**
   Queries and aggregations must run efficiently, avoiding excessive memory usage or long execution times that degrade user experience.

7. **Security and Access Control**
   Proper authentication, role-based permissions, and secure handling of sensitive business data are vital for system reliability and compliance.

8. **Stakeholder Engagement**
   Continuous feedback from users (e.g., managers or analysts) ensures the BI system meets actual business requirements and drives adoption.

9. **Comprehensive Documentation and Training**
   Clear documentation of ETL workflows, data models, and dashboard logic

supports maintainability and scalability of the system.

10. **Monitoring and Maintenance**
    Regular system audits, performance tracking, and error logging ensure long-term stability and reliability of the ClearVue solution.

11. **Accurate Alignment with Fiscal Calendar**
    The BI system must correctly align all reporting and aggregations with ClearVue Ltd.'s unique financial year structure (months defined from the last Saturday to the last Friday). This temporal precision is essential for management trust and decision-making.

12. **High-Performance Querying and Aggregation**
    The BI system should efficiently handle large and complex aggregations, leveraging NoSQL advantages (e.g., document embedding or referencing) to minimize query latency and avoid memory constraints.

7. Reflection

7.1 Trade-Offs and Design Decisions

Embedding vs. Referencing

ClearVue Ltd. adopted a hybrid MongoDB schema to navigate the trade-offs inherent in database design, prioritizing a balance of performance and data maintenance.

The design featured embedding for closely related transactional information: Sales, Payment, and Purchase Lines were placed inside their header documents. This choice directly improved query efficiency and aggregation performance in Metabase by eliminating the need for multiple data lookups. To maximize analytical speed, flattened structures were also created from these embedded collections, allowing BI tools to quickly compute key performance indicators like gross profit.

For shared reference data, such as customer and product details, the system used denormalization, placing them in standalone collections. This strategy reduced the number of joins needed across transactional documents, speeding up queries and streamlining the process of updating central reference data. Finally, supplier and aging collections were kept as is, as their data was independent of the main transactional flow.

This intentional mix of embedding, denormalization, and referencing results in a BI system that is highly effective for analysis yet structurally sound and easy to manage.

NoSQL vs. Relational Systems

Traditional relational systems like MS Access or SQL Server offer ACID consistency, strong referential integrity, and structured schemas. However, they proved too rigid for ClearVue's hierarchical, semi-structured datasets and custom fiscal calendar.

By contrast, MongoDB's schema flexibility allowed the team to store nested documents (such as line_items and age_analysis) without altering table definitions, enabling faster adaptation to new business requirements, such as supplier analytics.

Aspect	**Relational (SQL)**	**NoSQL (MongoDB)**	**Design Decision**
Schema	Fixed, table-based	Flexible, document-based	Allowed dynamic addition of fields without redesign
Data Relationships	JOINs between tables	Embedded or referenced documents	Used hybrid model for speed + integrity

Scalability	Vertical (hardware upgrades)	Horizontal (sharding, clustering)	Future-ready scalability
Consistency	Strong (ACID)	Eventual (BASE)	Acceptable for analytics workloads
Querying	Structured SQL	JSON-based queries	Simplified for Metabase integration

The trade-off was reduced native join support and transaction control, requiring more thoughtful data modelling and validation.
Still, the document-oriented approach best supported ClearVue's real time BI and supplier analytics roadmap.

Tool Choice

Metabase was selected for its user-friendly interface, easy MongoDB integration, simple queries, and native JSON compatibility.
Although it lacked direct real-time streaming integration with Kafka during prototype testing (manual refreshes were used instead), Metabase's dynamic filtering and visual clarity made it ideal for ClearVue's financial-period reporting.

Alternative tools such as Tableau or Python Plotly Dash were considered, but Metabase offered the best balance between ease of use, professional output, and scalability for enterprise adoption.

Streaming Pipeline

Integrating Apache Kafka with MongoDB Change Streams enabled real-time ingestion of payment transactions and updates. This significantly improved system responsiveness but required careful validation to avoid inconsistent records.

While Kafka increased timeliness and automation potential, it also introduced configuration complexity and monitoring challenges.
Future versions should include:

- Dead-letter queues for failed messages
- Automated error handling
- Stream monitoring dashboards for operational visibility

7.2 Future Enhancements

To strengthen and extend the ClearVue BI solution, the following future enhancements are proposed:

1. **Automated Metabase Service Integration**
   Deploy dashboards to the Metabase Service using scheduled refreshes or DirectQuery connectors to MongoDB Atlas for true live reporting.
2. **Enhanced Kafka Streaming Features**
   Add real-time anomaly detection and fraud alerting on payment data using Python ML libraries (e.g., Scikit-Learn or TensorFlow).
3. **Automated Data-Quality Monitoring**
   Implement triggers or scripts that flag missing or invalid data in MongoDB, sending alerts via Power Automate or Slack.
4. **Supplier and Stock API Integration**
   Connect with supplier APIs to automatically synchronize product availability, pricing trends, and delivery metrics, extending the BI dashboard into a real operational tool.

5. **Role-Based Access Control (RBAC)**
   Apply fine-grained access permissions to restrict sensitive data, ensuring finance, sales, and executive users view only relevant KPIs.
6. **Mobile Dashboard Deployment**
   Design lightweight, mobile-responsive dashboards for management teams to monitor sales and payments on the go.
7. **Cloud Scalability**
   Migrate the full system to MongoDB Atlas and Kafka Cloud Clusters with automated backups and replication for improved reliability.
8. **AI-Assisted Insights**
   Integrate Microsoft Fabric Copilot or OpenAI-powered analytics to generate auto-summaries (e.g., *“Sales increased 15% in Q3 for the Western Region”*).

7.3 Scalability Considerations

Scalability was a core design principle from the beginning. As ClearVue’s data volume and user demand grow, the following strategies ensure sustainable performance:

- **Document Size Management:** Keep within MongoDB’s 16 MB document limit; archive older data quarterly.
- **Index Optimization:** Apply compound indexes on high-frequency fields (e.g., Customer_ID, FIN_PERIOD, REGION).
- **Pre-Aggregated Collections:** Reduce Metabase load times by storing common aggregations.
- **Data Partitioning:** Separate Sales and Payments collections by financial year for faster queries.
- **Horizontal Scaling (Sharding):** Distribute data across clusters when transactions exceed single-node capacity.
- **Caching and ETL Pipelines:** Introduce Redis caching and optimized ETL jobs for repeated BI queries.

These design principles ensure that ClearVue's BI system remains enterprise-ready, high-performing, and adaptable to future expansion.

7.4 Outcome and Conclusion

The prototype successfully demonstrated that a document-oriented NoSQL architecture, integrated with Python ETL, Kafka streaming, and Metabase , can deliver near-real-time analytics without requiring a full-scale data warehouse.

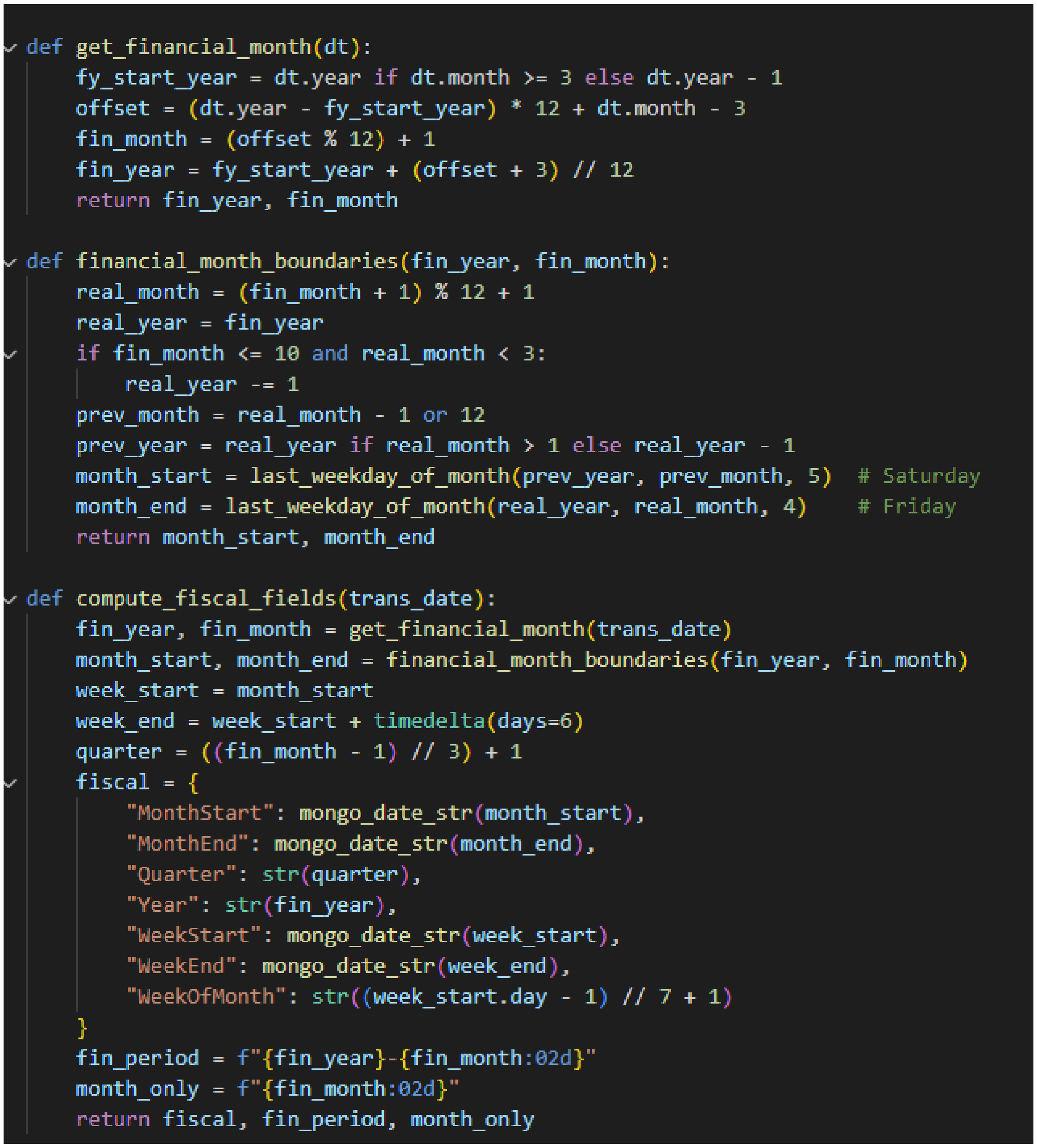
Key outcomes:

- Produced dynamic dashboards aligned with ClearVue's unique fiscal calendar.
- Integrated cleaned, validated, and streamed data into unified visual reporting.
- Delivered a scalable, modular proof of concept ready for supplier analytics expansion.
- In conclusion, the project achieved both technical and business objectives, proving that a NoSQL based BI ecosystem can support ClearVue Ltd.'s evolving analytics strategy, offering flexibility, scalability, and actionable insight for data-driven decision-making.

---

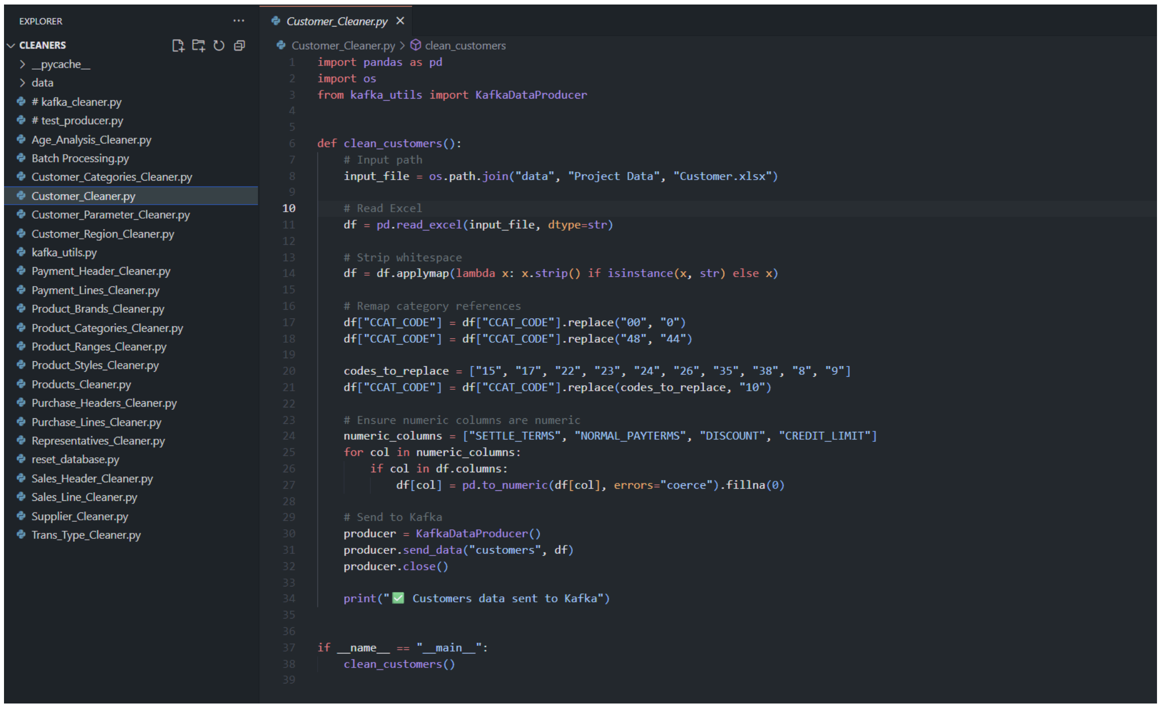
8. System Documentation

Sales data kafka producer(Financial Year snippet)



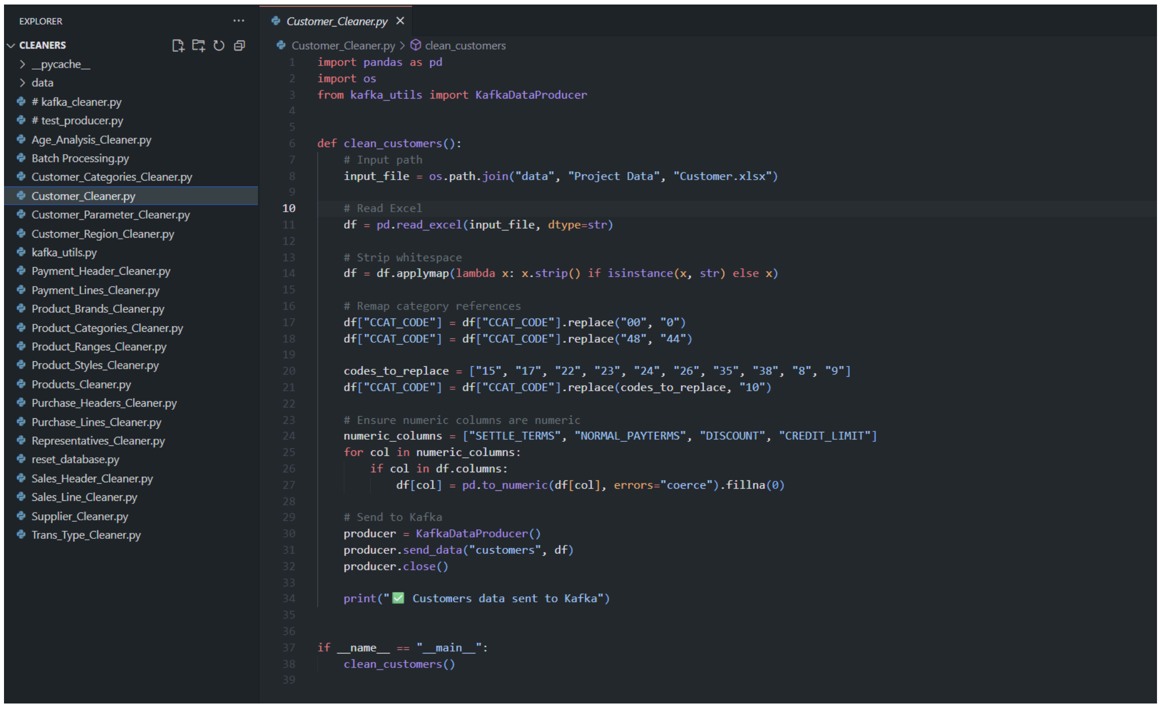
```python
def get_financial_month(dt):
    fy_start_year = dt.year if dt.month >= 3 else dt.year - 1
    offset = (dt.year - fy_start_year) * 12 + dt.month - 3
    fin_month = (offset % 12) + 1
    fin_year = fy_start_year + (offset + 3) // 12
    return fin_year, fin_month

def financial_month_boundaries(fin_year, fin_month):
    real_month = (fin_month + 1) % 12 + 1
    real_year = fin_year
    if fin_month <= 10 and real_month < 3:
        real_year -= 1
    prev_month = real_month - 1 or 12
    prev_year = real_year if real_month > 1 else real_year - 1
    month_start = last_weekday_of_month(prev_year, prev_month, 5)  # Saturday
    month_end = last_weekday_of_month(real_year, real_month, 4)    # Friday
    return month_start, month_end

def compute_fiscal_fields(trans_date):
    fin_year, fin_month = get_financial_month(trans_date)
    month_start, month_end = financial_month_boundaries(fin_year, fin_month)
    week_start = month_start
    week_end = week_start + timedelta(days=6)
    quarter = ((fin_month - 1) // 3) + 1
    fiscal = {
        "MonthStart": mongo_date_str(month_start),
        "MonthEnd": mongo_date_str(month_end),
        "Quarter": str(quarter),
        "Year": str(fin_year),
        "WeekStart": mongo_date_str(week_start),
        "WeekEnd": mongo_date_str(week_end),
        "WeekOfMonth": str((week_start.day - 1) // 7 + 1)
    }
    fin_period = f"{fin_year}-{fin_month:02d}"
    month_only = f"{fin_month:02d}"
    return fiscal, fin_period, month_only
```

Cleaning script example ()


EXPLORER
CLEANERS
__pycache__
data
# kafka_cleaner.py
# test_producer.py
Age_Analysis_Cleaner.py
Batch Processing.py
Customer_Categories_Cleaner.py
Customer_Cleaner.py
Customer_Parameter_Cleaner.py
Customer_Region_Cleaner.py
kafka_utils.py
Payment_Header_Cleaner.py
Payment_Lines_Cleaner.py
Product_Brands_Cleaner.py
Product_Categories_Cleaner.py
Product_Ranges_Cleaner.py
Product_Styles_Cleaner.py
Products_Cleaner.py
Purchase_Headers_Cleaner.py
Purchase_Lines_Cleaner.py
Representatives_Cleaner.py
reset_database.py
Sales_Header_Cleaner.py
Sales_Line_Cleaner.py
Supplier_Cleaner.py
Trans_Type_Cleaner.py
Customer_Cleaner.py
Customer_Cleaner.py > clean_customers

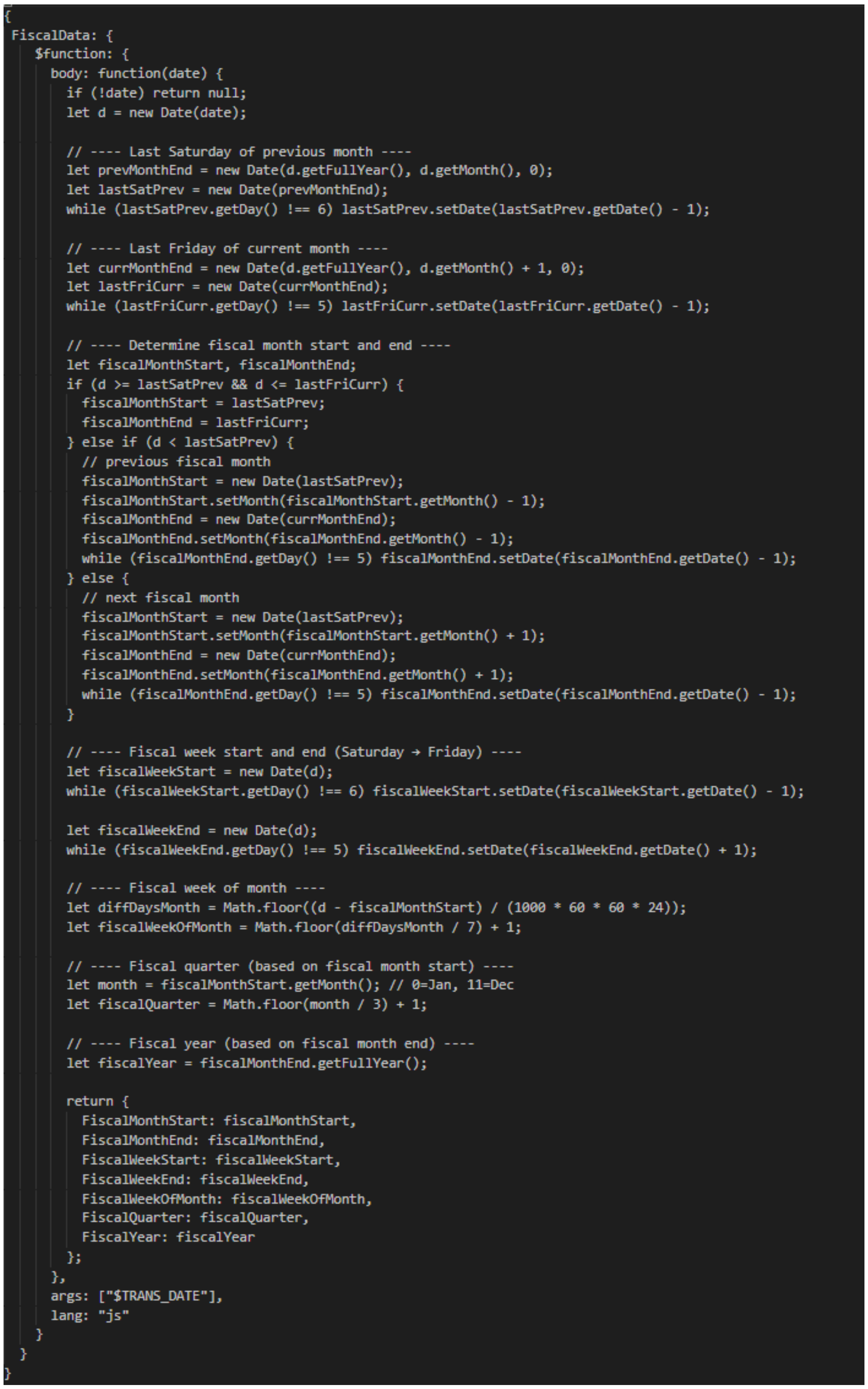
```python
import pandas as pd
import os
from kafka_utils import KafkaDataProducer


def clean_customers():
    # Input path
    input_file = os.path.join("data", "Project Data", "Customer.xlsx")

    # Read Excel
    df = pd.read_excel(input_file, dtype=str)

    # Strip whitespace
    df = df.applymap(lambda x: x.strip() if isinstance(x, str) else x)

    # Remap category references
    df["CCAT_CODE"] = df["CCAT_CODE"].replace("00", "0")
    df["CCAT_CODE"] = df["CCAT_CODE"].replace("48", "44")

    codes_to_replace = ["15", "17", "22", "23", "24", "26", "35", "38", "8", "9"]
    df["CCAT_CODE"] = df["CCAT_CODE"].replace(codes_to_replace, "10")

    # Ensure numeric columns are numeric
    numeric_columns = ["SETTLE_TERMS", "NORMAL_PAYTERMS", "DISCOUNT", "CREDIT_LIMIT"]
    for col in numeric_columns:
        if col in df.columns:
            df[col] = pd.to_numeric(df[col], errors="coerce").fillna(0)

    # Send to Kafka
    producer = KafkaDataProducer()
    producer.send_data("customers", df)
    producer.close()

    print("✅ Customers data sent to Kafka")


if __name__ == "__main__":
    clean_customers()
```

Example of a view using mongoDB compass aggregation pipeline(Sales):



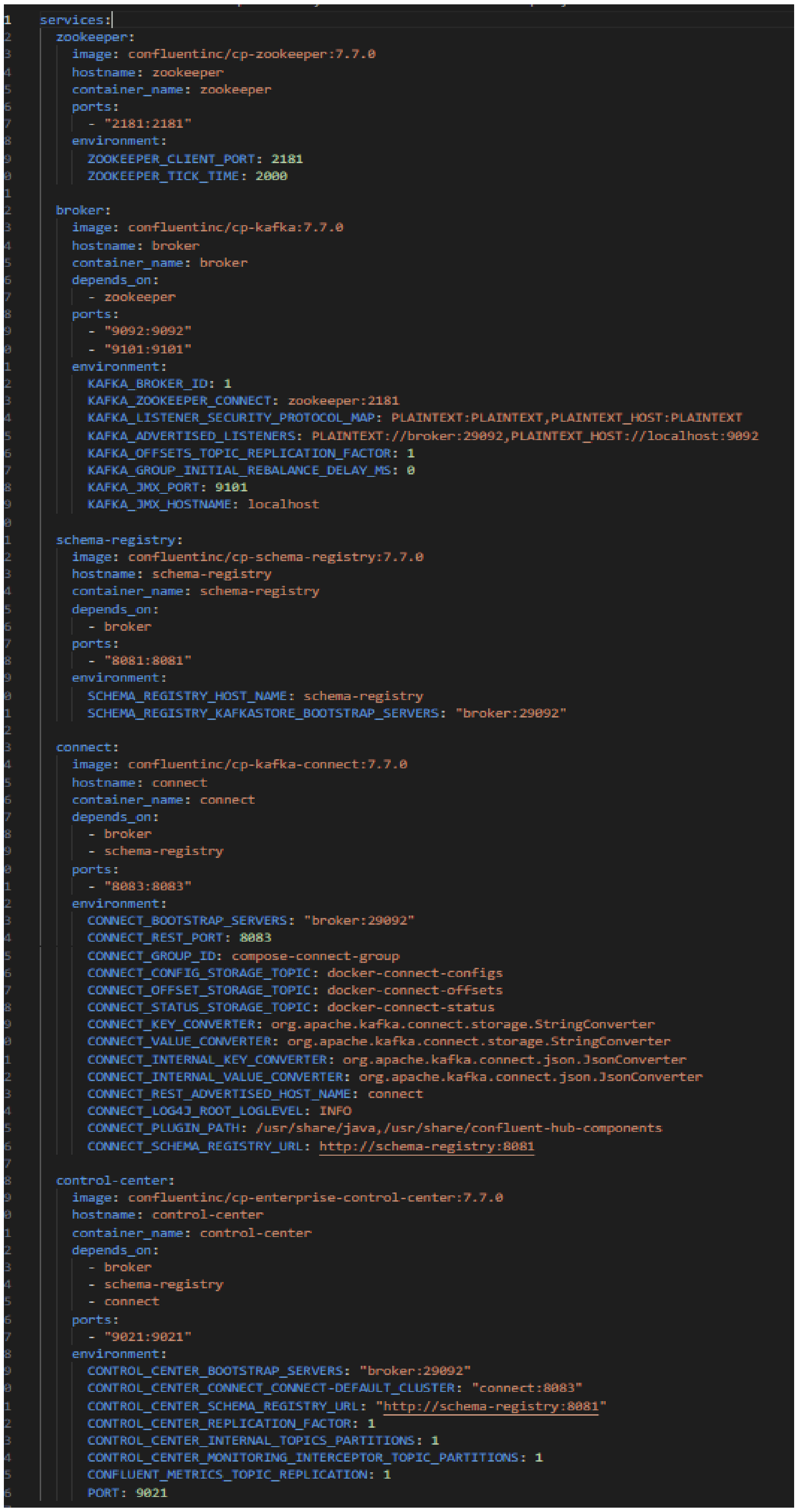
```python
import pandas as pd
import os
from kafka_utils import KafkaDataProducer


def clean_customers():
    # Input path
    input_file = os.path.join("data", "Project Data", "Customer.xlsx")

    # Read Excel
    df = pd.read_excel(input_file, dtype=str)

    # Strip whitespace
    df = df.applymap(lambda x: x.strip() if isinstance(x, str) else x)

    # Remap category references
    df["CCAT_CODE"] = df["CCAT_CODE"].replace("00", "0")
    df["CCAT_CODE"] = df["CCAT_CODE"].replace("48", "44")

    codes_to_replace = ["15", "17", "22", "23", "24", "26", "35", "38", "8", "9"]
    df["CCAT_CODE"] = df["CCAT_CODE"].replace(codes_to_replace, "10")

    # Ensure numeric columns are numeric
    numeric_columns = ["SETTLE_TERMS", "NORMAL_PAYTERMS", "DISCOUNT", "CREDIT_LIMIT"]
    for col in numeric_columns:
        if col in df.columns:
            df[col] = pd.to_numeric(df[col], errors="coerce").fillna(0)

    # Send to Kafka
    producer = KafkaDataProducer()
    producer.send_data("customers", df)
    producer.close()

    print("✅ Customers data sent to Kafka")


if __name__ == "__main__":
    clean_customers()
```

Example of aggregate cleaning and nesting(Financial year data):



```
{
 FiscalData: {
    $function: {
      body: function(date) {
        if (!date) return null;
        let d = new Date(date);

        // ---- Last Saturday of previous month ----
        let prevMonthEnd = new Date(d.getFullYear(), d.getMonth(), 0);
        let lastSatPrev = new Date(prevMonthEnd);
        while (lastSatPrev.getDay() !== 6) lastSatPrev.setDate(lastSatPrev.getDate() - 1);

        // ---- Last Friday of current month ----
        let currMonthEnd = new Date(d.getFullYear(), d.getMonth() + 1, 0);
        let lastFriCurr = new Date(currMonthEnd);
        while (lastFriCurr.getDay() !== 5) lastFriCurr.setDate(lastFriCurr.getDate() - 1);

        // ---- Determine fiscal month start and end ----
        let fiscalMonthStart, fiscalMonthEnd;
        if (d >= lastSatPrev && d <= lastFriCurr) {
          fiscalMonthStart = lastSatPrev;
          fiscalMonthEnd = lastFriCurr;
        } else if (d < lastSatPrev) {
          // previous fiscal month
          fiscalMonthStart = new Date(lastSatPrev);
          fiscalMonthStart.setMonth(fiscalMonthStart.getMonth() - 1);
          fiscalMonthEnd = new Date(currMonthEnd);
          fiscalMonthEnd.setMonth(fiscalMonthEnd.getMonth() - 1);
          while (fiscalMonthEnd.getDay() !== 5) fiscalMonthEnd.setDate(fiscalMonthEnd.getDate() - 1);
        } else {
          // next fiscal month
          fiscalMonthStart = new Date(lastSatPrev);
          fiscalMonthStart.setMonth(fiscalMonthStart.getMonth() + 1);
          fiscalMonthEnd = new Date(currMonthEnd);
          fiscalMonthEnd.setMonth(fiscalMonthEnd.getMonth() + 1);
          while (fiscalMonthEnd.getDay() !== 5) fiscalMonthEnd.setDate(fiscalMonthEnd.getDate() - 1);
        }

        // ---- Fiscal week start and end (Saturday → Friday) ----
        let fiscalWeekStart = new Date(d);
        while (fiscalWeekStart.getDay() !== 6) fiscalWeekStart.setDate(fiscalWeekStart.getDate() - 1);

        let fiscalWeekEnd = new Date(d);
        while (fiscalWeekEnd.getDay() !== 5) fiscalWeekEnd.setDate(fiscalWeekEnd.getDate() + 1);

        // ---- Fiscal week of month ----
        let diffDaysMonth = Math.floor((d - fiscalMonthStart) / (1000 * 60 * 60 * 24));
        let fiscalWeekOfMonth = Math.floor(diffDaysMonth / 7) + 1;

        // ---- Fiscal quarter (based on fiscal month start) ----
        let month = fiscalMonthStart.getMonth(); // 0=Jan, 11=Dec
        let fiscalQuarter = Math.floor(month / 3) + 1;

        // ---- Fiscal year (based on fiscal month end) ----
        let fiscalYear = fiscalMonthEnd.getFullYear();

        return {
          FiscalMonthStart: fiscalMonthStart,
          FiscalMonthEnd: fiscalMonthEnd,
          FiscalWeekStart: fiscalWeekStart,
          FiscalWeekEnd: fiscalWeekEnd,
          FiscalWeekOfMonth: fiscalWeekOfMonth,
          FiscalQuarter: fiscalQuarter,
          FiscalYear: fiscalYear
        };
      },
      args: ["$TRANS_DATE"],
      lang: "js"
    }
  }
}
```

Docker Compose setup file



```yaml
services:
  zookeeper:
    image: confluentinc/cp-zookeeper:7.7.0
    hostname: zookeeper
    container_name: zookeeper
    ports:
      - "2181:2181"
    environment:
      ZOOKEEPER_CLIENT_PORT: 2181
      ZOOKEEPER_TICK_TIME: 2000

  broker:
    image: confluentinc/cp-kafka:7.7.0
    hostname: broker
    container_name: broker
    depends_on:
      - zookeeper
    ports:
      - "9092:9092"
      - "9101:9101"
    environment:
      KAFKA_BROKER_ID: 1
      KAFKA_ZOOKEEPER_CONNECT: zookeeper:2181
      KAFKA_LISTENER_SECURITY_PROTOCOL_MAP: PLAINTEXT:PLAINTEXT,PLAINTEXT_HOST:PLAINTEXT
      KAFKA_ADVERTISED_LISTENERS: PLAINTEXT://broker:29092,PLAINTEXT_HOST://localhost:9092
      KAFKA_OFFSETS_TOPIC_REPLICATION_FACTOR: 1
      KAFKA_GROUP_INITIAL_REBALANCE_DELAY_MS: 0
      KAFKA_JMX_PORT: 9101
      KAFKA_JMX_HOSTNAME: localhost

  schema-registry:
    image: confluentinc/cp-schema-registry:7.7.0
    hostname: schema-registry
    container_name: schema-registry
    depends_on:
      - broker
    ports:
      - "8081:8081"
    environment:
      SCHEMA_REGISTRY_HOST_NAME: schema-registry
      SCHEMA_REGISTRY_KAFKASTORE_BOOTSTRAP_SERVERS: "broker:29092"

  connect:
    image: confluentinc/cp-kafka-connect:7.7.0
    hostname: connect
    container_name: connect
    depends_on:
      - broker
      - schema-registry
    ports:
      - "8083:8083"
    environment:
      CONNECT_BOOTSTRAP_SERVERS: "broker:29092"
      CONNECT_REST_PORT: 8083
      CONNECT_GROUP_ID: compose-connect-group
      CONNECT_CONFIG_STORAGE_TOPIC: docker-connect-configs
      CONNECT_OFFSET_STORAGE_TOPIC: docker-connect-offsets
      CONNECT_STATUS_STORAGE_TOPIC: docker-connect-status
      CONNECT_KEY_CONVERTER: org.apache.kafka.connect.storage.StringConverter
      CONNECT_VALUE_CONVERTER: org.apache.kafka.connect.storage.StringConverter
      CONNECT_INTERNAL_KEY_CONVERTER: org.apache.kafka.connect.json.JsonConverter
      CONNECT_INTERNAL_VALUE_CONVERTER: org.apache.kafka.connect.json.JsonConverter
      CONNECT_REST_ADVERTISED_HOST_NAME: connect
      CONNECT_LOG4J_ROOT_LOGLEVEL: INFO
      CONNECT_PLUGIN_PATH: /usr/share/java,/usr/share/confluent-hub-components
      CONNECT_SCHEMA_REGISTRY_URL: http://schema-registry:8081

  control-center:
    image: confluentinc/cp-enterprise-control-center:7.7.0
    hostname: control-center
    container_name: control-center
    depends_on:
      - broker
      - schema-registry
      - connect
    ports:
      - "9021:9021"
    environment:
      CONTROL_CENTER_BOOTSTRAP_SERVERS: "broker:29092"
      CONTROL_CENTER_CONNECT_CONNECT-DEFAULT_CLUSTER: "connect:8083"
      CONTROL_CENTER_SCHEMA_REGISTRY_URL: "http://schema-registry:8081"
      CONTROL_CENTER_REPLICATION_FACTOR: 1
      CONTROL_CENTER_INTERNAL_TOPICS_PARTITIONS: 1
      CONTROL_CENTER_MONITORING_INTERCEPTOR_TOPIC_PARTITIONS: 1
      CONFLUENT_METRICS_TOPIC_REPLICATION: 1
      PORT: 9021
```

MongoDB JSON Schemas for all collections

Schema’s
AGING:
{

```
"$jsonSchema": {
    "bsonType": "object",
    "required": [
      "_id",
      "AMT_120_DAYS",
      "AMT_150_DAYS",
      "AMT_180_DAYS",
      "AMT_210_DAYS",
      "AMT_240_DAYS",
      "AMT_270_DAYS",
      "AMT_300_DAYS",
      "AMT_30_DAYS",
      "AMT_330_DAYS",
      "AMT_360_DAYS",
      "AMT_60_DAYS",
      "AMT_90_DAYS",
      "AMT_CURRENT",
      "CUSTOMER_NUMBER",
      "FIN_PERIOD",
      "TOTAL_DUE"
    ],
    "properties": {
      "_id": {
        "bsonType": "objectId"
      },
      "AMT_120_DAYS": {
        "bsonType": "double"
      },
      "AMT_150_DAYS": {
        "bsonType": "double"
      },
      "AMT_180_DAYS": {
        "bsonType": "int"
      },
      "AMT_210_DAYS": {
        "bsonType": "int"
      },
      "AMT_240_DAYS": {
        "bsonType": "int"
      },
      "AMT_270_DAYS": {
        "bsonType": "int"
      },
      "AMT_300_DAYS": {
        "bsonType": "int"
      },
      "AMT_30_DAYS": {
        "bsonType": "double"
      },
      "AMT_330_DAYS": {
        "bsonType": "int"
      },
      "AMT_360_DAYS": {
        "bsonType": "int"
      },
      "AMT_60_DAYS": {
        "bsonType": "double"
      },
      "AMT_90_DAYS": {
        "bsonType": "double"
      },
      "AMT_CURRENT": {
        "bsonType": "double"
      },
      "CUSTOMER_NUMBER": {
        "bsonType": "string"
      },
      "FIN_PERIOD": {
        "bsonType": "string"
      },
      "TOTAL_DUE": {
        "bsonType": "double"
      }
    }
  }
}
```

CUSTOMERS:
{

```
"$jsonSchema": {
    "bsonType": "object",
    "required": [
      "_id",
      "CREDIT_LIMIT",
      "CUSTOMER_NUMBER",
      "DISCOUNT",
      "NORMAL_PAYTERMS",
      "PARAMETER",
      "REP_CODE",
      "SETTLE_TERMS"
    ],
    "properties": {
      "_id": {
        "bsonType": "objectId"
      },
"CCAT_CODE": {
        "bsonType": "string"
      },
      "CCAT_DESC": {
        "bsonType": "string"
      },
      "CREDIT_LIMIT": {
        "bsonType": "int"
      },
      "CUSTOMER_NUMBER": {
        "bsonType": "string"
      },
      "DISCOUNT": {
        "bsonType": "int"
      },
      "NORMAL_PAYTERMS": {
        "bsonType": "int"
      },
      "PARAMETER": {
        "bsonType": "string"
      },
      "REGION_CODE": {
        "bsonType": "string"
      },
      "REGION_DESC": {
        "bsonType": "string"
      },
      "REP_CODE": {
        "bsonType": "string"
      },
      "SETTLE_TERMS": {
        "bsonType": "double"
      }
    }
  }
}
```

PAYMENTS:

{

```
"$jsonSchema": {
    "bsonType": "object",
    "required": [
      "_id",
      "CUSTOMER_NUMBER",
      "DEPOSIT_REF",
      "payment_lines"
    ],
    "properties": {
      "_id": {
        "bsonType": "objectId"
      },
      "CUSTOMER_NUMBER": {
        "bsonType": "string"
      },
      "DEPOSIT_REF": {
        "bsonType": "string"
      },
      "payment_lines": {
        "bsonType": "array",
        "items": {
          "bsonType": "object",
          "properties": {
            "BANK_AMT": {
              "bsonType": [
                "double",
                "int"
              ]
            },
            "CUSTOMER_NUMBER": {
              "bsonType": "string"
            },
            "DEPOSIT_DATE": {
              "bsonType": "date"
            },
            "DEPOSIT_REF": {
              "bsonType": "string"
            },
            "DISCOUNT": {
              "bsonType": "int"
            },
            "FIN_PERIOD": {
              "bsonType": "string"
            },
            "TOT_PAYMENT": {
              "bsonType": "int"
            }
          },
          "required": [
            "BANK_AMT",
            "CUSTOMER_NUMBER",
            "DEPOSIT_DATE",
            "DEPOSIT_REF",
            "DISCOUNT",
            "FIN_PERIOD",
            "TOT_PAYMENT"
          ]
        }
      }
    }
  }
}
```

PAYMENT_VIEW:
{

```
"$jsonSchema": {
    "bsonType": "object",
    "required": [
      "_id",
      "BANK_AMT",
      "CUSTOMER_NUMBER",
      "DEPOSIT_DATE",
      "DEPOSIT_REF",
      "DISCOUNT",
      "FIN_PERIOD",
      "TOT_PAYMENT"
    ],
    "properties": {
      "_id": {
        "bsonType": "objectId"
      },
      "BANK_AMT": {
        "bsonType": [
          "double",
          "int"
        ]
      },
      "CUSTOMER_NUMBER": {
        "bsonType": "string"
      },
      "DEPOSIT_DATE": {
        "bsonType": "date"
      },
      "DEPOSIT_REF": {
        "bsonType": "string"
      },
      "DISCOUNT": {
        "bsonType": "int"
      },
      "FIN_PERIOD": {
        "bsonType": "string"
      },
      "TOT_PAYMENT": {
        "bsonType": "int"
      }
    }
  }
}
```

PRODUCTS:

{

```
"$jsonSchema": {
    "bsonType": "object",
    "required": [
      "_id",
      "INVENTORY_CODE",
      "LAST_COST",
      "PRODCAT_CODE",
      "product_categories",
      "product_style",
      "STOCK_IND"
    ],
    "properties": {
      "_id": {
        "bsonType": "objectId"
      },
      "INVENTORY_CODE": {
        "bsonType": "string"
      },
      "LAST_COST": {
        "bsonType": "int"
      },
      "PRODCAT_CODE": {
        "bsonType": "string"
      },
      "product_categories": {
        "bsonType": "object",
        "properties": {
          "BRAND": {
            "bsonType": "string"
          },
          "PRODCAT_DESC": {
```

```
          "bsonType": "string"
        },
        "RANGE": {
          "bsonType": "string"
        }
      },
      "required": [
        "BRAND",
        "PRODCAT_DESC",
        "RANGE"
      ]
    },
    "product_style": {
      "bsonType": "object",
      "properties": {
        "BRANDING": {
          "bsonType": "string"
        },
        "COLOUR": {
          "bsonType": "string"
        },
        "GENDER": {
          "bsonType": "string"
        },
        "MATERIAL": {
          "bsonType": "string"
        },
        "QUAL_PROBS": {
          "bsonType": "string"
        },
        "STYLE": {
          "bsonType": "string"
        }
      },
      "required": [
        "BRANDING",
        "COLOUR",
        "GENDER",
        "MATERIAL",
        "QUAL_PROBS",
        "STYLE"
      ]
      },
      "STOCK_IND": {
        "bsonType": "string"
      }
    }
  }
}
```

PURCHASES:

```
{
  "$jsonSchema": {
    "bsonType": "object",
    "required": [
      "_id",
      "PURCH_DATE",
      "PURCH_DOC_NO",
      "purchase_lines",
      "SUPPLIER_CODE"
    ],
    "properties": {
      "_id": {
        "bsonType": "objectId"
      },
      "PURCH_DATE": {
        "bsonType": "date"
      },
      "PURCH_DOC_NO": {
        "bsonType": "string"
      },
      "purchase_lines": {
        "bsonType": "array",
        "items": {
          "bsonType": "object",
          "properties": {
            "INVENTORY_CODE": {
              "bsonType": "string"
            },
            "PURCH_DOC_NO": {
              "bsonType": "string"
            },
            "QUANTITY": {
              "bsonType": "int"
            },
            "TOTAL_LINE_COST": {
              "bsonType": [
                "double",
                "int"
              ]
            },
            "UNIT_COST_PRICE": {
              "bsonType": [
                "double",
                "int"
              ]
            }
          },
          "required": [
            "INVENTORY_CODE",
            "PURCH_DOC_NO",
            "QUANTITY",
            "TOTAL_LINE_COST",
            "UNIT_COST_PRICE"
          ]
        }
      },
      "SUPPLIER_CODE": {
        "bsonType": "string"
      }
    }
  }
}
```

PURCHASES_VIEW:

{

```
"$jsonSchema": {
    "bsonType": "object",
    "required": [
      "_id",
      "INVENTORY_CODE",
      "PURCH_DATE",
      "PURCH_DOC_NO",
      "PURCHASE_LINE_PURCH_DOC_NO",
      "QUANTITY",
      "SUPPLIER_CODE",
      "TOTAL_LINE_COST",
      "UNIT_COST_PRICE"
    ],
    "properties": {
      "_id": {
        "bsonType": "objectId"
      },
      "INVENTORY_CODE": {
        "bsonType": "string"
      },
      "PURCH_DATE": {
        "bsonType": "date"
      },
      "PURCH_DOC_NO": {
        "bsonType": "string"
      },
      "PURCHASE_LINE_PURCH_DOC_NO": {
        "bsonType": "string"
      },
      "QUANTITY": {
        "bsonType": "int"
      },
      "SUPPLIER_CODE": {
        "bsonType": "string"
      },
      "TOTAL_LINE_COST": {
        "bsonType": [
          "double",
          "int"
        ]
      },
      "UNIT_COST_PRICE": {
        "bsonType": [
          "double",
          "int"
        ]
      }
    }
    }
}
```

REPRESENTATIVES:

```
{
  "$jsonSchema": {
    "bsonType": "object",
    "required": [
      "_id",
      "COMM_METHOD",
      "COMMISSION",
      "REP_CODE",
      "REP_DESC"
    ],
    "properties": {
      "_id": {
        "bsonType": "objectId"
      },
      "COMM_METHOD": {
        "bsonType": "string"
      },
      "COMMISSION": {
        "bsonType": "double"
      },
      "REP_CODE": {
        "bsonType": "string"
      },
      "REP_DESC": {
        "bsonType": "string"
      }
    }
  }
}
```

SALES:

```
{
  "$jsonSchema": {
    "bsonType": "object",
    "required": [
      "_id",
      "CUSTOMER_NUMBER",
      "DOC_NUMBER",
      "FIN_PERIOD",
      "FiscalData",
      "monthOnlyString",
      "REP_CODE",
      "sales_lines",
      "TRANS_DATE",
      "TRANSTYPE_CODE",
      "TRANSTYPE_DESC"
    ],
    "properties": {
      "_id": {
        "bsonType": "objectId"
      },
      "CUSTOMER_NUMBER": {
        "bsonType": "string"
      },
      "DOC_NUMBER": {
        "bsonType": "string"
      },
      "FIN_PERIOD": {
        "bsonType": "string"
      },
      "FiscalData": {
        "bsonType": "object",
        "properties": {
          "MonthEnd": {
            "bsonType": "date"
          },
          "MonthStart": {
            "bsonType": "date"
          },
          "Quarter": {
            "bsonType": "string"
          },
          "WeekEnd": {
            "bsonType": "date"
          },
          "WeekOfMonth": {
```

```
          "bsonType": "string"
        },
        "WeekStart": {
          "bsonType": "date"
        },
        "Year": {
          "bsonType": "string"
        }
      },
      "required": [
        "MonthEnd",
        "MonthStart",
        "Quarter",
        "WeekEnd",
        "WeekOfMonth",
        "WeekStart",
        "Year"
      ]
    },
    "monthOnlyString": {
      "bsonType": "string"
    },
    "REP_CODE": {
      "bsonType": "string"
    },
    "sales_lines": {
      "bsonType": "array",
      "items": {
        "bsonType": "object",
        "properties": {
          "DOC_NUMBER": {
            "bsonType": "string"
          },
          "INVENTORY_CODE": {
            "bsonType": "string"
          },
          "LAST_COST": {
            "bsonType": "int"
          },
          "QUANTITY": {
            "bsonType": "int"
          },
             "TOTAL_LINE_PRICE": {
               "bsonType": "int"
             },
             "UNIT_SELL_PRICE": {
               "bsonType": "int"
             }
           },
           "required": [
             "DOC_NUMBER",
             "INVENTORY_CODE",
             "LAST_COST",
             "QUANTITY",
             "TOTAL_LINE_PRICE",
             "UNIT_SELL_PRICE"
           ]
         }
       },
       "TRANS_DATE": {
         "bsonType": "date"
       },
       "TRANSTYPE_CODE": {
         "bsonType": "int"
       },
       "TRANSTYPE_DESC": {
         "bsonType": "string"
       }
     }
   }
}
```

SALES_VIEW:

{

```
"$jsonSchema": {
    "bsonType": "object",
    "required": [
      "_id",
      "CUSTOMER_NUMBER",
      "DOC_NUMBER",
      "FIN_PERIOD",
      "INVENTORY_CODE",
      "LAST_COST",
      "MonthEnd",
      "monthOnlyString",
      "MonthStart",
      "QUANTITY",
      "Quarter",
      "REP_CODE",
      "TOTAL_LINE_PRICE",
      "TRANS_DATE",
      "TRANSTYPE_CODE",
      "TRANSTYPE_DESC",
      "UNIT_SELL_PRICE",
      "WeekEnd",
      "WeekOfMonth",
      "WeekStart",
      "Year"
    ],
    "properties": {
      "_id": {
        "bsonType": "objectId"
      },
      "CUSTOMER_NUMBER": {
        "bsonType": "string"
      },
      "DOC_NUMBER": {
        "bsonType": "string"
      },
      "FIN_PERIOD": {
        "bsonType": "string"
      },
      "INVENTORY_CODE": {
        "bsonType": "string"
      },
      "LAST_COST": {
        "bsonType": "int"
      },
      "MonthEnd": {
        "bsonType": "date"
      },
      "monthOnlyString": {
        "bsonType": "string"
      },
      "MonthStart": {
        "bsonType": "date"
      },
      "QUANTITY": {
        "bsonType": "int"
      },
      "Quarter": {
        "bsonType": "string"
      },
      "REP_CODE": {
        "bsonType": "string"
      },
      "TOTAL_LINE_PRICE": {
        "bsonType": "int"
      },
      "TRANS_DATE": {
        "bsonType": "date"
      },
      "TRANSTYPE_CODE": {
        "bsonType": "int"
      },
      "TRANSTYPE_DESC": {
        "bsonType": "string"
      },
      "UNIT_SELL_PRICE": {
        "bsonType": "int"
      },
      "WeekEnd": {
        "bsonType": "date"
      },
      "WeekOfMonth": {
        "bsonType": "string"
      },
      "WeekStart": {
        "bsonType": "date"
      },
      "Year": {
        "bsonType": "string"
      }
    }
  }
}
```

SUPPLIERS:

```
{
"$jsonSchema": {
    "bsonType": "object",
    "required": [
      "_id",
      "CREDIT_LIMIT",
      "EXCLSV",
      "NORMAL_PAYTERMS",
      "SUPPLIER_CODE",
      "SUPPLIER_DESC"
    ],
    "properties": {
      "_id": {
        "bsonType": "objectId"
      },
      "CREDIT_LIMIT": {
        "bsonType": "int"
      },
      "EXCLSV": {
        "bsonType": "string"
      },
      "NORMAL_PAYTERMS": {
        "bsonType": "int"
      },
      "SUPPLIER_CODE": {
        "bsonType": "string"
      },
      "SUPPLIER_DESC": {
        "bsonType": "string"
      }
    }
  }
}
```

References

Amazon. s.a. *Relational vs Nonrelational Databases – Difference Between Types of Databases. AWS. Amazon Web Services, Inc.* Available at: https://aws.amazon.com/compare/the-difference-between-relational-and-non-relational-databases/

Anderson, B. & Nicholson, B. 2024. *SQL vs. NoSQL Databases: What's the Difference? IBM.* Available at: https://www.ibm.com/think/topics/sql-vs-nosql

Atlan. 2023. *Relational Database vs NoSQL: 15 Key Differences to Know! Atlan.com.* Available at: https://atlan.com/relational-database-vs-nosql/

Happy. 2025. *Understanding NoSQL: Advantages and Real-World Use Cases. SHIFT ASIA | Dev Blog.* Available at: https://shiftasia.com/community/understanding-nosql-advantages-and-real-world-use-cases/

Hong, S.-S., Lee, J., Chung, S. & Kim, B. 2023. Fast Real-Time Data Process Analysis Based on NoSQL for IoT Pavement Quality Management Platform. *Applied Sciences*, 13(1):658.

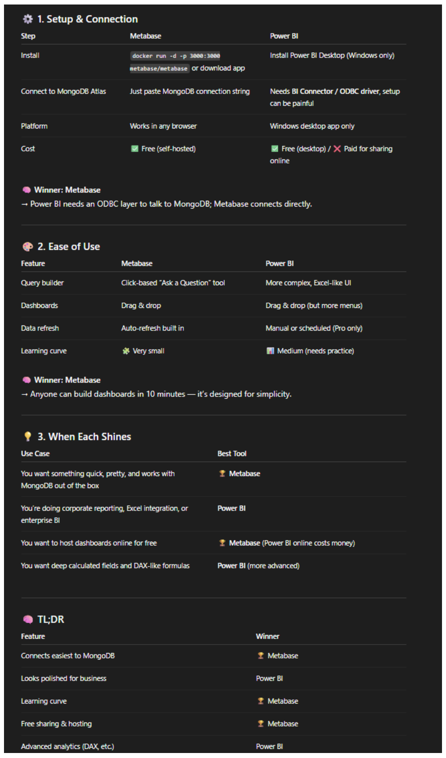
Khan, W., Kumar, T., Zhang, C., Raj, K., Roy, A.M. & Luo, B. 2023. SQL and NoSQL Database Software Architecture Performance Analysis and Assessments—A Systematic Literature Review. *Big Data and Cognitive Computing*, 7(2):97.

9. AI Log

Mostly used AI for technical things, understanding streaming pipeline, creating BI dashboards in Metabase, and manipulating data in MongoDB. Cleaning methods for addressing whitespace, normalizing codes, cleaning, translating, and flattening Excel and MongoDB data for BI, converting financial periods to appropriate date formats, and guaranteeing consistent references. Advice on how to denormalize collections while preserving data integrity, combining headers and line items, embedding versus referring, and schema design issues. Help creating metabase queries.

Memory constraints, query efficiency, and aggregation pipeline techniques were among the performance and optimization issues. Additionally, it helped troubleshooting for Metabase, sharing Docker volumes.

ChatGPT was used to help us compare the differences between BI software and pick the one that suits our needs best. We went forward with its recommendation which we feel was the right choice.



⚙️ 1. Setup & Connection

Step	Metabase	Power BI
Install	`docker run -d -p 3000:3000 metabase/metabase` or download app	Install Power BI Desktop (Windows only)
Connect to MongoDB Atlas	Just paste MongoDB connection string	Needs **BI Connector / ODBC driver**, setup can be painful
Platform	Works in any browser	Windows desktop app only
Cost	✅ Free (self-hosted)	✅ Free (desktop) / ❌ Paid for sharing online

🧠 **Winner: Metabase**
→ Power BI needs an ODBC layer to talk to MongoDB; Metabase connects directly.

🎨 2. Ease of Use

Feature	Metabase	Power BI
Query builder	Click-based "Ask a Question" tool	More complex, Excel-like UI
Dashboards	Drag & drop	Drag & drop (but more menus)
Data refresh	Auto-refresh built in	Manual or scheduled (Pro only)
Learning curve	🧩 Very small	📊 Medium (needs practice)

🧠 **Winner: Metabase**
→ Anyone can build dashboards in 10 minutes — it's designed for simplicity.

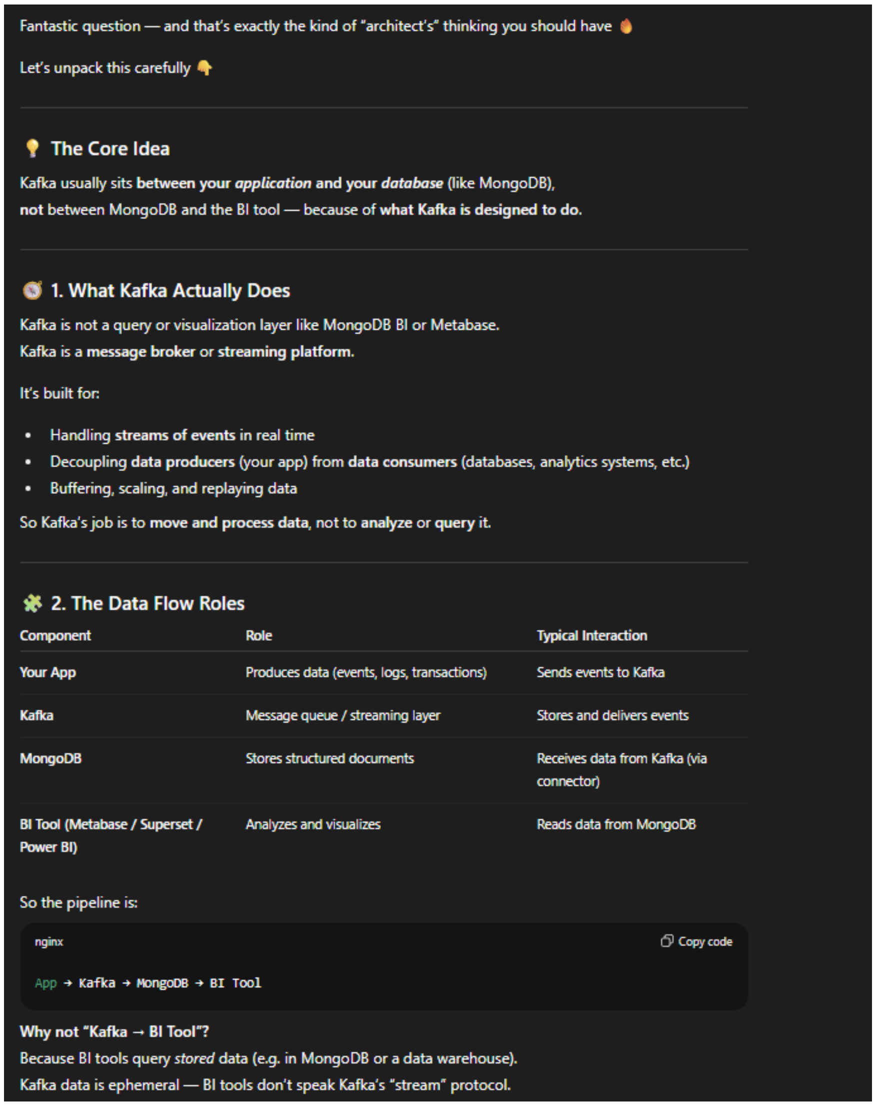
💡 3. When Each Shines

Use Case	Best Tool
You want something quick, pretty, and works with MongoDB out of the box	🏆 **Metabase**
You're doing corporate reporting, Excel integration, or enterprise BI	**Power BI**
You want to host dashboards online for free	🏆 **Metabase** (Power BI online costs money)
You want deep calculated fields and DAX-like formulas	**Power BI** (more advanced)

🧠 TL;DR

Feature	Winner
Connects easiest to MongoDB	🏆 Metabase
Looks polished for business	Power BI
Learning curve	🏆 Metabase
Free sharing & hosting	🏆 Metabase
Advanced analytics (DAX, etc.)	Power BI

ChatGPT was used to understand the pipeline and where streaming fits in which aided in our understanding and implementation.


Fantastic question — and that's exactly the kind of "architect's" thinking you should have 🔥

Let's unpack this carefully 👇

💡 The Core Idea

Kafka usually sits **between your *application* and your *database*** (like MongoDB),
not between MongoDB and the BI tool — because of **what Kafka is designed to do.**

🧭 1. What Kafka Actually Does

Kafka is not a query or visualization layer like MongoDB BI or Metabase.
Kafka is a **message broker** or **streaming platform.**

It's built for:

- Handling **streams of events** in real time
- Decoupling **data producers** (your app) from **data consumers** (databases, analytics systems, etc.)
- Buffering, scaling, and replaying data

So Kafka's job is to **move and process data**, not to **analyze** or **query** it.

🧩 2. The Data Flow Roles

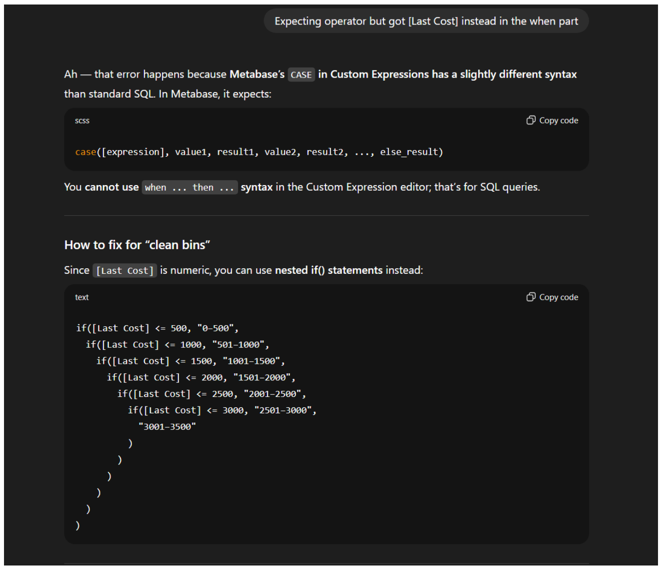
Component	Role	Typical Interaction
Your App	Produces data (events, logs, transactions)	Sends events to Kafka
Kafka	Message queue / streaming layer	Stores and delivers events
MongoDB	Stores structured documents	Receives data from Kafka (via connector)
BI Tool (Metabase / Superset / Power BI)	Analyzes and visualizes	Reads data from MongoDB

So the pipeline is:

```nginx
App → Kafka → MongoDB → BI Tool
```

Why not "Kafka → BI Tool"?
Because BI tools query *stored* data (e.g. in MongoDB or a data warehouse).
Kafka data is ephemeral — BI tools don't speak Kafka's "stream" protocol.

Metabase query assistance to create a histogram



Expecting operator but got [Last Cost] instead in the when part

Ah — that error happens because **Metabase's `CASE` in Custom Expressions has a slightly different syntax** than standard SQL. In Metabase, it expects:

```scss
case([expression], value1, result1, value2, result2, ..., else_result)
```

You **cannot use** `when ... then ...` **syntax** in the Custom Expression editor; that's for SQL queries.

How to fix for "clean bins"

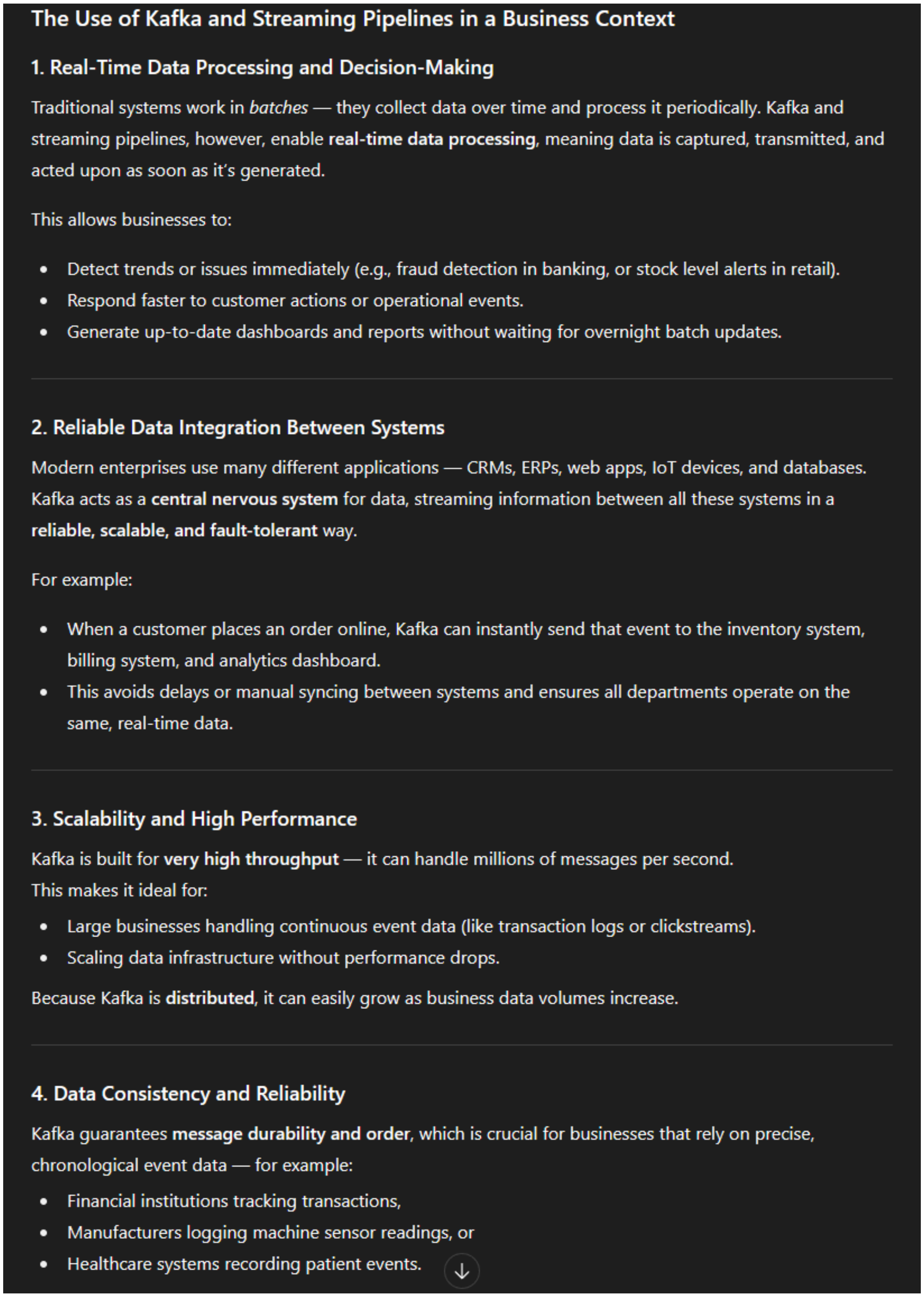
Since `[Last Cost]` is numeric, you can use **nested if() statements** instead:

```text
if([Last Cost] <= 500, "0-500",
  if([Last Cost] <= 1000, "501-1000",
    if([Last Cost] <= 1500, "1001-1500",
      if([Last Cost] <= 2000, "1501-2000",
        if([Last Cost] <= 2500, "2001-2500",
          if([Last Cost] <= 3000, "2501-3000",
            "3001-3500"
          )
        )
      )
    )
  )
)
```

https://chatgpt.com/share/68f1612d-7404-8003-b33e-deb9ad45c001

ChatGPT was used to help create the data aligned to Clearvues unique fiscal year and the aggregate functions in creating this data inside MongoDB compass. AI was used in the creation of scripts to help clean the data and send it to apache kafka. Along with this chatGPT was used in creating a consumer to take the data from kafka and input it into mongoDB.

AI was used in learning about Apache Kafka and how it integrates into the ETL pipeline and the overall project. It also assisted in the use of Apache Kafka and streaming pipelines in the business environment.



The Use of Kafka and Streaming Pipelines in a Business Context

1. Real-Time Data Processing and Decision-Making

Traditional systems work in *batches* — they collect data over time and process it periodically. Kafka and streaming pipelines, however, enable **real-time data processing**, meaning data is captured, transmitted, and acted upon as soon as it's generated.

This allows businesses to:

- Detect trends or issues immediately (e.g., fraud detection in banking, or stock level alerts in retail).
- Respond faster to customer actions or operational events.
- Generate up-to-date dashboards and reports without waiting for overnight batch updates.

---

2. Reliable Data Integration Between Systems

Modern enterprises use many different applications — CRMs, ERPs, web apps, IoT devices, and databases. Kafka acts as a **central nervous system** for data, streaming information between all these systems in a **reliable, scalable, and fault-tolerant** way.

For example:

- When a customer places an order online, Kafka can instantly send that event to the inventory system, billing system, and analytics dashboard.
- This avoids delays or manual syncing between systems and ensures all departments operate on the same, real-time data.

---

3. Scalability and High Performance

Kafka is built for **very high throughput** — it can handle millions of messages per second.
This makes it ideal for:

- Large businesses handling continuous event data (like transaction logs or clickstreams).
- Scaling data infrastructure without performance drops.

Because Kafka is **distributed**, it can easily grow as business data volumes increase.

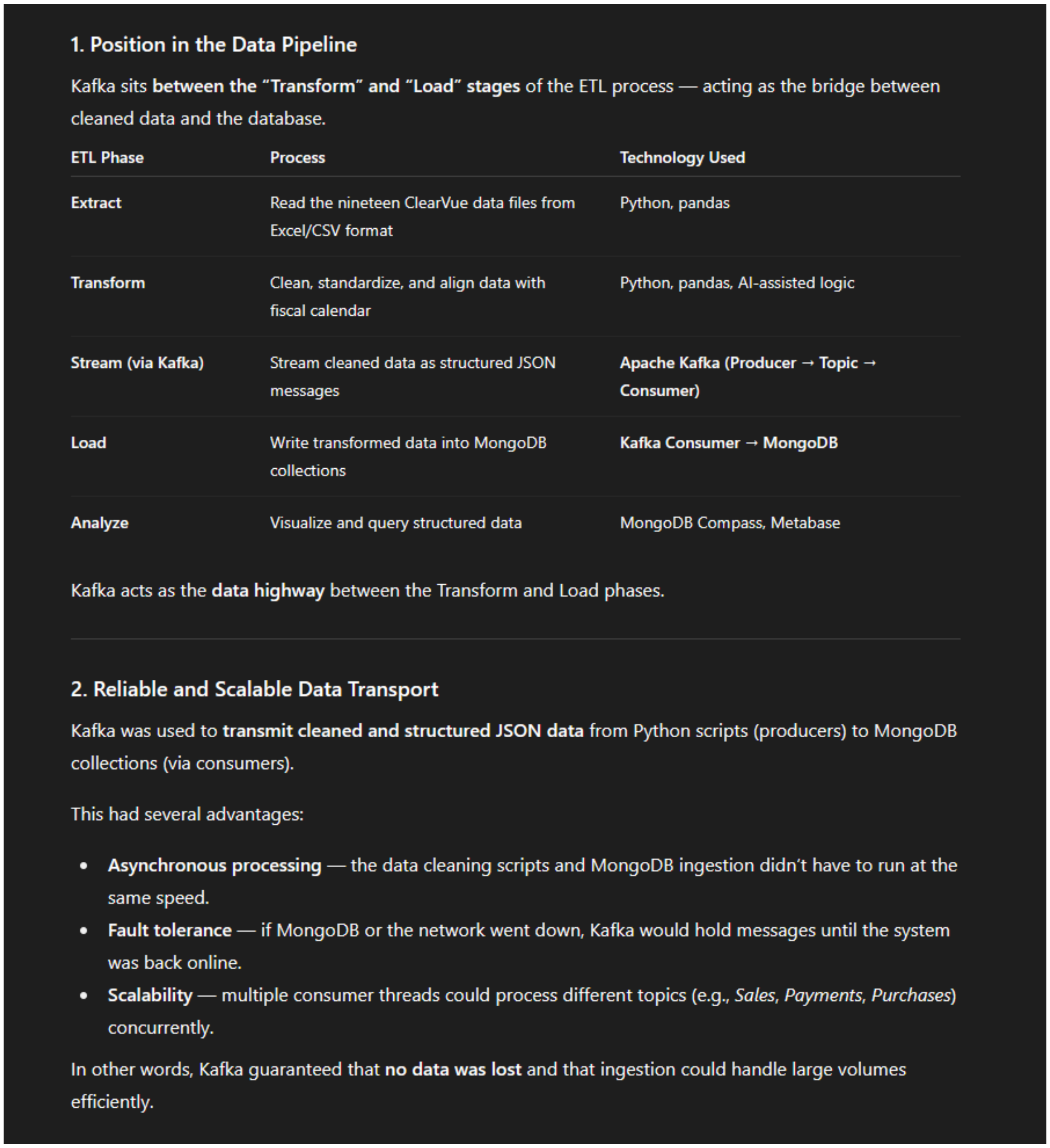
---

4. Data Consistency and Reliability

Kafka guarantees **message durability and order**, which is crucial for businesses that rely on precise, chronological event data — for example:

- Financial institutions tracking transactions,
- Manufacturers logging machine sensor readings, or
- Healthcare systems recording patient events.

ChatGPT Assistance Summary Moon Modeler Project In this session, ChatGPT helped me with several tasks related to database modeling in Moon Modeler. Using Moon Modeler I learned how to create new collections, link them with the Reference (key) tool, and adjust relationship types (1:N, N:1, etc.). Connecting to MongoDB Compass, ChatGPT explained how to connect Moon Modeler to MongoDB using the Compass or Atlas connection string and noted that the free version might not support full synchronization. Building Relationships. After I uploaded my logical diagram, ChatGPT listed all correct relationships between tables (e.g., sales → sales_lines, payments → customers, products → product_categories, etc.) and explained their directions. Exporting to JSON: I asked how to export my model to JSON. ChatGPT showed where to find the Export / Convert options and mentioned that these features may be limited in the free edition. Final Review. After updating my model, ChatGPT reviewed it and confirmed that my relationships and structure were logical and consistent. Finally, ChatGPT helped me rewrite this explanation in a more formal way for my academic submission.



1. Position in the Data Pipeline

Kafka sits **between the "Transform" and "Load" stages** of the ETL process — acting as the bridge between cleaned data and the database.

ETL Phase	Process	Technology Used
Extract	Read the nineteen ClearVue data files from Excel/CSV format	Python, pandas
Transform	Clean, standardize, and align data with fiscal calendar	Python, pandas, AI-assisted logic
Stream (via Kafka)	Stream cleaned data as structured JSON messages	**Apache Kafka (Producer → Topic → Consumer)**
Load	Write transformed data into MongoDB collections	**Kafka Consumer → MongoDB**
Analyze	Visualize and query structured data	MongoDB Compass, Metabase

Kafka acts as the **data highway** between the Transform and Load phases.

---

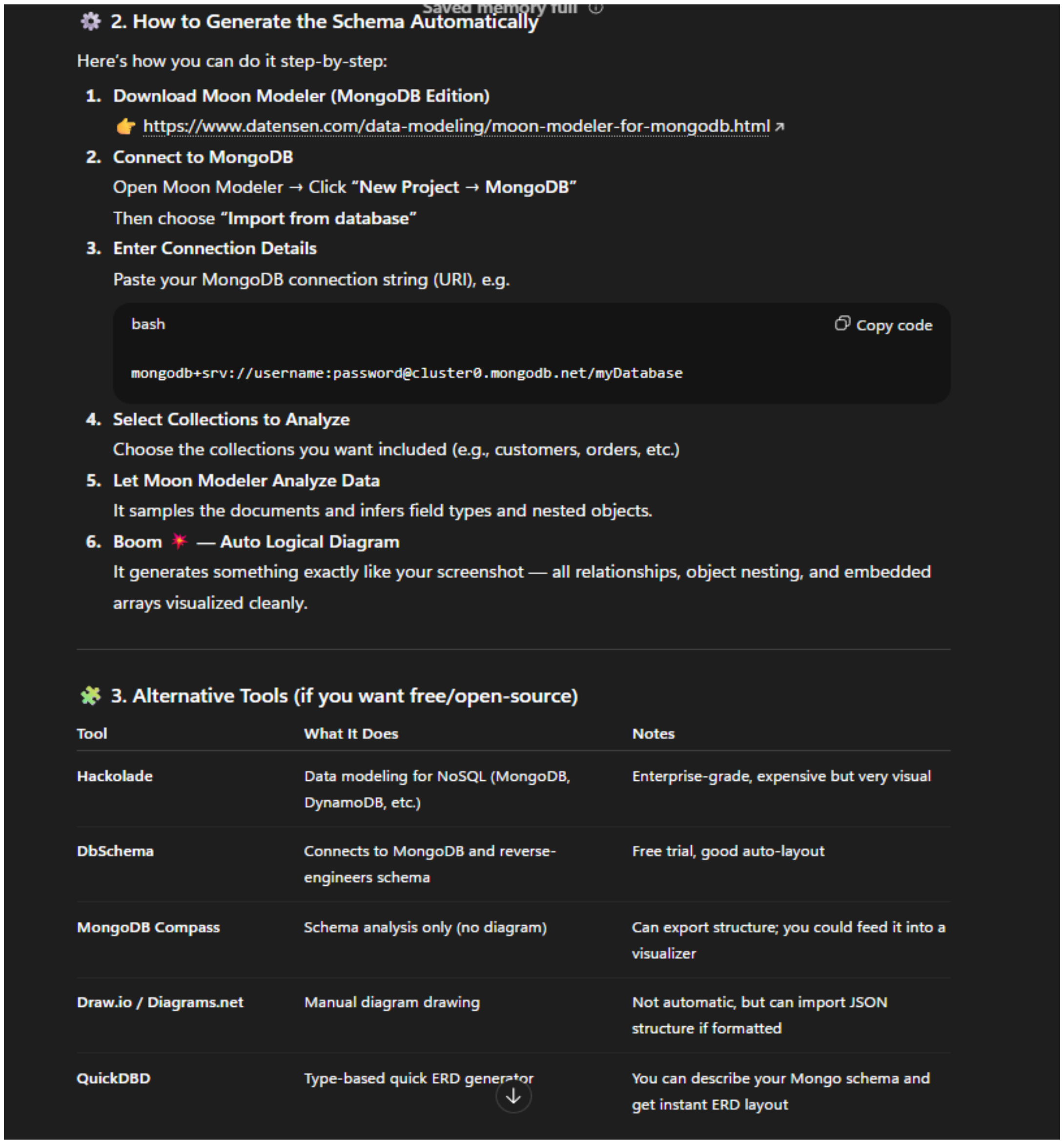
2. Reliable and Scalable Data Transport

Kafka was used to **transmit cleaned and structured JSON data** from Python scripts (producers) to MongoDB collections (via consumers).

This had several advantages:

- **Asynchronous processing** — the data cleaning scripts and MongoDB ingestion didn't have to run at the same speed.
- **Fault tolerance** — if MongoDB or the network went down, Kafka would hold messages until the system was back online.
- **Scalability** — multiple consumer threads could process different topics (e.g., *Sales*, *Payments*, *Purchases*) concurrently.

In other words, Kafka guaranteed that **no data was lost** and that ingestion could handle large volumes efficiently.


Saved memory full

⚙️ 2. How to Generate the Schema Automatically

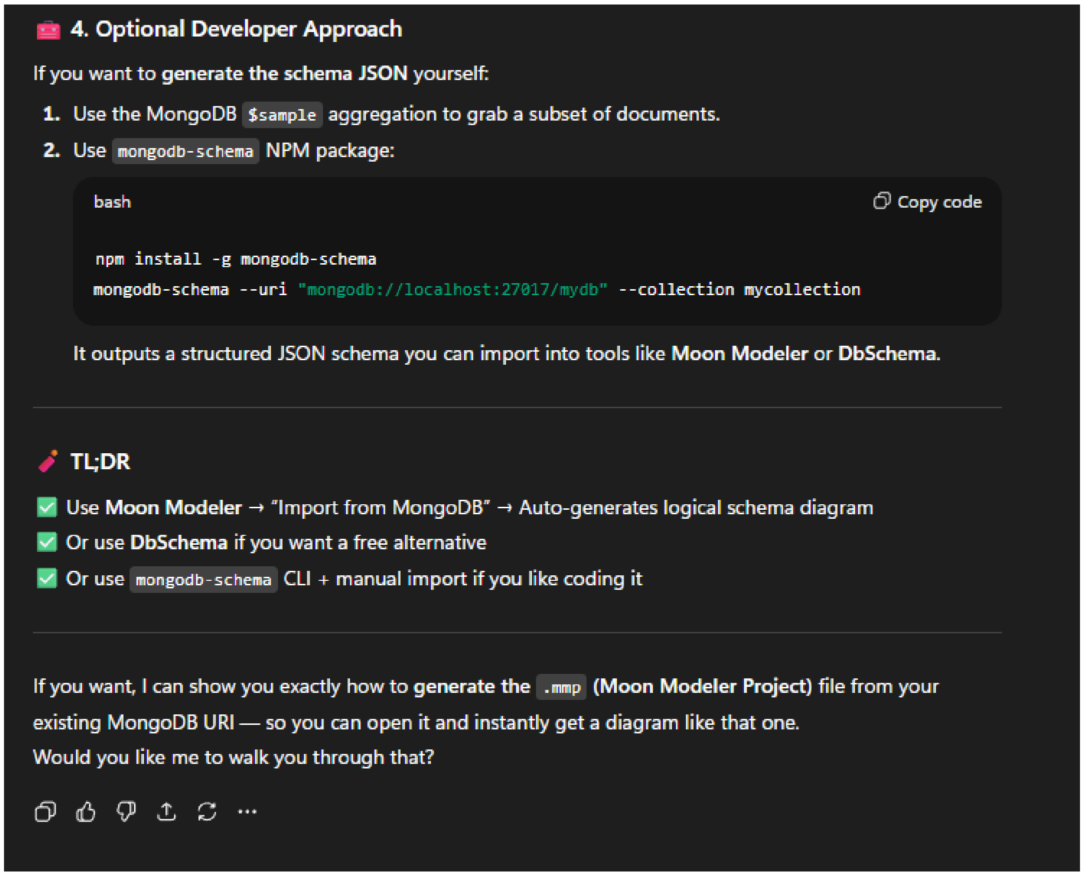
Here's how you can do it step-by-step:

1. **Download Moon Modeler (MongoDB Edition)**
   👉 https://www.datensen.com/data-modeling/moon-modeler-for-mongodb.html ↗
2. **Connect to MongoDB**
   Open Moon Modeler → Click **"New Project → MongoDB"**
   Then choose **"Import from database"**
3. **Enter Connection Details**
   Paste your MongoDB connection string (URI), e.g.

   ```bash
   mongodb+srv://username:password@cluster0.mongodb.net/myDatabase
   ```
4. **Select Collections to Analyze**
   Choose the collections you want included (e.g., customers, orders, etc.)
5. **Let Moon Modeler Analyze Data**
   It samples the documents and infers field types and nested objects.
6. **Boom 💥 — Auto Logical Diagram**
   It generates something exactly like your screenshot — all relationships, object nesting, and embedded arrays visualized cleanly.

🧩 3. Alternative Tools (if you want free/open-source)

Tool	What It Does	Notes
Hackolade	Data modeling for NoSQL (MongoDB, DynamoDB, etc.)	Enterprise-grade, expensive but very visual
DbSchema	Connects to MongoDB and reverse-engineers schema	Free trial, good auto-layout
MongoDB Compass	Schema analysis only (no diagram)	Can export structure; you could feed it into a visualizer
Draw.io / Diagrams.net	Manual diagram drawing	Not automatic, but can import JSON structure if formatted
QuickDBD	Type-based quick ERD generator	You can describe your Mongo schema and get instant ERD layout



🧰 4. Optional Developer Approach

If you want to **generate the schema JSON** yourself:

1. Use the MongoDB `$sample` aggregation to grab a subset of documents.
2. Use `mongodb-schema` NPM package:

```bash
npm install -g mongodb-schema
mongodb-schema --uri "mongodb://localhost:27017/mydb" --collection mycollection
```

   It outputs a structured JSON schema you can import into tools like **Moon Modeler** or **DbSchema**.

🧨 TL;DR

✅ Use **Moon Modeler** → "Import from MongoDB" → Auto-generates logical schema diagram
✅ Or use **DbSchema** if you want a free alternative
✅ Or use `mongodb-schema` CLI + manual import if you like coding it

If you want, I can show you exactly how to **generate the** `.mmp` **(Moon Modeler Project)** file from your existing MongoDB URI — so you can open it and instantly get a diagram like that one.
Would you like me to walk you through that?