

# Group6 - IB9HP0 - Technical Report

## Table of contents

<b>Introduction</b>	2
<b>Part 1: Database Design and Implementation</b>	2
1.1: E-R Diagram Design	2
1.2: SQL Database Schema Creation	3
<b>Part 2: Data Generation and Management</b>	9
2.1: Synthetic Data Generation	9
2.2: Data Import and Quality Assurance	15
<b>Part 3: Data Pipeline Generation</b>	23
3.1: GitHub Repository and Workflow Setup	23
3.2: GitHub Actions for Continuous Integration	23
<b>Part 4: Data Analysis and Reporting with Quarto in R</b>	26
4.1: Advanced Data Analysis in R	26
4.2: Comprehensive Reporting with Quarto	27
<b>Conclusion:</b>	31
<b>Appendix:</b>	31
A1. Data Dictionary	31

## Introduction

This project undertook a comprehensive approach to data management for an apparel e-commerce retailer based in the UK, encompassing database design, data analysis, and reporting. The database architecture was structured around eight entities including products, customers, shipments, promotions, advertisements, suppliers, categories, and transactions, depicted through an Entity-Relationship (ER) diagram. An SQL schema was then implemented to substantiate this design. R was employed to generate synthetic data aligned with our schema to simulate real-world retail transactions. Rigorous data quality assurance was conducted before writing the data into the database. Subsequently, Quarto with R was utilised for data analysis to offer practical insights for strategic decision-making. Automation of data validation, loading, and analysis processes was achieved through a GitHub workflow, ensuring collaborative oversight and accountability throughout the project lifecycle.

## Part 1: Database Design and Implementation

### 1.1: E-R Diagram Design

The E-R diagram contained eight entities and relationships, depicting the operational structure of an apparel e-commerce retailer based in the UK. It illustrated the journey of an order starting from a registered customer selecting a product (SKU), through to the order's delivery, and handling of any subsequent returns. It also covered the process of customer acquisition through three main channels, organic, advertisement and referrals. Please refer to Appendix 1 for the definition of each attribute of each entity.

We made the following assumptions:

- Product is tracked at the SKU level.
- Promotion codes are applied at SKU level.
- Customers must register before placing orders.
- Third-party analytics (e.g., Apps flyer) is used for marketing and customer attribution, thus we can track paid customers are acquired from which ads.
- One order can be paid by only one transaction.
- One new customer can be referred by only one existing customer.
- Customers can only return the order within 30 days from order placement date.
- Every order will be delivered in one shipment.

### **Relationships Between Entities:**

Our E-R diagram had three types of relationships between different entities, one-to-one, many-to-one, and many-to-many.

#### **One-to-one:**

- CUSTOMERS refers CUSTOMERS: A self-recursive relationship, in which one customer can refer another customer.

#### **One-to-many:**

- SUPPLIERS supply SKU: One supplier can supply many products while one SKU is only provided by only one supplier.
- CUSTOMERS acquired through ADS: One advertisement can be used to acquire many customers whilst one customer can only be acquired through one advertisement.
- SKU belongs to CATEGORY: Many products can belong to one category.
- PROMOTIONS apply to SKU: One promotion can be applied to many products.
- CUSTOMERS pay TRANSACTIONS: one transaction can only be paid by one customer while one customer can pay multiple transactions.

#### **Many-to-many:**

- CUSTOMERS order SKU: One customer can order many products and one product can be ordered multiple times.
- SKU delivered through ORDER\_SHIPMENT: Many products can have multiple order shipments.

## **1.2: SQL Database Schema Creation**

In creating logical and physical schema, to satisfy 3NF normalisation, we ensured that (1) each attribute was single-value, (2) each table had its unique primary key, (3) all attributes in all tables represented distinctive information, (4) no inferred data was stored (e.g. calculated field), (5) in a table, except for the primary key, other attributes were not dependent on each other. Subsequently, considerations were made for data volume and performance requirements, selecting appropriate data types and indexes to optimise query performance. Based on these designs, the physical schema was created using the SQL, including table structures, indexes, and constraints. Notably, as ORDER is a many-to-many relationship, an ORDERS table was created in the logical and physical schema.

### **Derivation of Physical Schema**

To derive the physical schema of the database, entity-relationship modeling and normalization were conducted initially. This involved identifying entities, attributes, and relationships, ensuring that the data model adhered to third normal form (3NF). Subsequently, considerations were made for data volume and performance requirements, selecting appropriate data types and indexes to optimize query performance. Finally, based on these designs, the physical schema was created using the SQL language, including table structures, indexes, and constraints.

## **Logical Schema**

**ORDERS** (ORDER\_ID, CUSTOMER\_ID, TRANSACTION\_ID, PRODUCT\_ID, SHIPPING\_ID,  
ORDER\_DATE, ORDER\_STATUS, DELIVERED\_DATE, ORDER\_QUANTITY,  
RETURN\_QUANTITY, RETURN\_DATE)

**ORDER\_SHIPMENT** (SHIPPING\_ID, POST\_CODE, CARRIER)

**TRANSACTION** (TRANSACTION\_ID, PAYMENT\_METHOD, TRANSACTION\_STATUS)

**PROMOTION** (PROMOTION\_ID, PRODUCT\_ID, PROMOTION\_VALUE,  
PROMOTION\_START\_DATE, PROMOTION\_END\_DATE, MINIMUM\_PURCHASE\_AMOUNT)

**ADS** (AD\_ID, PLATFORM, AD\_TITLE, AD\_TYPE, AD\_START\_DATE, AD\_END\_DATE,  
COST\_PER\_CLICK, CLICK\_THROUGH\_RATE, NUMBER\_OF\_CLICKS)

**CUSTOMERS** (CUSTOMER\_ID, AD\_ID, REFERENCE\_ID, CUSTOMER\_EMAIL,  
CUSTOMER\_GENDER, PHONE\_NUMBER, LAST\_NAME, FIRST\_NAME, DATA\_OF\_BIRTH,  
POST\_CODE, HOUSE NUMBER, ACQUISITION\_TYPE)

**CATEGORY** (CATEGORY\_ID, CATEGORY\_NAME, GENDER, AGE\_GROUP)

**SUPPLIER** (SUPPLIER\_ID, SUPPLIER\_NAME, SUPPLIER\_PHONE, POST\_CODE,  
SUPPLIER\_EMAIL)

**SKU** (PRODUCT\_ID, SUPPLIER\_ID, CATEGORY\_ID, PRODUCT\_NAME,  
PRODUCT\_PURCHASING\_PRICE, COLOR, SIZE, MARKUP)

## Physical Schema

```
rm(list=ls())  
library(readr)  
library(RSQLite)  
library(dplyr)
```

Creating a connection to a database

```
my_db <- RSQLite::dbConnect(RSQLite::SQLite(),"/cloud/project/ecommerce.db")
```

- ADS

```
CREATE TABLE ADS(  
  AD_ID VARCHAR(30) PRIMARY KEY,  
  PLATFORM VARCHAR(255),  
  AD_TITLE VARCHAR(255),  
  AD_TYPE VARCHAR(70),  
  AD_START_DATE VARCHAR(15),  
  AD_END_DATE VARCHAR(15),  
  COST_PER_CLICK FLOAT,  
  CLICK_THROUGH_RATE FLOAT,  
  NUMBER_OF_CLICK FLOAT  
);
```

- CATEGORY

```
CREATE TABLE CATEGORY(  
  CATEGORY_NAME TEXT NOT NULL,  
  GENDER TEXT,  
  AGE_GROUP VARCHAR(30),  
  CATEGORY_ID VARCHAR(30) PRIMARY KEY  
);
```

- SUPPLIER

```
CREATE TABLE SUPPLIER (  
  SUPPLIER_NAME VARCHAR(50),  
  SUPPLIER_EMAIL VARCHAR(30),  
  SUPPLIER_PHONE NUMERIC(10),  
  POST_CODE VARCHAR(30),  
  SUPPLIER_ID VARCHAR(30) PRIMARY KEY  
);
```

- CUSTOMERS

```

CREATE TABLE IF NOT EXISTS CUSTOMERS(
  CUSTOMER_ID VARCHAR(30) PRIMARY KEY,
  ACQUISITION_TYPE TEXT,
  REFERENCE_ID VARCHAR(30),
  PHONE_NUMBER NUMERIC(10),
  CUSTOMER_GENDER TEXT,
  DATE_OF_BIRTH VARCHAR(15),
  FIRST_NAME TEXT,
  LAST_NAME TEXT,
  CUSTOMER_EMAIL VARCHAR(30),
  POST_CODE VARCHAR(30),
  HOUSE_NUMBER INT,
  AD_ID VARCHAR(30),
  FOREIGN KEY (AD_ID) REFERENCES ADS (AD_ID)
  FOREIGN KEY (REFERENCE_ID) REFERENCES CUSTOMER(CUSTOMER_ID)
);

```

- PRODUCT

```

CREATE TABLE IF NOT EXISTS SKU(
  COLOR TEXT,
  SIZE TEXT,
  PRODUCT_NAME TEXT,
  PRODUCT_ID VARCHAR(30) PRIMARY KEY,
  PRODUCT_PURCHASING_PRICE FLOAT,
  MARKUP FLOAT,
  SUPPLIER_ID VARCHAR(30),
  CATEGORY_ID VARCHAR(30),
  FOREIGN KEY (SUPPLIER_ID) REFERENCES SUPPLIER(SUPPLIER_ID),
  FOREIGN KEY (CATEGORY_ID) REFERENCES CATEGORY(CATEGORY_ID)
);

```

- PROMOTION

```

CREATE TABLE PROMOTION (
  PROMOTION_ID VARCHAR(30) PRIMARY KEY,
  PROMOTION_VALUE FLOAT,
  PROMOTION_START_DATE VARCHAR(15),
  PROMOTION_END_DATE VARCHAR(15),
  MINIMUM_PURCHASE_AMOUNT FLOAT,
  PRODUCT_ID VARCHAR(30),

```

```
FOREIGN KEY (PRODUCT_ID) REFERENCES SKU(PRODUCT_ID)
);
```

- TRANSACTION

```
CREATE TABLE TRANSACTIONS (
  TRANSACTION_ID VARCHAR(30) PRIMARY KEY,
  PAYMENT_METHOD VARCHAR(50),
  TRANSACTION_STATUS VARCHAR(50)
);
```

- ORDER\_SHIPMENT

```
CREATE TABLE ORDER_SHIPMENT(
  SHIPPING_ID VARCHAR(30) PRIMARY KEY,
  POST_CODE VARCHAR(30),
  CARRIER TEXT
);
```

- ORDERS

```
CREATE TABLE ORDERS (
  ORDER_ID VARCHAR(30),
  CUSTOMER_ID VARCHAR(30),
  ORDER_DATE VARCHAR(15),
  ORDER_STATUS TEXT,
  SHIPPING_ID VARCHAR(30),
  DELIVERY_DATE VARCHAR(15),
  TRANSACTION_ID VARCHAR(30),
  PRODUCT_ID VARCHAR(30),
  ORDER_QUANTITY INTEGER,
  RETURN_QUANTITY INTEGER,
  RETURN_DATE VARCHAR(15),
  PRIMARY KEY (ORDER_ID, PRODUCT_ID, CUSTOMER_ID),
  FOREIGN KEY (PRODUCT_ID) REFERENCES SKU(PRODUCT_ID),
  FOREIGN KEY (CUSTOMER_ID) REFERENCES CUSTOMERS(CUSTOMER_ID),
  FOREIGN KEY (TRANSACTION_ID) REFERENCES TRANSACTIONS(TRANSACTION_ID)
  FOREIGN KEY (SHIPPING_ID) REFERENCES SHIPMENT(SHIPPING_ID)
);
```



## Part 2: Data Generation and Management

### 2.1: Synthetic Data Generation

All synthetic data generation was conducted solely in R and saved into csv files before pushed to GitHub, ensuring adherence to attribute conditionalities and inter-entity connections with the support of LLM. Initially, Mockaroo was explored for data generation, but its high level of randomisation proved more complex compared to R for setting precise rules to control data values (see Figure 1).

Need some mock data to test your app? Mockaroo lets you generate up to 1,000 rows of realistic test data in CSV, JSON, SQL, and Excel formats.

Need more data? Plans start at just \$60/year. Mockaroo is also available as a docker image that you can deploy in your own private cloud.

### Create a custom distribution

Add rules using the Mockaroo formula syntax to create a custom distribution. Each rule must evaluate to true or false. For example, `month == 'August'` or `price > 10`. Each number in the table below represents how often that value will occur relative to other values. A value with '2' will occur twice as often as a value with '1'. All values are assigned a '1' when no rules are matched. Rules are evaluated in the order in which they are defined until a matching rule is found. If a rule is blank, it will match all values.

	Rule	Dress	Shirts	Inner_wear	Activewear	Shoes	Jackets	Gloves	Thermals	Pants	Jeans	Trousers	Formal_Wear	Casuals
×														
↑	gender == 'Male'	0	1	1	1	1	1	1	1	1	1	1	1	1
↓														

or

Figure 1 - Customising data values in Mockaroo

In R, independent entities such as CATEGORY, SUPPLIER, PROMOTION, ADS, SKU and CUSTOMERS were created first. We asked LLM to generate values for CATEGORY\_NAME, PRODUCT\_NAME, SUPPLIER\_NAME, AD\_TITLE, tailored to the fashion retail industry (Figure 2 and 3). The generation of SUPPLIER\_EMAIL and CUSTOMER\_EMAIL utilised a function suggested by LLM (Figure 4). Postal codes were randomly generated from UK postcodes using “PostcodesioR” package drawing on data from Office for National Statistic (Walczak, E., 2021). Customer names were randomly created employing “randomNames” package (Betebenner, D.W., 2021). All numerical fields such as PRODUCT\_PURCHASING\_PRICE, MARKUP, etc were randomised using either “runif” or “sample” functions to ensure realistic value distributions. Furthermore, for CUSTOMERS, REFERENCE\_ID, denoting CUSTOMER\_ID of the referees, was conditioned such that the referred customers were only included in our database after their referees. REFERENCE\_ID was exclusively limited to customers with an ACQUISITION\_TYPE of “Referral” while AD\_ID was randomly assigned from ADS table for customers with an ACQUISITION\_TYPE of “Paid”.

Please refer to Appendix 2 for the full prompt sequences in LLM.

```

# Create name
## Filter out names with apostrophes
filtered_names <- randomNames(nrow(CUSTOMER))
filtered_names <- filtered_names[!grepl("'", filtered_names)]

## If the number of filtered names is less than the number of rows in CUSTOMER, generate additional names
if (length(filtered_names) < nrow(CUSTOMER)) {
  additional_names <- randomNames(nrow(CUSTOMER) - length(filtered_names))
  additional_names <- additional_names[!grepl("'", additional_names)]
  filtered_names <- c(filtered_names, additional_names)
}

## Generate random full names
random_full_names <- sample(filtered_names, nrow(CUSTOMER), replace = TRUE)

#W Split full names into first and last names
split_names <- strsplit(random_full_names, ",")

## Extract first names
CUSTOMER$FIRST_NAME <- sapply(split_names, "[", 2)

## Extract last names
CUSTOMER$LAST_NAME <- sapply(split_names, "[", 1)

# CREATE CUSTOMER_POST_CODE
CUSTOMER$POST_CODE <- sapply(1:nrow(CUSTOMER), function(x) random_postcode()$postcode)

```

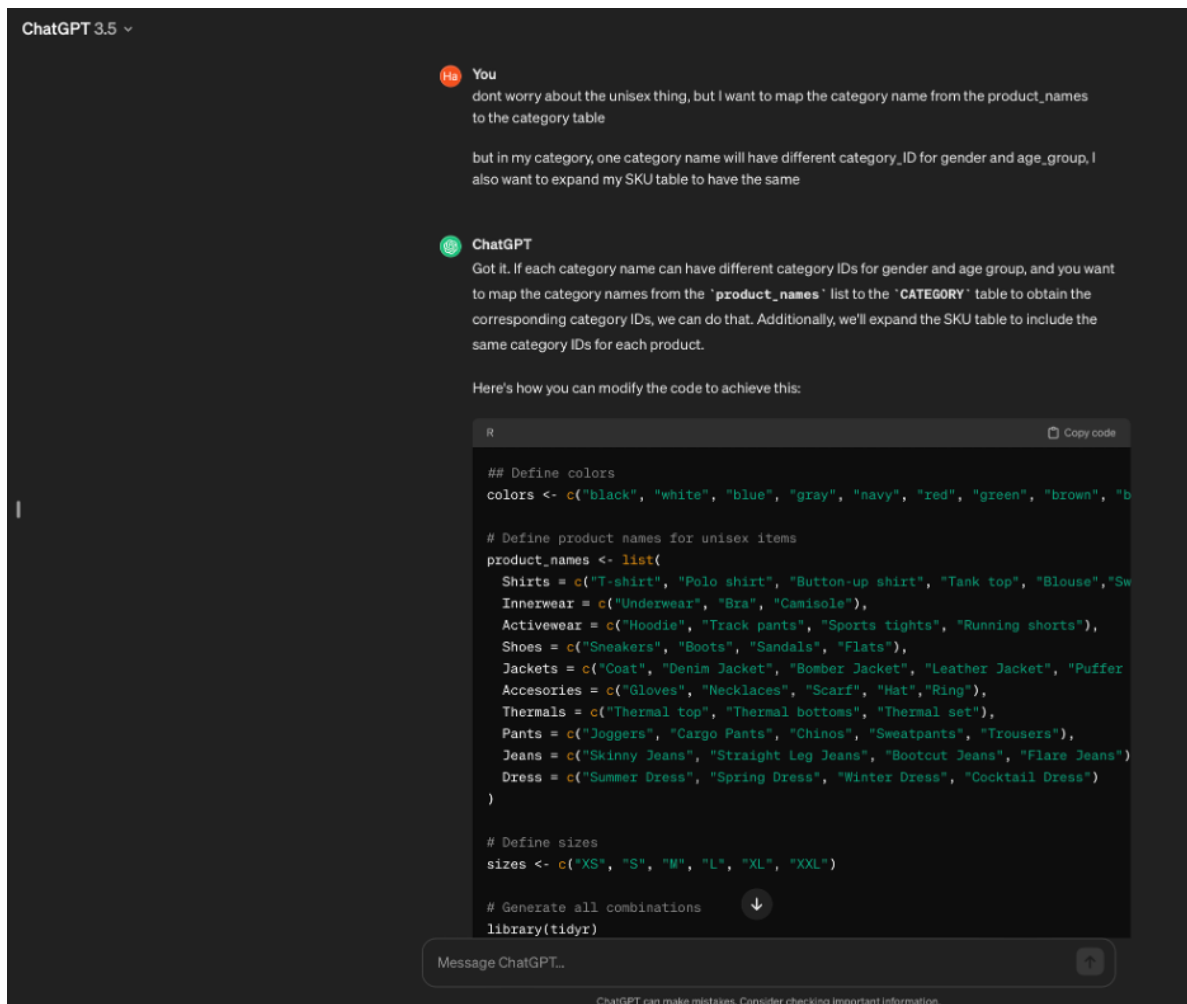


Figure 2 - LLM prompts for product name by category

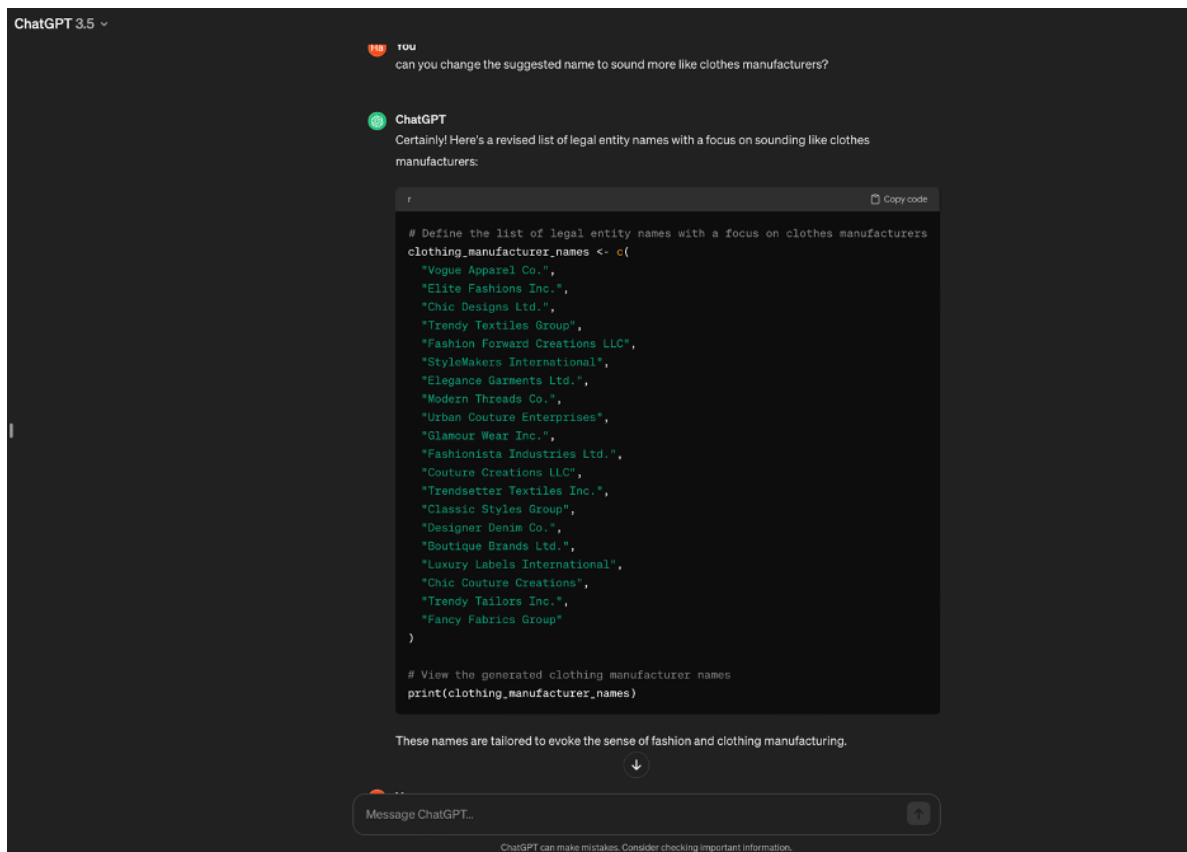


Figure 3 - LLM prompts for supplier names.

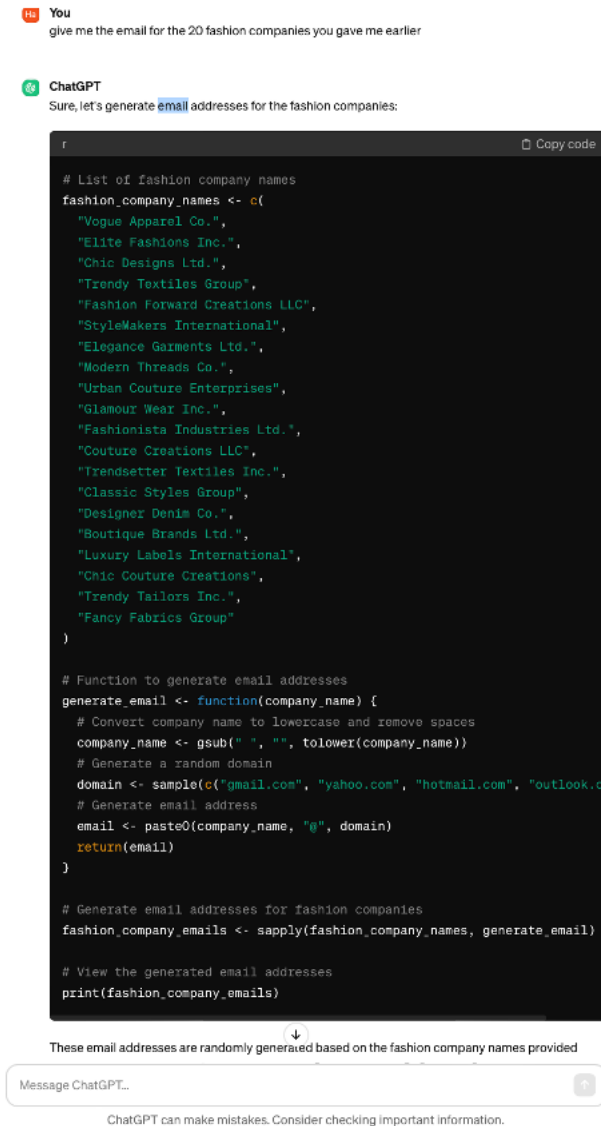


Figure 4 - LLM prompt for email generation

Subsequently, dependent tables including ORDERS, ORDER\_SHIPMENT and TRANSACTIONS were created based on the data from the previous tables. Specifically, in ORDERS, combinations of CUSTOMER\_ID and PRODUCT\_ID were randomly selected from SKU and CUSTOMERS tables so that a customer can purchase multiple products in an order. The generation of DELIVERY\_DATE and RETURN\_DATE adhered to constraints derived from ORDER\_DATE, ensuring that DELIVERY\_DATE fell within a maximum of 7 days after order placement, and RETURN\_DATE occurred after DELIVERY\_DATE while remaining within 30 days from ORDER\_DATE. ORDER\_STATUS was randomised to reflect an authentic dis-

tribution encompassing “Returned”, “Delivered” and “In transit”. RETURN\_QUANTITY and RETURN\_DATE was exclusively assigned to orders marked as “Returned”, with RETURN\_QUANTITY restricted not to exceed ORDER\_QUANTITY. Moreover, for TRANSACTIONS table, we also ensured that all orders have matching successful transactions, with the proportion of failed transaction set to 6% of total transaction numbers.

```
# Create ORDER dataframe
ORDER <- data.frame(ORDER_ID = paste0("OD", seq_len(1000) + 10000),
  stringsAsFactors = TRUE
)

# Add CUSTOMER_ID
ORDER$CUSTOMER_ID <- sample(CUSTOMER$CUSTOMER_ID,nrow(ORDER),replace=TRUE)

# Add ORDER_DATE
ORDER$ORDER_DATE <- sample(seq(start_date, end_date, by = "day"), nrow(ORDER), replace = TRUE)

# Add ORDER_STATUS:
order_status <- c("Delivered", "Returned")
proportions <- c("Delivered" = 0.8, "Returned" = 0.2)

# Sample proportionally with replacement and assign to ORDER$ORDER_STATUS
ORDER$ORDER_STATUS <- sample(order_status, size = nrow(ORDER), replace = TRUE, prob = proportions)
ORDER$ORDER_STATUS[ORDER$ORDER_DATE>'2024-03-05'] = "In transit"

# Add SHIPPING_ID
ORDER$SHIPPING_ID <- ORDER_SHIPMENT$SHIPPING_ID

# Add DELIVERY_DATE

ORDER$DELIVERY_DATE <- ORDER$ORDER_DATE + sample(7,nrow(ORDER),replace=TRUE)
ORDER$DELIVERY_DATE[ORDER$ORDER_STATUS %in% c("Order placed", "In transit")] <- NULL

# Add TRANSACTION_ID
ORDER$TRANSACTION_ID <- TRANSACTION$TRANSACTION_ID[TRANSACTION$TRANSACTION_STATUS == "Successful"]

# Create an empty dataframe to store order-product mappings
order_product_mapping <- data.frame(ORDER_ID = character(), PRODUCT_ID = character(), stringsAsFactors = FALSE)

# Define the number of products per order
products_per_order <- round(runif(nrow(ORDER), min = 1, max = 10))
```

```

# Loop through each order ID and sample products
for (i in 1:nrow(ORDER)) {
  order_id <- ORDER$ORDER_ID[i]
  product_ids <- sample(SKU$PRODUCT_ID, size = products_per_order[i], replace = FALSE)
  order_product_mapping <- rbind(order_product_mapping, data.frame(ORDER_ID = rep(order_id,
}

# Merge order_product_mapping with ORDER dataframe to retain other order details
ORDER <- merge(ORDER, order_product_mapping, by = "ORDER_ID", all.x = TRUE)

# Add ORDER_QUANTITY
ORDER$ORDER_QUANTITY <- round(runif(nrow(ORDER), min=1, max=5), 0)

# Add RETURN_QUANTITY
ORDER$RETURN_QUANTITY <- ifelse(ORDER$ORDER_STATUS=="Returned", mapply(function(x) sample(1:
# Add RETURN_DATE
ORDER$RETURN_DATE <- ORDER$ORDER_DATE + sample(9:30, nrow(ORDER), replace=TRUE)
ORDER$RETURN_DATE[ORDER$ORDER_STATUS != "Returned"] <- ""

# Change DATE to CHARACTER
ORDER$ORDER_DATE <- as.character(ORDER$ORDER_DATE)
ORDER$RETURN_DATE <- as.character(ORDER$RETURN_DATE)
ORDER$DELIVERY_DATE <- as.character(ORDER$DELIVERY_DATE)

# Save to csv
write.csv(ORDER[1:15,], file = file.path("/cloud/project/Data", "ORDERS.1.csv"), row.names = F)
write.csv(ORDER[16:nrow(ORDER),], file = file.path("/cloud/project/Data", "ORDERS.2.csv"), row.names = F)

```

## 2.2: Data Import and Quality Assurance

An R script was developed to load all data files in their respective corresponding tables within our database, ensuring robust data quality and integrity. This process involved connecting to the database and iteratively validating data entries against predefined criteria.

The script employed a loop cross-checking primary key values to prevent duplications and enforcing strict validation rules on a row by row basis. Entries were scrutinised for null values, adherence to formatting standards for phone numbers and emails, and non-negativity in numeric fields. For formatting check, we defined functions to automatically validate. Data entries satisfying these validation criteria were then appended to the database whilst problematic entries were recorded in “error\_log.txt” file for further reviews.

```

library(readr)
library(RSQLite)
library(dplyr)
library(DBI)

# Define a function to check if phone numbers are of length 10 and contain only numeric characters
validate_phone_numbers <- function(phone) {
  phone <- as.character(phone) # Convert integer to character
  if (!is.na(phone) && nchar(phone) == 10 && !any(is.na(as.numeric(phone)))) {
    return(TRUE) # Phone number is valid
  } else {
    return(FALSE) # Phone number is not valid
  }
}

# Function to validate email addresses
email_pattern <- "^[a-zA-Z0-9._%+-]+@[a-zA-Z0-9.-]+\\.[a-zA-Z]{2,}$"
validate_emails <- function(email) {
  grepl(email_pattern, email)
}

# Function to validate that numeric attributes are non-negative
check_non_negative_numeric <- function(data) {
  numeric_cols <- sapply(data, is.numeric)
  negative_values <- sapply(data[numeric_cols], function(col) any(col < 0))
  return(!any(negative_values))
}

# Establishing the connection to db
my_db <- RSQLite::dbConnect(RSQLite::SQLite(), "ecommerce.db")

file_paths <- list(
  "ADS" = list.files(path = "Data", pattern = "ADS.*\\.csv$", full.names = TRUE),
  "CATEGORY" = list.files(path = "Data", pattern = "CATEGORY.*\\.csv$", full.names = TRUE),
  "SUPPLIER" = list.files(path = "Data", pattern = "SUPPLIER.*\\.csv$", full.names = TRUE),
  "CUSTOMERS" = list.files(path = "Data", pattern = "CUSTOMERS.*\\.csv$", full.names = TRUE),
  "SKU" = list.files(path = "Data", pattern = "SKU.*\\.csv$", full.names = TRUE),
  "PROMOTION" = list.files(path = "Data", pattern = "PROMOTION.*\\.csv$", full.names = TRUE),
  "TRANSACTIONS" = list.files(path = "Data", pattern = "TRANSACTIONS.*\\.csv$", full.names = TRUE),
  "ORDER_SHIPMENT" = list.files(path = "Data", pattern = "ORDER_SHIPMENT.*\\.csv$", full.names = TRUE)
)

```



```

"ORDERS" = list.files(path = "Data", pattern = "ORDERS.*\\.csv$", full.names = TRUE)
)

tables <- list(
  "ADS" = "AD_ID",
  "CATEGORY" = "CATEGORY_ID",
  "SUPPLIER" = "SUPPLIER_ID",
  "CUSTOMERS" = "CUSTOMER_ID",
  "SKU" = "PRODUCT_ID",
  "PROMOTION" = "PROMOTION_ID",
  "TRANSACTIONS" = "TRANSACTION_ID",
  "ORDER_SHIPMENT" = "SHIPPING_ID",
  "ORDERS" = c("ORDER_ID", "PRODUCT_ID", "CUSTOMER_ID")
)

# Define write_errors function with folder path argument
write_errors <- function(errors, folder_path, file_name) {
  # Ensure the folder exists, if not, create it
  if (!dir.exists(folder_path)) {
    dir.create(folder_path, recursive = TRUE)
  }

  file_path <- file.path(folder_path, file_name)

  if (length(errors) > 0) {
    cat("Errors:\n", file = file_path)
    for (error in errors) {
      cat(error, "\n", file = file_path, append = TRUE)
    }
    cat("\n", file = file_path, append = TRUE)
  }
}

# List to store errors
error_list <- c()

# Function to check if data entries exist and load new entries
for (table_name in names(tables)) {
  for (file_path in file_paths[[table_name]]) {
    table_data <- read_csv(file_path, n_max = Inf)

    ## Apply specific rules for attributes based on the table

```

```

if (table_name == "ADS") {
  # Initialize error list
  error_list <- vector("list")

  # Convert AD_START_DATE and AD_END_DATE to character
  table_data$AD_START_DATE <- as.character(table_data$AD_START_DATE)
  table_data$AD_END_DATE <- as.character(table_data$AD_END_DATE)

  # Initialize vector to store indices of invalid rows
  invalid_rows <- vector("numeric")

  # Ensure numeric attributes are non-negative
  numeric_attrs <- c("COST_PER_CLICK", "CLICK_THROUGH_RATE", "NUMBER_OF_CLICK")
  for (attr in numeric_attrs) {
    if (any(table_data[[attr]] < 0)) {
      error_list <- c(error_list, paste("Negative values found in", attr, "column of ADS"))
      invalid_rows <- c(invalid_rows, which(table_data[[attr]] < 0))
    }
  }

  # Remove invalid rows from table_data
  if (length(invalid_rows) > 0) {
    table_data <- table_data[-invalid_rows, ]
  }
}

if (table_name == "CUSTOMERS") {
  # Initialize error list
  error_list <- vector("list")

  # Convert DATE_OF_BIRTH to character
  table_data$DATE_OF_BIRTH <- as.character(table_data$DATE_OF_BIRTH)

  # Initialize vector to store indices of invalid rows
  invalid_rows <- vector("numeric")

  # Check phone numbers and emails
  for (i in 1:nrow(table_data)) {
    if (!validate_phone_numbers(table_data$PHONE_NUMBER[i])) {
      error_list <- c(error_list, paste("Invalid phone number in CUSTOMERS table:", table_data$PHONE_NUMBER[i]))
      invalid_rows <- c(invalid_rows, i)
    }
  }
}

```

```

    if (!validate_emails(table_data$CUSTOMER_EMAIL[i])) {
      error_list <- c(error_list, paste("Invalid email in CUSTOMERS table:", table_data$
      invalid_rows <- c(invalid_rows, i)
    }
  }

  # Remove invalid rows from table_data
  if (length(invalid_rows) > 0) {
    table_data <- table_data[-invalid_rows, ]
  }
}

if (table_name == "SKU") {
  # Initialize error list
  error_list <- vector("list")

  # Ensure numeric attributes are non-negative
  numeric_attrs <- c("MARKUP", "PRODUCT_PURCHASING_PRICE")
  for (attr in numeric_attrs) {
    if (any(table_data[[attr]] < 0)) {
      error_list <- c(error_list, paste("Negative values found in", attr, "column of SKU
      invalid_rows <- c(invalid_rows, which(table_data[[attr]] < 0))
    }
  }

  # Remove invalid rows from table_data
  if (length(invalid_rows) > 0) {
    table_data <- table_data[-invalid_rows, ]
  }
}

if (table_name == "PROMOTION") {
  # Initialize error list
  error_list <- vector("list")

  # Convert PROMOTION_START_DATE and PROMOTION_END_DATE to character
  table_data$PROMOTION_START_DATE <- as.character(table_data$PROMOTION_START_DATE)
  table_data$PROMOTION_END_DATE <- as.character(table_data$PROMOTION_END_DATE)

  # Initialize vector to store indices of invalid rows
  invalid_rows <- vector("numeric")

  # Ensure numeric attributes are non-negative
  numeric_attrs <- c("MINIMUM_PURCHASE_AMOUNT")

```

```

for (attr in numeric_attrs) {
  if (any(table_data[[attr]] < 0)) {
    error_list <- c(error_list, paste("Negative values found in", attr, "column of PRO",
    invalid_rows <- c(invalid_rows, which(table_data[[attr]] < 0))
  }
}

# Remove invalid rows from table_data
if (length(invalid_rows) > 0) {
  table_data <- table_data[-invalid_rows, ]
}
}

if (table_name == "SUPPLIER") {
  # Initialize error list
  error_list <- vector("list")

  # Initialize vector to store indices of invalid rows
  invalid_rows <- vector("numeric")

  # Check phone numbers and emails
  for (i in 1:nrow(table_data)) {
    if (!validate_phone_numbers(table_data$SUPPLIER_PHONE[i])) {
      error_list <- c(error_list, paste("Invalid phone number in SUPPLIER table:", table_data$SUPPLIER_PHONE[i]))
      invalid_rows <- c(invalid_rows, i)
    }

    if (!validate_emails(table_data$SUPPLIER_EMAIL[i])) {
      error_list <- c(error_list, paste("Invalid email in SUPPLIER table:", table_data$SUPPLIER_EMAIL[i]))
      invalid_rows <- c(invalid_rows, i)
    }
  }

  # Remove invalid rows from table_data
  if (length(invalid_rows) > 0) {
    table_data <- table_data[-invalid_rows, ]
  }
}

if (table_name == "ORDERS") {
  # Initialize error list
  error_list <- vector("list")

  # Convert ORDER_DATE, DELIVERY_DATE, RETURN_DATE to character

```

```

table_data$ORDER_DATE <- as.character(table_data$ORDER_DATE)
table_data$DELIVERY_DATE <- as.character(table_data$DELIVERY_DATE)
table_data$RETURN_DATE <- as.character(table_data$RETURN_DATE)

# Initialize vector to store indices of invalid rows
invalid_rows <- vector("numeric")

# Check numeric attributes for non-negativity
numeric_attrs <- c("ORDER_QUANTITY", "RETURN_QUANTITY")
for (attr in numeric_attrs) {
  if (any(!is.na(table_data[[attr]]) & table_data[[attr]] < 0)) {
    error_list <- c(error_list, paste("Invalid or negative values found in", attr, "column"))
    invalid_rows <- c(invalid_rows, which(!is.na(table_data[[attr]]) & table_data[[attr]] < 0))
  }
}

# Remove invalid rows from table_data
if (length(invalid_rows) > 0) {
  table_data <- table_data[-invalid_rows, ]
}

## Check for primary key duplication
for (i in seq(nrow(table_data))) {
  new_record <- table_data[i, ]
  pk_columns <- tables[[table_name]]
  pk_values <- new_record[pk_columns]
  # Check if primary key values are non-null
  if (any(is.na(pk_values))) {
    error_list <- c(error_list, paste("Null primary key value found in", table_name, "table"))
    next # Skip to the next record if primary key is null
  }

  conditions <- paste(pk_columns, "=", paste0("'", pk_values, "'"), collapse = " AND ")

  key_exists <- dbGetQuery(my_db, paste("SELECT COUNT(*) FROM", table_name, "WHERE", conditions))

  if (key_exists == 0) {
    tryCatch({
      RSQLite::dbAppendTable(my_db, table_name, new_record)
    }, error = function(e) {
      error_list <- c(error_list, paste("Error inserting record with primary key", paste(pk_columns, "=", paste0("'", pk_values, "'"), collapse = " AND ")
    })
  }
}

```

```

        print(paste("Error inserting record with primary key", paste(pk_values, collapse = " "), "\n", e))
        print(e)
    })
} else {
    print(paste("Record with primary key", paste(pk_values, collapse = " "), "\n", "already exists"))
}
}
}

# Save errors to a folder named "Error logs" within the current directory
write_errors(error_list, "Error logs", "error_log.txt")

```

## Part 3: Data Pipeline Generation

### 3.1: GitHub Repository and Workflow Setup

We initiated our project by creating a new Git repository, connecting Posit Cloud/ RStudio to the repository and uploading essential files, including (1) database, (2) data schema, (3) synthetic data generation, (4) data validation and database writing, (5) data query and analysis scripts. This setup allows us to efficiently track changes and revert to previous versions as needed.

### 3.2: GitHub Actions for Continuous Integration

To automate our project's operations, we implemented a GitHub Actions workflow, detailed through the following key components:

- **Trigger:** Activated by either a push event to the main branch or a scheduled run every 24 hours, ensuring real-time integration of contributions.
- **Runner:** Utilise the most recent Ubuntu environment to execute the job.
- **Steps:** Comprising eight sequential tasks, each step executes a specific operation within the job:
- **Repository Checkout:** Clone the project's code into the runner, providing a foundation for subsequent tasks.
- **R Environment Setup:** Prepare the R environment, ensuring all R-based operations can be performed without hitches.
- **R Package Caching:** Preserve installed R packages between runs, significantly reducing setup time by bypassing redundant installations.
- **Package Installation:** Engage only if the cache does not contain the necessary packages, ensuring all dependencies are available for the script execution.
- **Script Execution:** Run our R scripts from the repository to validate and load satisfying data entries to the database, subsequently creating analyses.
- **Add Changes:** Scans the project's database for any changes following the script's execution and notifies with a "Changes found" message if updates are identified. Meanwhile, new analyses are automatically generated and saved to folder "figures".
- **Commit Changes:** If changes are detected, this step prepares and commits the updated files, maintaining a current state within the repository.
- **Push Updates:** Conclude the workflow by uploading the latest commit to the repository, ensuring all changes are synchronised and stored.

```

name: ETL workflow

on:
  push:
    branches: [ main ]
  schedule:
    - cron: '0 */24 * * *' # Runs every 24 hours

jobs:
  build:
    runs-on: ubuntu-latest
    steps:
      - name: Checkout code
        uses: actions/checkout@v2

      - name: Setup R environment
        uses: r-lib/actions/setup-r@v2
        with:
          r-version: '4.2.0'

      - name: Cache R packages
        uses: actions/cache@v2
        with:
          path: ${ env.R_LIBS_USER }
          key: ${ runner.os }-r-${ hashFiles('*/lockfile') }
          restore-keys: |
            ${ runner.os }-r-

      - name: Create figures directory
        run: mkdir -p figures

      - name: Install packages
        if: steps.cache.outputs.cache-hit != 'true'
        run: |
          Rscript -e 'install.packages(c("ggplot2", "dplyr", "readr", "RSQLite", "DBI"))'

      - name: Execute R script
        run: |
          Rscript R_codes/Workflow.R

      - name: Execute Data Analysis
        run: |

```



```

    Rscript R_codes/Query_workflow.R
- name: Add database changes and commit
run: |
    git config --global user.email "Teng-Yi.Chen@warwick.ac.uk"
    git config --global user.name "DylanCTY"
    git add ecommerce.db
    git add --all figures/
    git commit -m "Update database" || echo "No changes to commit"
    # Check if error logs exist
    if [ -d "Error logs" ] && [ "$(ls -A Error\ logs)" ]; then
        echo "Error logs exist"
        # Create error logs folder
        mkdir -p "Error logs"
        # Write error_log.txt
        echo "Errors:" > "Error logs/error_log.txt"
        # Append each error to the error_log.txt file
        for error in Error\ logs/*; do
            cat "$error" >> "Error logs/error_log.txt"
            echo "" >> "Error logs/error_log.txt" # Add a newline after each error
        done
        # Commit error logs
        git add "Error logs/error_log.txt"
        git commit -m "Add error log" || echo "No error log changes to commit"
    else
        echo "No error logs found"
    fi

- name: Push changes
uses: ad-m/github-push-action@v0.6.0
with:
    github_token: ${ secrets.GITHUB_TOKEN }
    branch: main

```

## Part 4: Data Analysis and Reporting with Quarto in R

### 4.1: Advanced Data Analysis in R

Once the database was updated with the newly generated data via GitHub automation, the data was analysed utilising the R package packages dplyr, GGplot2, and tidyr in conjunction with the SQL DQL command. The procedure entailed retrieving the data and converting it into a format suitable for subsequent analysis. The complete procedure is outlined below.

#### Data Query

The data from the database was obtained using a SQL Data Query Language (DQL) statement. By utilising the calculated function and aggregate command, the data were converted into a format and value suitable for analysis. The following example is a query that displays the top 10 revenue earned by product SKUs in the previous 30 days, calculated by multiplying unit sales with the selling price.

```
# Query data from database and save into R data frame using RSQLite::dbGetQuery
Query_1 <- "WITH PRODUCTS AS(
  SELECT PRODUCT_ID, SALES
  FROM
  (
    SELECT
      *,
      ROW_NUMBER() OVER (ORDER BY SALES DESC) AS row_num
    FROM
    (
      SELECT PRODUCT_ID, sum(order_quantity) as SALES
      FROM ORDERS
      WHERE DATE(ORDER_DATE) >= date('now','-30 days')
      AND ORDER_STATUS = 'Delivered'
      GROUP BY PRODUCT_ID
    ) T1
    ) T2
  WHERE T2.row_num <= 10
)
SELECT T1.PRODUCT_ID, T2.PRODUCT_NAME, T1.SALES, T1.SALES*T2.SELLING_PRICE AS REVENUE
FROM PRODUCTS T1
LEFT JOIN (
SELECT DISTINCT PRODUCT_ID, PRODUCT_NAME, MARKUP * PRODUCT_PURCHASING_PRICE + PRODUCT_PURCHASING_PRICE AS SELLING_PRICE
FROM SKU ) T2
on T1.PRODUCT_ID = T2.PRODUCT_ID"

top_10_SKUs <- RSQLite::dbGetQuery(my_db, Query_1)
```

## Data Manipulation

Once the data was imported into a R data frame, data manipulation was carried out using R to prepare the data for visualisation. The following code example generates a new column containing the SKU description, followed by a reordering by unit sold.

```
# Create product description name of each product_ID for using in analysis
top_10_SKUs$name_sku <- paste(top_10_SKUs$PRODUCT_NAME, top_10_SKUs$PRODUCT_ID, sep = "_")

top_10_SKUs$PRODUCT_ID <- factor(top_10_SKUs$PRODUCT_ID,
levels = top_10_SKUs$PRODUCT_ID[order(top_10_SKUs$SALES)])
```

## Data visualisation

Finally, once we retrieved and formatted the data for analysis, we used the ggplot2 package to create visual representations of the data. The code snippet shown below demonstrates a bar chart that visually represents the top 10 SKUs with the highest sales in the past 30 days, as obtained from the preceding phases of our query.

```
ggplot(top_10_returned_SKUs, aes(x = PRODUCT_ID, y = as.numeric(SALES))) +
  geom_bar(stat = "identity", fill = "steelblue") +
  labs(x = "Product Names", y = "Units Returned", title = "Top 10 Most Returned SKUs",
  subtitle = "Labels indicate the money to be refunded back to the customers") +
  coord_flip() +
  theme(axis.text.x = element_text(hjust = 1)) +
  scale_x_discrete(labels = top_10_returned_SKUs$name_sku) +
  scale_y_continuous(breaks = seq(0, max(as.numeric(top_10_returned_SKUs$SALES)),
  by = 1)) +
  geom_text(aes(label = paste("&", round(REVENUE), sep = "")), hjust = -0.2)
```

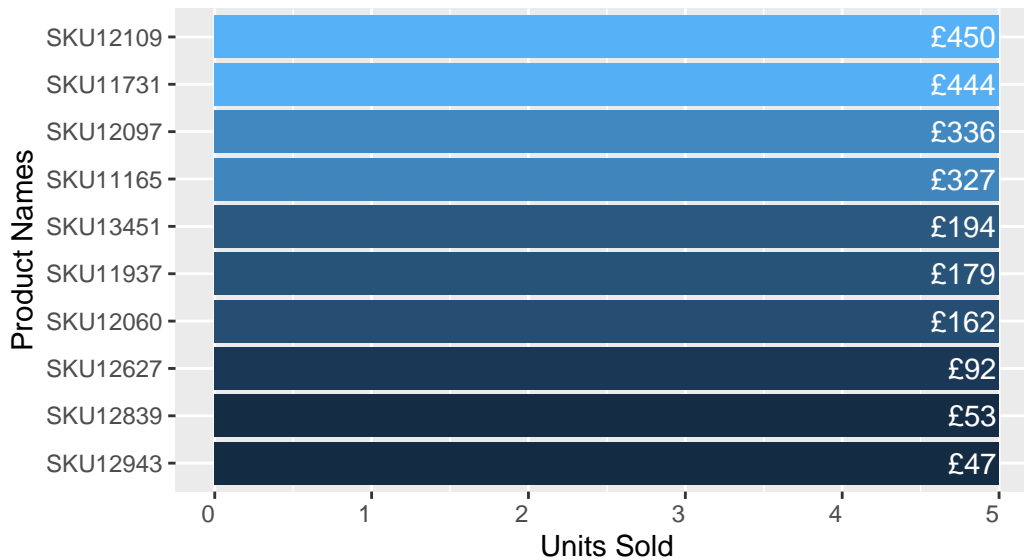
## 4.2: Comprehensive Reporting with Quarto

To present the data analysis, we use the R Quarto report, which can be easily changed to reflect the latest dataset. The analysis was divided into two parts. The first is a short-term analysis for monitoring current performance in the last thirty days. The second is performing long-term analysis, where we analyse changes on a yearly basis to demonstrate long-term progress.

### Short-Term Analysis

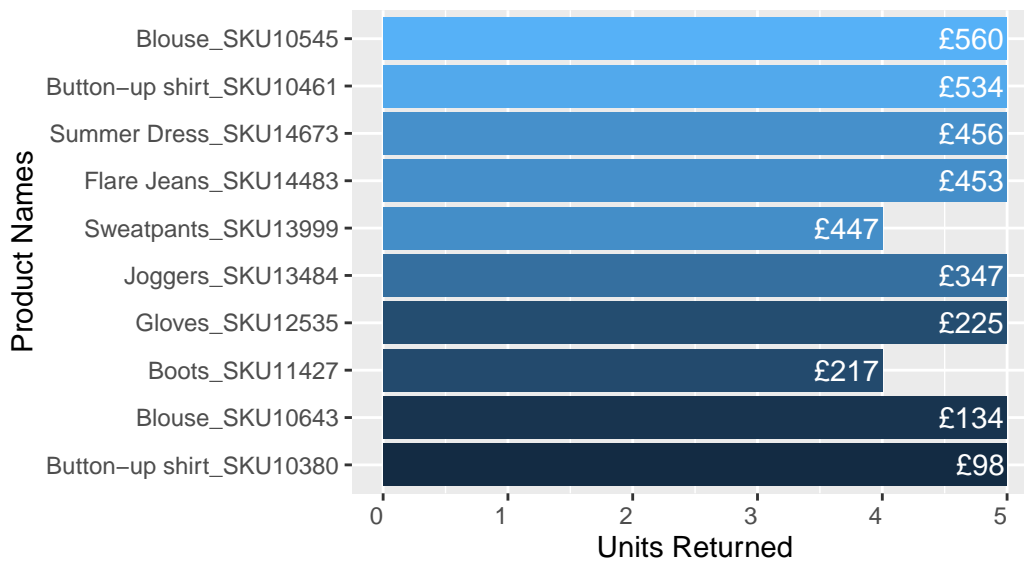
### Top 10 Most Sold SKUs in Units

Labels indicate the revenue generated by the SKUs



### Top 10 Most Returned SKUs

Labels indicate the money to be refunded back to the customer



### Top Supplier with Most Returned Products

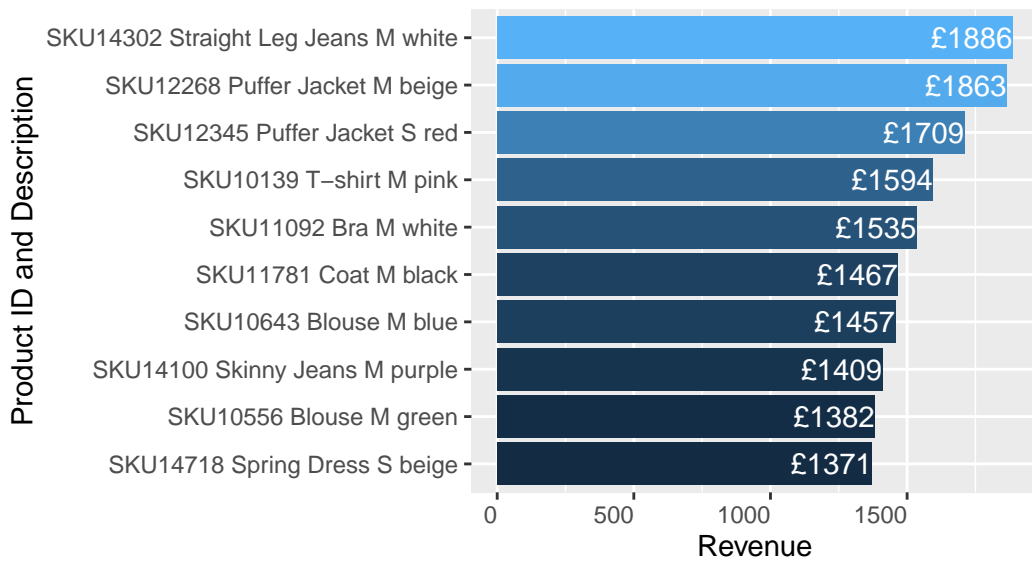
Labels indicate the cost incurred in purchasing the item



### Long-Term Analysis

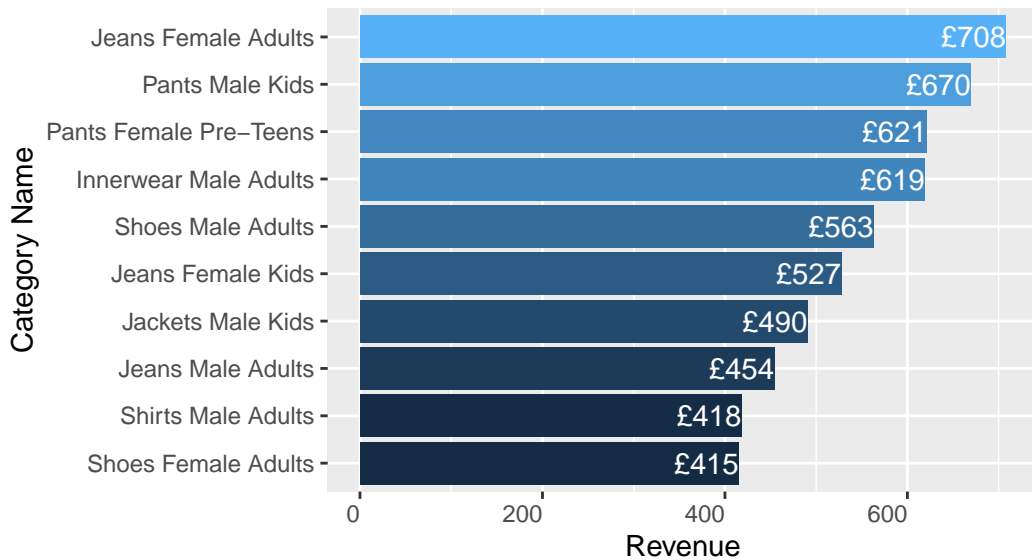
#### 10 Best Selling Products in Last 1 Year

Labels indicate the revenue in GBP generated



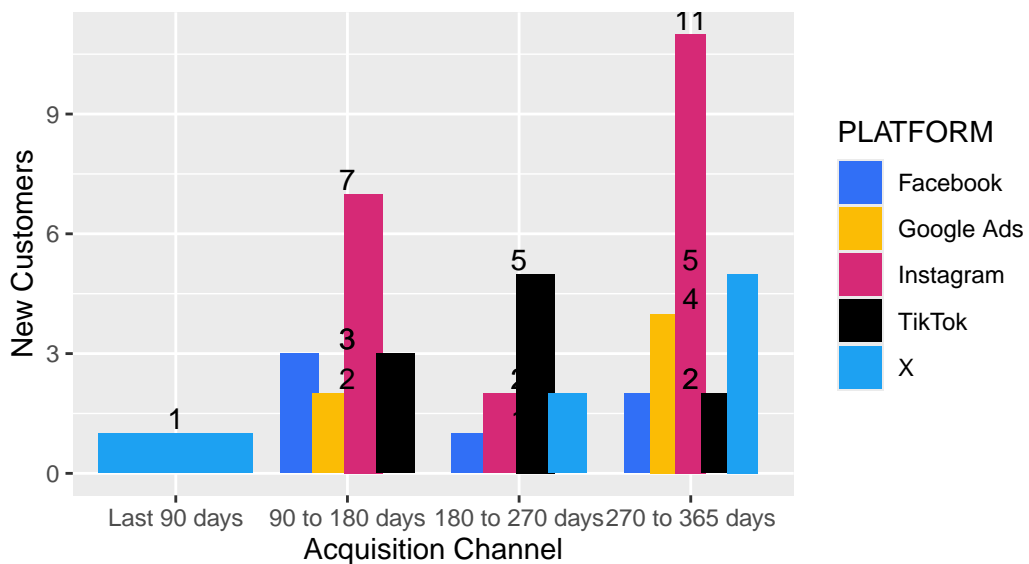
## 10 Best Selling Categories in Last 1 Year

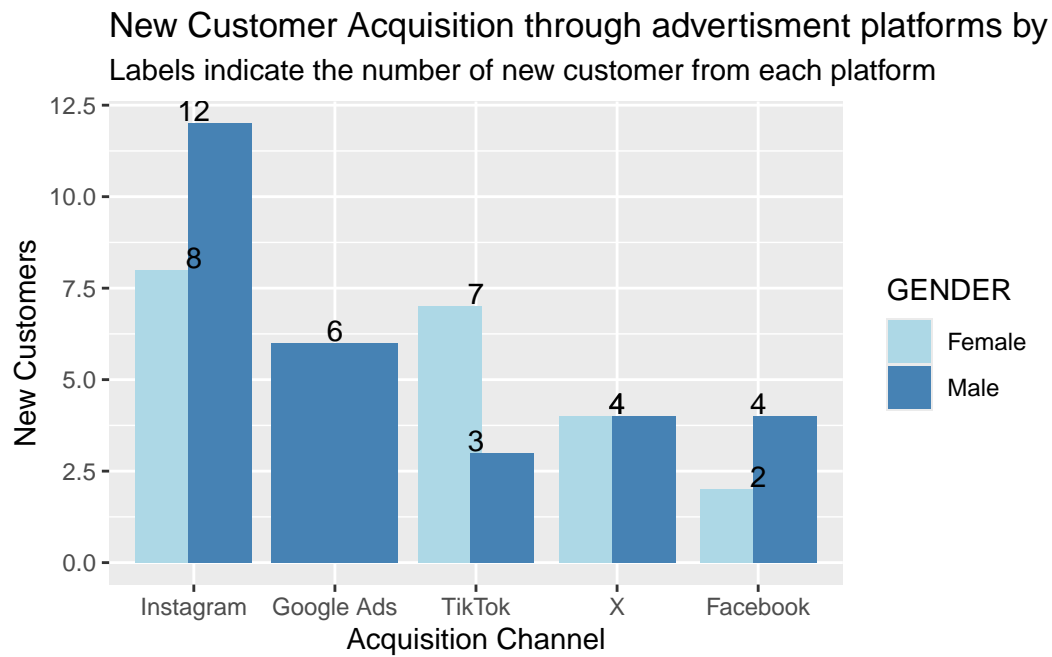
Labels indicate the revenue in GBP generating from eac



## New Customer Acquisition through advertisement platforms in La

Labels indicate the number of new customer from each platform





**Conclusion:**

**Appendix:**

**A1. Data Dictionary**