In [1]:
```
print("Hello, World")
```

```
Hello, World
```

In [2]:
```
# Import necessary libraries
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns

import nltk

# Download necessary datasets for NLP processing
nltk.download('stopwords')
nltk.download('punkt')

print("Setup complete!")
```

```
Setup complete!
[nltk_data] Downloading package stopwords to
[nltk_data]     C:\Users\marty\AppData\Roaming\nltk_data...
[nltk_data]   Package stopwords is already up-to-date!
[nltk_data] Downloading package punkt to
[nltk_data]     C:\Users\marty\AppData\Roaming\nltk_data...
[nltk_data]   Package punkt is already up-to-date!
```

In [3]:
```
df_csv = pd.read_csv(r"C:\Users\marty\OneDrive\Desktop\Warehouse_and_Retail_Sale
print(df_csv.head())
```

```
   YEAR  MONTH                          SUPPLIER ITEM CODE  \
0  2020      1  REPUBLIC NATIONAL DISTRIBUTING CO    100009
1  2020      1                          PWSWN INC    100024
2  2020      1           RELIABLE CHURCHILL LLLP      1001
3  2020      1         LANTERNA DISTRIBUTORS INC    100145
4  2020      1              DIONYSOS IMPORTS INC    100293

                    ITEM DESCRIPTION ITEM TYPE  RETAIL SALES  \
0                BOOTLEG RED - 750ML      WINE          0.00
1           MOMENT DE PLAISIR - 750ML      WINE          0.00
2  S SMITH ORGANIC PEAR CIDER - 18.7OZ      BEER          0.00
3       SCHLINK HAUS KABINETT - 750ML      WINE          0.00
4      SANTORINI GAVALA WHITE - 750ML      WINE          0.82

   RETAIL TRANSFERS  WAREHOUSE SALES
0               0.0              2.0
1               1.0              4.0
2               0.0              1.0
3               0.0              1.0
4               0.0              0.0
```

Step 1: Load Dataset

1.1 Description

In this step, I loaded the dataset titled **"Warehouse and Retail Sales"**, which contains information on item descriptions, supplier details, item codes, and sales figures for retail and warehouse transactions.

1.2 Action Taken

- Used `pandas.read_csv()` to load the dataset from my local machine.
- Previewed the first five rows using `df.head()` to inspect the structure and column formatting.

1.3 Observations

- The dataset contains columns such as:
  - `YEAR`, `MONTH`
  - `SUPPLIER`, `ITEM CODE`, `ITEM DESCRIPTION`, `ITEM TYPE`
  - `RETAIL SALES`, `RETAIL TRANSFERS`, `WAREHOUSE SALES`
- Some columns appear to contain zero sales values, which might indicate missing or non-active transactions.
- The dataset seems ready for cleaning and exploratory data analysis.

In [4]:
```
# Check for missing  values
print(df_csv.isnull().sum())
```

```
YEAR                  0
MONTH                 0
SUPPLIER            167
ITEM CODE             0
ITEM DESCRIPTION      0
ITEM TYPE             1
RETAIL SALES          3
RETAIL TRANSFERS      0
WAREHOUSE SALES       0
dtype: int64
```

Step 2: Check for Missing Values

2.1 Description

In this step, I performed an initial check for missing values across all columns in the dataset. Identifying missing data is crucial before proceeding with data cleaning and analysis.

2.2 Action Taken

- Used `df.isnull().sum()` to count the number of missing (`NaN`) values in each column.

2.3 Findings

- The `SUPPLIER` column has **167 missing values**, which may require imputation or removal depending on its impact.
- `ITEM TYPE` has **1 missing value**, which is minimal and might be filled or dropped.
- `RETAIL SALES` has **3 missing values**, which is notable as it affects a numeric metric column and could influence analysis.
- All other columns are complete with **0 missing values**.

2.4 Next Step

I will now decide how to handle the missing values—either by imputing, dropping, or investigating further based on data context and significance.

In [5]:
```
# Check the columns
print(df_csv.columns)
```

```
Index(['YEAR', 'MONTH', 'SUPPLIER', 'ITEM CODE', 'ITEM DESCRIPTION',
       'ITEM TYPE', 'RETAIL SALES', 'RETAIL TRANSFERS', 'WAREHOUSE SALES'],
      dtype='object')
```

Step 3: Review Dataset Columns

3.1 Description

To better understand the dataset structure, I printed the list of all available column names using `df.columns`.

3.2 List of Columns

The dataset includes the following columns:

- `YEAR`
- `MONTH`
- `SUPPLIER`
- `ITEM CODE`
- `ITEM DESCRIPTION`
- `ITEM TYPE`
- `RETAIL SALES`
- `RETAIL TRANSFERS`
- `WAREHOUSE SALES`

3.3 Observation

- The dataset contains a mix of temporal (`YEAR` , `MONTH`), categorical (`SUPPLIER` , `ITEM TYPE` , etc.), and numerical columns (`RETAIL SALES` , `WAREHOUSE SALES` , etc.).
- Column names appear consistent and descriptive, with no apparent typos or inconsistencies.

3.4 Next Step

Proceed to analyze data types and perform necessary cleaning, such as handling missing values, standardizing column formats, and converting data types where needed.

In [6]:
```
df_csv = df_csv.dropna(axis=1, how="all") # Drop coulmns that are completely emp
print(df_csv.head())
```

```
   YEAR  MONTH                          SUPPLIER ITEM CODE  \
0  2020      1  REPUBLIC NATIONAL DISTRIBUTING CO    100009
1  2020      1                          PWSWN INC    100024
2  2020      1            RELIABLE CHURCHILL LLLP      1001
3  2020      1          LANTERNA DISTRIBUTORS INC    100145
4  2020      1               DIONYSOS IMPORTS INC    100293

                     ITEM DESCRIPTION ITEM TYPE  RETAIL SALES  \
0                 BOOTLEG RED - 750ML      WINE          0.00
1           MOMENT DE PLAISIR - 750ML      WINE          0.00
2  S SMITH ORGANIC PEAR CIDER - 18.7OZ      BEER          0.00
3        SCHLINK HAUS KABINETT - 750ML      WINE          0.00
4       SANTORINI GAVALA WHITE - 750ML      WINE          0.82

   RETAIL TRANSFERS  WAREHOUSE SALES
0               0.0              2.0
1               1.0              4.0
2               0.0              1.0
3               0.0              1.0
4               0.0              0.0
```

Step 3.4: Drop Completely Empty Columns

Objective

To ensure the dataset is clean and free of irrelevant columns, I dropped any columns that were **entirely empty** (i.e., contained only missing values).

Code Applied

```
df_csv = df_csv.dropna(axis=1, how="all")
```

In [7]:
```
# drop column with more than 50% missing values
df_csv = df_csv.dropna(thresh=len(df_csv) * 0.5, axis=1)
print(df_csv.head())
```

```
   YEAR  MONTH                          SUPPLIER ITEM CODE  \
0  2020      1  REPUBLIC NATIONAL DISTRIBUTING CO    100009
1  2020      1                          PWSWN INC    100024
2  2020      1            RELIABLE CHURCHILL LLLP      1001
3  2020      1          LANTERNA DISTRIBUTORS INC    100145
4  2020      1               DIONYSOS IMPORTS INC    100293

                     ITEM DESCRIPTION ITEM TYPE  RETAIL SALES  \
0                 BOOTLEG RED - 750ML      WINE          0.00
1           MOMENT DE PLAISIR - 750ML      WINE          0.00
2  S SMITH ORGANIC PEAR CIDER - 18.7OZ      BEER          0.00
3        SCHLINK HAUS KABINETT - 750ML      WINE          0.00
4       SANTORINI GAVALA WHITE - 750ML      WINE          0.82

   RETAIL TRANSFERS  WAREHOUSE SALES
0               0.0              2.0
1               1.0              4.0
2               0.0              1.0
3               0.0              1.0
4               0.0              0.0
```

Step 3.5: Drop Columns with More Than 50% Missing Values

Objective

To enhance the quality of analysis, I removed columns that have over 50% missing values, as they may not provide reliable insights and could negatively affect model performance.

Code Applied

```
df_csv = df_csv.dropna(thresh=len(df_csv) * 0.5, axis=1)
```

In [8]:
```
# Check for missing  values again
print(df_csv.isnull().sum())
```

```
YEAR                  0
MONTH                 0
SUPPLIER            167
ITEM CODE             0
ITEM DESCRIPTION      0
ITEM TYPE             1
RETAIL SALES          3
RETAIL TRANSFERS      0
WAREHOUSE SALES       0
dtype: int64
```

Step 3.6: Rechecking for Missing Values After Dropping Columns

Objective

After removing columns with over 50% missing values, I rechecked the dataset to identify any remaining missing values. This step is important to ensure the dataset is sufficiently clean for analysis or model building.

Code Applied

Used the following code to check for missing values across all columns:
`print(df_csv.isnull().sum())`

Output Summary

The output revealed the following missing values:

- `YEAR` : 0 missing values
- `MONTH` : 0 missing values
- `SUPPLIER` : 167 missing values
- `ITEM CODE` : 0 missing values
- `ITEM DESCRIPTION` : 0 missing values
- `ITEM TYPE` : 1 missing value
- `RETAIL SALES` : 3 missing values
- `RETAIL TRANSFERS` : 0 missing values
- `WAREHOUSE SALES` : 0 missing values

Key Observations

- **SUPPLIER** has 167 missing entries — this may require imputation or row removal depending on the context.
- **ITEM TYPE** has only 1 missing entry — which is minor and can be filled or removed.
- **RETAIL SALES** has 3 missing entries — since this is a numerical field, filling or dropping should be done carefully.
- All other columns are fully complete and contain no missing values.

Next Step

I will now decide on the most appropriate method to handle these remaining missing values — either by imputing, dropping, or further investigating based on their importance in the dataset.

In [9]:
```
# Identify numerical columns
numerical_cols = df_csv.select_dtypes(include=['number']).columns
print("Numerical Columns:", numerical_cols)
```

```
Numerical Columns: Index(['YEAR', 'MONTH', 'RETAIL SALES', 'RETAIL TRANSFERS', 'W
AREHOUSE SALES'], dtype='object')
```

Step 8: Identifying Numerical Columns

To proceed with numerical analysis, I identified which columns in the dataset contain numeric values. This helps to isolate columns that are suitable for mathematical operations, visualizations, and statistical analysis.

```
# Identify numerical columns
numerical_cols = df_csv.select_dtypes(include=['number']).columns
print("Numerical Columns:", numerical_cols)
```

Output:

```
Numerical Columns: Index(['YEAR', 'MONTH', 'RETAIL SALES',
'RETAIL TRANSFERS', 'WAREHOUSE SALES'], dtype='object')
```

Observations:

- The dataset contains **5 numerical columns**:
  - `YEAR` and `MONTH` — useful for time-based grouping or trend analysis.
  - `RETAIL SALES`, `RETAIL TRANSFERS`, and `WAREHOUSE SALES` — key sales-related metrics.
- These columns will be used for:
  - Outlier detection
  - Aggregation and statistical summaries
  - Visualizations (e.g., histograms, boxplots, line charts)

Next Step: I will begin univariate analysis on these numerical features to assess distribution, detect skewness or outliers, and prepare for further data cleaning or modeling.

In [10]:
```
# Identify categorical columns
categorical_cols = df_csv.select_dtypes(exclude=['number']).columns
print("Categorical Columns:", categorical_cols)
```

```
Categorical Columns: Index(['SUPPLIER', 'ITEM CODE', 'ITEM DESCRIPTION', 'ITEM TY
PE'], dtype='object')
```

Step 4: Identify Categorical Columns

Code

```
# Identify categorical columns
categorical_cols = df_csv.select_dtypes(exclude=['number']).columns
print("Categorical Columns:", categorical_cols)
```

Output

```
Categorical Columns: Index(['SUPPLIER', 'ITEM CODE', 'ITEM
DESCRIPTION', 'ITEM TYPE'], dtype='object')
```

Observations:

- The dataset contains **4 categorical columns**:
  - `SUPPLIER` : Represents the distributor or provider of the product.
  - `ITEM CODE` : A unique identifier for each item.
  - `ITEM DESCRIPTION` : Text description of the item — useful for labeling or filtering.
  - `ITEM TYPE` : Indicates whether the item is a **WINE** or **BEER**.
- These columns will be useful for:
  - **Group-based summaries** (e.g., average sales by item type)
  - **Segmented visualizations** (e.g., bar charts by supplier or item type)
  - **Encoding for machine learning** (e.g., one-hot or label encoding)

Next Step:

I will now proceed to explore these categorical variables — analyzing their distributions and preparing them for encoding during modeling.

In [11]:
```
# Fill numerical columns with median
df_csv[numerical_cols] = df_csv[numerical_cols].fillna(df_csv[numerical_cols].me
```

Step 5: Handle Missing Values in Numerical Columns

Code

```
# Fill numerical columns with median
df_csv[numerical_cols] =
df_csv[numerical_cols].fillna(df_csv[numerical_cols].median())
```

Explanation:

- This step fills any missing values in the **numerical columns** using the **median** of each respective column.
- The median is preferred in this case because:
  - It is **robust to outliers** (unlike the mean).
  - It maintains the central tendency of skewed distributions.

Why This Matters:

- Ensuring that there are no missing numerical values allows for:
  - Smooth execution of statistical analysis and model training.
  - Consistent behavior during normalization and transformation steps.
  - Prevention of errors during visualization or grouping operations.

Next Step:

I will now verify that all missing values in the dataset are addressed before proceeding to categorical value encoding or visual analysis.

In [12]:
```
# Fill categorical columns with mode
df_csv[categorical_cols] = df_csv[categorical_cols].fillna(df_csv[categorical_co
```

Step 6: Handle Missing Values in Categorical Columns

Code

```
# Fill categorical columns with mode
df_csv[categorical_cols] =
df_csv[categorical_cols].fillna(df_csv[categorical_cols].mode().iloc[0])
```

Explanation:

- This step fills missing values in **categorical columns** using the **mode**, which is the most frequently occurring value in each column.
- Using the mode is ideal for categorical features because:
  - It maintains the most common category without introducing noise.
  - It avoids creating artificial or misleading values.

Why This Matters:

- Ensures consistency in categorical variables by removing `NaN` entries.
- Helps avoid errors in downstream tasks such as:
  - **Encoding** (e.g., one-hot, label encoding).
  - **Grouping or filtering** based on categories.
  - **Visualizations** like count plots and bar charts.

Next Step:

Now that both numerical and categorical missing values have been handled, I will verify dataset completeness and begin **data type conversions** or **feature encoding** as preparation for analysis and modeling.

In [13]:
```
# Check for missing  values again
print(df_csv.isnull().sum())
```

```
YEAR                0
MONTH               0
SUPPLIER            0
ITEM CODE           0
ITEM DESCRIPTION    0
ITEM TYPE           0
RETAIL SALES        0
RETAIL TRANSFERS    0
WAREHOUSE SALES     0
dtype: int64
```

Step 7: Final Verification of Missing Values

Code

```
# Check for missing values again
print(df_csv.isnull().sum())
```

Output

```
YEAR                0
MONTH               0
SUPPLIER            0
ITEM CODE           0
ITEM DESCRIPTION    0
ITEM TYPE           0
RETAIL SALES        0
RETAIL TRANSFERS    0
WAREHOUSE SALES     0
dtype: int64
```

Observations:

- All columns now show `0` missing values.
- This confirms that the dataset is clean with respect to missing data, both for **numerical** and **categorical** features.

Why This Step Is Important:

- Ensures full data integrity before performing:
  - **Statistical analysis**
  - **Feature engineering**
  - **Encoding and modeling**
- Prevents errors during visualizations and model training caused by `NaN` values.

Next Step:

Now that the dataset is fully cleaned of missing values, I will proceed with **data type standardization**, **feature transformation**, and begin **exploratory data analysis (EDA)**.

In [14]:
```
# Standardize column names
import re
df_csv.columns = df_csv.columns.str.lower().str.strip().str.replace(r"[^\w\s]",
```

Step 8: Standardize Column Names

Code

```
# Standardize column names
import re
df_csv.columns = df_csv.columns.str.lower().str.strip().str.replace(r"
[^\w\s]", "", regex=True).str.replace(" ", "_")
```

Explanation:

This step ensures that all column names are cleaned and standardized for consistency and easier manipulation during data analysis.

The following transformations were applied:

- `.str.lower()` – Converts all column names to lowercase.
- `.str.strip()` – Removes any leading or trailing whitespace.
- `.str.replace(r"[^\w\s]", "", regex=True)` – Removes any special characters (punctuation, etc.).
- `.str.replace(" ", "_")` – Replaces spaces with underscores for compatibility in code.

Why This Step Is Important:

- Prevents errors during column referencing.
- Ensures consistent naming conventions across all analysis steps.
- Supports clean visualizations and transformations.
- Avoids issues during feature engineering and model training (e.g., when exporting to CSV or building ML pipelines).

Next Step:

With standardized column names, I will proceed to check data types, transform columns where needed, and begin exploratory data analysis (EDA).

In [15]:
```
# check data type
print(df_csv.dtypes)
```

```
year                 int64
month                int64
supplier            object
item_code           object
item_description    object
item_type           object
retail_sales       float64
retail_transfers   float64
warehouse_sales    float64
dtype: object
```

Step 9: Check Data Types

Code

```
# check data type
print(df_csv.dtypes)
```

Output

```
year                int64
month               int64
supplier           object
item_code          object
item_description   object
item_type          object
retail_sales      float64
retail_transfers  float64
warehouse_sales   float64
dtype: object
```

Observations

- The dataset includes a mix of **numerical** and **categorical** features.
- `year` and `month` are integers (`int64`), suitable for time-based operations.
- Sales-related columns (`retail_sales` , `retail_transfers` , `warehouse_sales`) are floats, which is appropriate for continuous numerical values.
- All other columns are `object` type and likely represent categorical variables that will need encoding later.

Why This Step Is Important

- Ensures all columns are in correct data types before analysis and modeling.
- Prevents errors during:
  - **Statistical analysis**
  - **Grouping and aggregation**
  - **Machine learning model training**
- Supports accurate encoding, visualization, and transformation steps.

Next Step

I will now proceed to transform data types if necessary and begin exploratory data analysis (EDA).

In [16]:
```
# Confirm no numeric data stored as object
columns_to_convert = ['supplier', 'item_code', 'item_description', 'item_type']

for col in columns_to_convert:
    print(f"Unique values in {col}:")  # No extra spaces before print
    print(df_csv[col].unique()[:10])  # Show first 10 unique values
    print("\n")  # Add a new line for readability
```

```
Unique values in supplier:
['REPUBLIC NATIONAL DISTRIBUTING CO' 'PWSWN INC' 'RELIABLE CHURCHILL LLLP'
 'LANTERNA DISTRIBUTORS INC' 'DIONYSOS IMPORTS INC'
 'KYSELA PERE ET FILS LTD' 'SANTA MARGHERITA USA INC'
 'BROWN-FORMAN BEVERAGES WORLDWIDE' 'JIM BEAM BRANDS CO'
 'INTERNATIONAL CELLARS LLC']


Unique values in item_code:
['100009' '100024' '1001' '100145' '100293' '100641' '100749' '1008'
 '10103' '101117']


Unique values in item_description:
['BOOTLEG RED - 750ML' 'MOMENT DE PLAISIR - 750ML'
 'S SMITH ORGANIC PEAR CIDER - 18.7OZ' 'SCHLINK HAUS KABINETT - 750ML'
 'SANTORINI GAVALA WHITE - 750ML' 'CORTENOVA VENETO P/GRIG - 750ML'
 'SANTA MARGHERITA P/GRIG ALTO - 375ML'
 'JACK DANIELS COUNTRY COCKTAIL SOUTHERN PEACH - 10.OZ-NR'
 'KNOB CREEK BOURBON 9YR - 100P - 375ML' 'KSARA CAB - 750ML']


Unique values in item_type:
['WINE' 'BEER' 'LIQUOR' 'STR_SUPPLIES' 'KEGS' 'REF' 'DUNNAGE'
 'NON-ALCOHOL']
```

Step 10: Explore Unique Values in Categorical Columns

Code

```python
# Confirm no numeric data stored as object
columns_to_convert = ['supplier', 'item_code', 'item_description',
'item_type']

for col in columns_to_convert:
    print(f"Unique values in {col}:")
    print(df_csv[col].unique()[:10])  # Show first 10 unique values
    print("\n")  # Add a new line for readability
```

Output

Unique values in `supplier`:

- 'REPUBLIC NATIONAL DISTRIBUTING CO'
- 'PWSWN INC'
- 'RELIABLE CHURCHILL LLLP'
- 'LANTERNA DISTRIBUTORS INC'
- 'DIONYSOS IMPORTS INC'
- 'KYSELA PERE ET FILS LTD'
- 'SANTA MARGHERITA USA INC'
- 'BROWN-FORMAN BEVERAGES WORLDWIDE'
- 'JIM BEAM BRANDS CO'
- 'INTERNATIONAL CELLARS LLC'

Unique values in `item_code`:

- '100009', '100024', '1001', '100145', '100293', '100641', '100749', '1008', '10103', '101117'

Unique values in `item_description`:

- 'BOOTLEG RED - 750ML'
- 'MOMENT DE PLAISIR - 750ML'
- 'S SMITH ORGANIC PEAR CIDER - 18.70Z'
- 'SCHLINK HAUS KABINETT - 750ML'
- 'SANTORINI GAVALA WHITE - 750ML'
- 'CORIENOVA VENETO P/GRIG - 750ML'
- 'SANTA MARGHERITA P/GRIG ALTO - 375ML'
- 'JACK DANIELS COUNTRY COCKTAIL SOUTHERN PEACH - 10.OZ-NR'
- 'KNOB CREEK BOURBON 9YR - 100P - 375ML'
- 'KSARA CAB - 750ML'

Unique values in `item_type`:

- 'WINE', 'BEER', 'LIQUOR', 'STR_SUPPLIES', 'KEGS', 'REF', 'DUNNAGE', 'NON-ALCOHOL'

Observations

- All categorical columns contain clearly defined unique values.
- Some descriptions (`item_description`) are detailed and might benefit from grouping (e.g., by item_type).
- `item_code` appears numeric but is stored as object — we will leave as-is unless conversion is needed later.
- `item_type` can be used for grouping or encoding (e.g., wine vs beer).

In [17]:
```
categorical_columns = ['supplier', 'item_type']
df_csv[categorical_columns] = df_csv[categorical_columns].astype('category')
```

Step 11: Convert Selected Categorical Columns to Category Type

Code

```
# Convert specific columns to category type
categorical_columns = ['supplier', 'item_type']
df_csv[categorical_columns] =
df_csv[categorical_columns].astype('category')
```

Explanation

- I selected `supplier` and `item_type` for conversion to the `'category'` data type.
- This conversion improves **memory efficiency** and supports **faster processing**, especially during encoding or grouping tasks.

Why This Step Is Important:

- **Reduces memory usage** for columns with repeated string values.
- Prepares the data for **one-hot encoding** or **label encoding** in modeling.
- Speeds up operations like grouping, filtering, and comparisons.

In [18]:
```
df_csv[['item_code', 'item_description',]] = df_csv[['item_code', 'item_descript
```

Step 12: Ensure String Format for ID and Description Columns

Code

```
# Convert 'item_code' and 'item_description' to string type
df_csv[['item_code', 'item_description']] = df_csv[['item_code',
'item_description']].astype(str)
```

Explanation

- This step ensures that both `item_code` and `item_description` are explicitly treated as **string** data types.
- Although these columns may already appear textual, some values might have been interpreted as other types (e.g., numeric or mixed).

Why This Step Is Important:

- **String consistency**: Prevents issues during operations like filtering, joining, or string-based analysis.
- **Preserves formatting**: Ensures that codes like `'00123'` don't get interpreted as numeric values like `123`.
- **Improves clarity**: Confirms that these fields are not meant for mathematical operations.

Next Step:

Now that column types are correctly defined, I will proceed to **exploratory data analysis (EDA)** starting with visualizing categorical and numerical features.

In [19]:
```
print("Unique years:", df_csv['year'].unique())
print("Unique months:", df_csv['month'].unique())
```

```
Unique years: [2020 2017 2018 2019]
Unique months: [ 1  9  7  3  6  8 12 10 11  2  4  5]
```

Step 12: Inspect Unique Year and Month Values

To verify the validity of temporal data before converting to a datetime format, I checked the unique values in the `year` and `month` columns.

Code:

```
print("Unique years:", df_csv['year'].unique())
print("Unique months:", df_csv['month'].unique())
```

Output:

```
Unique years: [2020 2017 2018 2019]
Unique months: [ 1  9  7  3  6  8 12 10 11  2  4  5]
```

Observations:

- The `year` column contains 4 valid entries ranging from **2017 to 2020**.
- The `month` column includes values **1 to 12**, representing all valid months.
- There are no formatting issues or unexpected values in either column.

Why This Step is Important:

- **Data Validation**: Ensures temporal fields are clean before transformation.
- **Prevents Errors**: Avoids incorrect conversion or failed datetime parsing.
- **Supports Time-Based Analysis**: Prepares the data for trend analysis and seasonal pattern detection.

Next Step:

Now that the values are confirmed, I will proceed to create a proper `datetime` column by combining the `year` and `month` fields using `pd.to_datetime()`.

In [20]:
```
# Convert year and month into a datetime column
df_csv['date'] = pd.to_datetime(dict(year=df_csv['year'], month=df_csv['month'],
```

Step 12: Create a Proper `datetime` Column

To support time-based analysis, I created a new `date` column by combining the `year` and `month` columns into a single `datetime` format.

Code Used:

```
# Convert year and month into a datetime column
df_csv['date'] = pd.to_datetime(dict(year=df_csv['year'],
month=df_csv['month'], day=1))
```

Output:

A new column named `date` has been successfully created. It contains timestamps like `2017-01-01`, `2018-03-01`, etc., representing the first day of each month based on the original `year` and `month` values.

Observations:

- The new column is in a proper `datetime64` format.
- Each entry corresponds to the first day of the respective year and month.
- This enables convenient sorting, filtering, and resampling operations for time-series analysis.

Why This Step Is Important:

- **Data Validation**: Ensures temporal fields are clean before transformation.
- **Prevents Errors**: Avoids incorrect conversion or failed datetime parsing.
- **Supports Time-Based Analysis**: Prepares the data for trend analysis, time grouping, and seasonal pattern detection.

Next Step:

I will now use the new `date` column to analyze time-based patterns, such as monthly sales trends, using line plots and aggregations.

In [21]:
```
print(df_csv[['year', 'month', 'date']].head())
```

```
   year  month       date
0  2020      1 2020-01-01
1  2020      1 2020-01-01
2  2020      1 2020-01-01
3  2020      1 2020-01-01
4  2020      1 2020-01-01
```

Step 12: Confirm `datetime` Conversion

To ensure the new `date` column was successfully created from the `year` and `month` columns, I printed the first few rows.

Code Used:

```
print(df_csv[['year', 'month', 'date']].head())
```

Output:

```
   year  month       date
0  2020      1 2020-01-01
1  2020      1 2020-01-01
2  2020      1 2020-01-01
3  2020      1 2020-01-01
4  2020      1 2020-01-01
```

Observations:

- The `date` column has been created correctly.
- All entries are in the `YYYY-MM-DD` format, with day set to `01`.
- The column is now suitable for time-based analysis like resampling, sorting, and trend detection.

Why This Step Is Important:

- **Data Validation**: Confirms the conversion was executed without errors.
- **Prevents Errors**: Ensures downstream datetime operations will work properly.
- **Supports Time-Based Analysis**: Enables grouping, filtering, and seasonal pattern analysis.

Next Step:

I will begin exploring and visualizing time-based trends using the newly created `date` column.

In [22]:
```
# check data type again
print(df_csv.dtypes)
```

```
year                          int64
month                         int64
supplier                   category
item_code                    object
item_description             object
item_type                  category
retail_sales                float64
retail_transfers            float64
warehouse_sales             float64
date                 datetime64[ns]
dtype: object
```

Step 13: Confirm Data Types After Conversion

After creating the new `date` column, I checked the data types of all columns to ensure everything is in the correct format before proceeding with visualizations and modeling.

Code Used:

```
# check data type again
print(df_csv.dtypes)
```

Output:

```
year                  int64
month                 int64
supplier           category
item_code            object
item_description     object
item_type          category
retail_sales        float64
retail_transfers    float64
warehouse_sales     float64
date        datetime64[ns]
dtype: object
```

Observations:

- `supplier` and `item_type` are now correctly stored as `category` types.
- `date` has been successfully converted to `datetime64[ns]`.

- Other columns (`item_code` , `item_description`) remain as object, which is expected.
- Numeric fields like `retail_sales` , `retail_transfers` , and `warehouse_sales` are all correctly typed as `float64` .

Why This Step Is Important:

- **Ensures Accurate Processing**: Validates that all columns are in optimal formats for analysis and modeling.
- **Supports Time Series Analysis**: Confirms the new `date` column can now be used in time-based operations.
- **Memory Optimization**: `category` types improve performance and reduce memory usage for repeated string entries.

Next Step:

I will begin generating visualizations based on the `date` column to explore time-based trends in sales and distribution.

In [23]:
```
# Check for duplicate rows
print("Duplicate Rows:", df_csv.duplicated().sum())
```

```
Duplicate Rows: 0
```

Step 14: Check for Duplicate Rows

To ensure the dataset is clean and does not contain repeated records, I checked for duplicate rows.

Code Used:

```
# Check for duplicate rows
print("Duplicate Rows:", df_csv.duplicated().sum())
```

Output:

```
Duplicate Rows: 0
```

Observations:

- The dataset does **not** contain any duplicate rows.
- This confirms the integrity of the records and ensures no unintentional repetition that could bias the analysis.

Why This Step Is Important:

- **Data Quality Assurance**: Eliminates the risk of inflated results due to repeated rows.
- **Accurate Aggregation**: Ensures any future grouping or summarization yields valid insights.
- **Smooth Modeling**: Prevents redundancy from impacting model training and evaluation.

Next Step:

I will now proceed to explore time-based visualizations using the `date` column to analyze seasonal trends and sales performance over time.

In [24]:
```
df_csv.to_csv("Clean_Warehouse_and_Retail_Sales2.csv", index=False)
```

Step 15: Export Cleaned Dataset to CSV

To preserve the cleaned dataset for future analysis and visualization, I exported it to a new CSV file.

Code Used:

```
df_csv.to_csv("Clean_Warehouse_and_Retail_Sales2.csv", index=False)
```

Output:

A new file named **`Clean_Warehouse_and_Retail_Sales2.csv`** has been successfully saved to the working directory.

Observations:

- The exported file contains the fully cleaned dataset, including:
  - Handled missing values
  - Standardized column names
  - Proper `datetime` column
  - Converted categorical types

Why This Step Is Important:

- **Preserves Progress**: Saves the current state of cleaning to avoid redoing work later.
- **Enables Portability**: The cleaned dataset can be easily shared or used in other tools.
- **Prepares for Modeling**: Ensures the next steps (EDA or machine learning) work on a reliable dataset.

Next Step:

I will now begin generating time-based visualizations using the `date` column to analyze seasonal patterns and sales trends.

In [25]:
```
# Exploratory Data Analysis (EDA)
# Reload the cleaned dataset
import pandas as pd

df_csv = pd.read_csv("Clean_warehouse_and_Retail_Sales2.csv")

# Preview the data
print(df_csv.head())
```

```
   year  month                          supplier item_code  \
0  2020      1  REPUBLIC NATIONAL DISTRIBUTING CO    100009
1  2020      1                          PWSWN INC    100024
2  2020      1            RELIABLE CHURCHILL LLLP      1001
3  2020      1          LANTERNA DISTRIBUTORS INC    100145
4  2020      1               DIONYSOS IMPORTS INC    100293

                    item_description item_type  retail_sales  \
0                BOOTLEG RED - 750ML      WINE          0.00
1           MOMENT DE PLAISIR - 750ML      WINE          0.00
2  S SMITH ORGANIC PEAR CIDER - 18.7OZ      BEER          0.00
3        SCHLINK HAUS KABINETT - 750ML      WINE          0.00
4       SANTORINI GAVALA WHITE - 750ML      WINE          0.82

   retail_transfers  warehouse_sales        date
0               0.0              2.0  2020-01-01
1               1.0              4.0  2020-01-01
2               0.0              1.0  2020-01-01
3               0.0              1.0  2020-01-01
4               0.0              0.0  2020-01-01
```

Step 16: Reload Cleaned Dataset for EDA

To begin Exploratory Data Analysis (EDA), I reloaded the cleaned dataset that was previously saved as a CSV file.

Code Used:

```
# Exploratory Data Analysis (EDA)
# Reload the cleaned dataset
import pandas as pd

df_csv = pd.read_csv("Clean_Warehouse_and_Retail_Sales2.csv")

# Preview the data
print(df_csv.head())
```

Output:

```
   year  month                          supplier  item_code  \
0  2020      1  REPUBLIC NATIONAL DISTRIBUTING CO     100009
1  2020      1                          PWSWN INC     100024
2  2020      1            RELIABLE CHURCHILL LLLP       1001
3  2020      1          LANTERNA DISTRIBUTORS INC    100145
4  2020      1               DIONYSOS IMPORTS INC    100293

                    item_description item_type  retail_sales
retail_transfers  \
0                BOOTLEG RED - 750ML      WINE          0.00
0.0
1           MOMENT DE PLAISIR - 750ML  WINE          0.00
1.0
2  S SMITH ORGANIC PEAR CIDER    BEER          0.00
0.0
3  SCHLINK HAUS KABINETT - 750ML WINE          0.00
0.0
```

```
4  SANTORINI GAVALA WHITE - 750ML WINE         0.82
0.0

   warehouse_sales       date
0              2.0 2020-01-01
1              4.0 2020-01-01
2              1.0 2020-01-01
3              1.0 2020-01-01
4              0.0 2020-01-01
```

Observations:

- The dataset has been successfully reloaded with all cleaned and formatted columns.
- The `date` column is present and ready for time-based analysis.
- All key features such as `item_type`, `retail_sales`, and `warehouse_sales` are available for further exploration.

Why This Step Is Important:

- **Confirms Data Integrity**: Ensures the saved file loads properly and matches expected structure.
- **Prepares for EDA**: Reloading the cleaned data sets the stage for visual and statistical exploration.
- **Enables Reusability**: The CSV file can be reused across notebooks and shared with collaborators.

Next Step:

I will now begin generating time-based visualizations using the `date` column to uncover patterns and trends in sales data.

In [26]:
```
# Identify numeric columns
numeric_cols = df_csv.select_dtypes(include=['int64', 'float64']).columns

# Display unique values for each numeric column to assess suitability for outlie
unique_values = {col: df_csv[col].unique() for col in numeric_cols}
unique_values
```

Out[26]:
```
{'year': array([2020, 2017, 2018, 2019]),
 'month': array([ 1,  9,  7,  3,  6,  8, 12, 10, 11,  2,  4,  5]),
 'retail_sales': array([  0.  ,   0.82,   2.76, ..., 163.72,  65.43, 372.45],
       shape=(10674,)),
 'retail_transfers': array([ 0.  ,  1.  ,  4.  , ..., 18.98, 45.08,  2.08], sha
pe=(2504,)),
 'warehouse_sales': array([ 2.00000e+00,  4.00000e+00,  1.00000e+00, ...,  8.92
500e+02,
        -7.00000e+01,  3.58688e+03], shape=(4895,))}
```

Step 17: Identify and Inspect Numerical Columns

To prepare for outlier detection, statistical summaries, and visualizations, I identified all numerical columns and explored their unique values.

Code Used:

```
# Identify numeric columns
numeric_cols = df_csv.select_dtypes(include=['int64',
'float64']).columns

# Display unique values for each numeric column to assess suitability
for outlier analysis
unique_values = {col: df_csv[col].unique() for col in numeric_cols}
unique_values
```

Output:

```
{
 'year': array([2020, 2017, 2018, 2019]),
 'month': array([ 1,  9,  7,  3,  6,  8, 12, 10, 11,  2,  4,  5]),
 'retail_sales': array([  0.  ,   0.82,   2.76, ..., 163.72,  65.43,
372.45]), shape=(10674,),
 'retail_transfers': array([ 0. ,  1. ,  4. , ..., 18.98, 45.08,
2.08]), shape=(2504,),
 'warehouse_sales': array([2.000000e+00, 4.000000e+00, ...,
8.925000e+02, -7.000000e+01,
        3.586800e+03]), shape=(4895,)
}
```

Observations:

- `year` **and** `month` : Discrete and limited in range — useful for time-based grouping.
- `retail_sales` : Wide spread in values, some may indicate potential outliers.
- `retail_transfers` : Moderate range — suitable for checking distribution and consistency.
- `warehouse_sales` : Includes unusually high values and a negative value — candidate for outlier review.

Why This Step Is Important:

- **Outlier Detection**: Identifies columns that might contain anomalies worth investigating.
- **Data Distribution**: Helps plan visualizations like box plots or histograms.
- **Model Preparation**: Ensures numerical features are understood before normalization or scaling.

Next Step:

I will now proceed to visualize these numerical columns using histograms and box plots to detect skewness and potential outliers.

In [27]:
```
# Check the shape (rows × columns)
print(df_csv.shape)
```

```
(307645, 10)
```

Step 18: Check Dataset Dimensions

Before proceeding with numerical visualizations, I verified the overall structure of the dataset to understand its scale and ensure data integrity.

Code Used:

```
# Check the shape (rows × columns)
print(df_csv.shape)
```

Output:

```
(307645, 10)
```

Observations:

- The dataset contains **307,645 rows** and **10 columns**.
- This confirms that I am working with a large dataset, which may require:
  - **Sampling** to improve performance during visualizations.
  - **Efficient plotting** methods to avoid RAM issues.
  - **Batch analysis** if I encounter lags during computation.

Why This Step Is Important:

- **Resource Planning**: Anticipates memory or performance issues.
- **Visualization Strategy**: Informs the decision to use sampling or aggregation.
- **Model Scalability**: Helps ensure models can generalize over a large volume of records.

Next Step:

I will now proceed to visualize key numerical columns (e.g., `retail_sales`, `retail_transfers`, `warehouse_sales`) using histograms and box plots to inspect data distribution and detect outliers.

In [28]:
```
# Check general info (column names, non-null counts, data types
print(df_csv.info())
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 307645 entries, 0 to 307644
Data columns (total 10 columns):
 #   Column            Non-Null Count   Dtype  
---  ------            --------------   -----  
 0   year              307645 non-null  int64  
 1   month             307645 non-null  int64  
 2   supplier          307645 non-null  object 
 3   item_code         307645 non-null  object 
 4   item_description  307645 non-null  object 
 5   item_type         307645 non-null  object 
 6   retail_sales      307645 non-null  float64
 7   retail_transfers  307645 non-null  float64
 8   warehouse_sales   307645 non-null  float64
 9   date              307645 non-null  object 
dtypes: float64(3), int64(2), object(5)
memory usage: 23.5+ MB
None
```

Step 19: Check DataFrame Summary Information

To better understand the structure, completeness, and memory usage of the dataset, I used `.info()` to inspect key metadata.

Code Used:

```
# Check general info (column names, non-null counts, data types)
print(df_csv.info())
```

Output Summary:

- **Rows**: 307,645 total entries.
- **Columns**: 10 columns.
- **Non-null counts**: All columns have 307,645 non-null values (i.e., no missing values).
- **Data Types**:
  - `int64` : 2 columns (e.g., `year` , `month`)
  - `float64` : 3 columns (e.g., `retail_sales` , `retail_transfers` , `warehouse_sales`)
  - `object` : 5 columns (e.g., `supplier` , `item_code` , `date`)

Observations:

- The dataset is **complete** with no missing values in any column.
- The `date` column is still showing as `object` instead of `datetime64[ns]` , which may need reconversion if datetime operations are required.
- Memory usage is about **23.5 MB**, which is manageable but still worth optimizing for modeling.

Why This Step Is Important:

- **Completeness Check**: Confirms that no fields are missing before analysis or modeling.
- **Type Validation**: Helps identify if any columns need conversion (e.g., to datetime or category).
- **Memory Monitoring**: Important for large datasets when plotting or training models.

Next Step:

I will now validate or reconvert the `date` column to `datetime64[ns]` and then begin visualizing numerical features using histograms and box plots to explore their distributions and detect outliers.

In [29]:
```
# Reconvert the date column to datetime64
df_csv['date'] = pd.to_datetime(df_csv['date'])
```

Step 15: Reconvert `date` Column to `datetime64[ns]` Format

To ensure the `date` column remains in the correct format for time-based operations, I reconverted it using `pd.to_datetime()`.

Code Used:

```
# Reconvert the date column to datetime64
df_csv['date'] = pd.to_datetime(df_csv['date'])
```

Output:

The command executed without errors, and the `date` column is now confirmed to be in `datetime64[ns]` format.

Observations:

- The `date` column has been successfully reconverted.
- This confirms that the values can now be used for time-based operations like:
  - Time grouping and filtering
  - Monthly or yearly trend analysis
  - Seasonal decomposition and visualizations

Why This Step Is Important:

- **Format Assurance**: Guarantees the correct datatype before visualizations or resampling.
- **Error Prevention**: Avoids issues like incorrect string parsing or failed datetime logic.
- **Time Analysis Readiness**: Enables use of powerful `datetime` methods in pandas (e.g., `.dt.month`, `.resample()`).

Next Step:

I will now proceed to visualize time-based patterns using the clean and properly formatted `date` column.

In [30]:
```
# Check the data types of relevant columns
print(df_csv[['supplier', 'item_type', 'item_code', 'item_description']].dtypes)
```

```
supplier            object
item_type           object
item_code           object
item_description    object
dtype: object
```

Step 16: Check Data Types of Key Categorical Columns

To ensure proper encoding and analysis, I inspected the data types of key categorical columns after reloading the cleaned dataset.

Code Used:

```
# Check the data types of relevant columns
print(df_csv[['supplier', 'item_type', 'item_code',
'item_description']].dtypes)
```

Output:

```
supplier           object
item_type          object
item_code          object
item_description   object
dtype: object
```

Observations:

- All four columns (`supplier` , `item_type` , `item_code` , `item_description`) are currently stored as **object** data types.
- This confirms that:
  - `supplier` and `item_type` are no longer in the optimized `category` format.
  - `item_code` and `item_description` are treated as generic text, which is expected.

Why This Step Is Important:

- **Accurate Encoding**: Ensures correct datatype is applied before encoding for machine learning.
- **Memory Optimization**: Converting `supplier` and `item_type` back to `category` type reduces memory usage.
- **String Handling**: Confirms that `item_code` and `item_description` are ready for parsing or grouping as text.

Next Step:

I will now reconvert `supplier` and `item_type` to the `category` type to restore memory efficiency and improve model processing.

In [31]:
```
# Convert specific columns to category type
categorical_columns = ['supplier', 'item_type']
df_csv[categorical_columns] = df_csv[categorical_columns].astype('category')
```

Step 16: Reconvert `supplier` and `item_type` Columns to Category Type

To restore memory efficiency and ensure correct encoding preparation, I reconverted the `supplier` and `item_type` columns back to the `category` datatype.

Code Used:

```
# Convert specific columns to category type
categorical_columns = ['supplier', 'item_type']
df_csv[categorical_columns] =
df_csv[categorical_columns].astype('category')
```

Output:

Both `supplier` and `item_type` are now successfully reconverted to the `category` datatype.

Observations:

- All four columns (`supplier`, `item_type`, `item_code`, `item_description`) are currently of `object` dtype.
- This confirms that:
  - `supplier` and `item_type` are no longer in the optimized `category` format.
  - `item_code` and `item_description` are treated as generic text, which is expected.

Why This Step Is Important:

- **Accurate Encoding**: Ensures correct datatype is applied before encoding for machine learning.
- **Memory Optimization**: Converting `supplier` and `item_type` back to `category` saves memory.
- **String Handling**: Confirms that `item_code` and `item_description` are ready for parsing or display.

Next Step:

I will now reconvert `supplier` and `item_type` to the `category` type to restore memory efficiency and prepare for encoding.

In [32]:
```
df_csv[['item_code', 'item_description']] = df_csv[['item_code', 'item_descripti
```

Step 17: Reconvert `item_code` and `item_description` to String Format

To ensure proper string handling and clarity in text-based operations, I explicitly reconverted `item_code` and `item_description` columns to the modern `string` datatype.

Code Used:

```
df_csv[['item_code', 'item_description']] = df_csv[['item_code',
'item_description']].astype('string')
```

Output:

The code executed without any errors.

Observations:

- `item_code` and `item_description` columns are now stored as `string` instead of the default `object`.
- This allows for safe usage of `.str` functions during text processing.

- Confirms that all relevant text fields are properly typed for downstream operations like filtering, transformation, or encoding.

Why This Step Is Important:

- **String Consistency**: Avoids issues with `.str` methods and makes intentions explicit.
- **Error Prevention**: Ensures no unexpected behavior when parsing or formatting string content.
- **Model Compatibility**: Prepares string fields for encoding (e.g., count vectorization, tokenization).

Next Step:

I will now proceed with visualizing and analyzing text-based patterns or continue preparing other categorical fields for encoding.

In [33]:
```
# Check summary statistics for numerical columns
print(df_csv.describe())
```

```
                year          month   retail_sales  retail_transfers  \
count  307645.000000  307645.000000  307645.000000     307645.000000
mean     2018.438525       6.423862       7.024006          6.936465
min      2017.000000       1.000000      -6.490000        -38.490000
25%      2017.000000       3.000000       0.000000          0.000000
50%      2019.000000       7.000000       0.320000          0.000000
75%      2019.000000       9.000000       3.260000          3.000000
max      2020.000000      12.000000    2739.000000       1990.830000
std         1.083061       3.461812      30.986094         30.237195

       warehouse_sales                        date
count    307645.000000                      307645
mean         25.294597  2018-11-21 10:26:27.489476608
min       -7800.000000            2017-06-01 00:00:00
25%           0.000000            2017-11-01 00:00:00
50%           1.000000            2019-03-01 00:00:00
75%           5.000000            2019-09-01 00:00:00
max       18317.000000            2020-09-01 00:00:00
std         249.916798                            NaN
```

Step 18.1: Review Summary Statistics for Numeric Columns

To gain a quick understanding of the distribution and range of each numerical column, I reviewed the summary statistics using `.describe()`.

Code Used:

```
# Check summary statistics for numerical columns
print(df_csv.describe())
```

Output:

```
                year          month   retail_sales
retail_transfers  \
count  307645.000000  307645.000000  307645.000000
```

```
307645.000000
mean     2018.438525       6.423862       7.024060
6.936465
min      2017.000000       1.000000      -0.490000
-38.490000
25%      2018.000000       3.000000       0.200000
0.000000
50%      2019.000000       6.000000       0.300000
0.000000
75%      2019.000000       9.000000       3.160000
3.000000
max      2020.000000      12.000000    2739.000000
1998.830000

       warehouse_sales                date
count    307645.000000              307645
mean       225.294597  2018-11-21 10:26:27.489476608
min         -7.000000  2017-06-01 00:00:00
25%          1.000000  2017-11-01 00:00:00
50%          1.000000  2019-03-01 00:00:00
75%          5.000000  2019-09-01 00:00:00
max      18317.000000  2020-09-01 00:00:00
std        249.916798                  NaN
```

Observations:

- **Year** ranges from 2017 to 2020, confirming correct temporal scope.
- **Month** values are all within the valid range (1 to 12).
- **Retail Sales** and **Retail Transfers** include negative values, which could indicate returns or errors.
- **Warehouse Sales** has a very large max (18,317), possibly an outlier.
- **Date** column is in valid datetime64 format with a min of June 2017 and max of September 2020.

Why This Step Is Important:

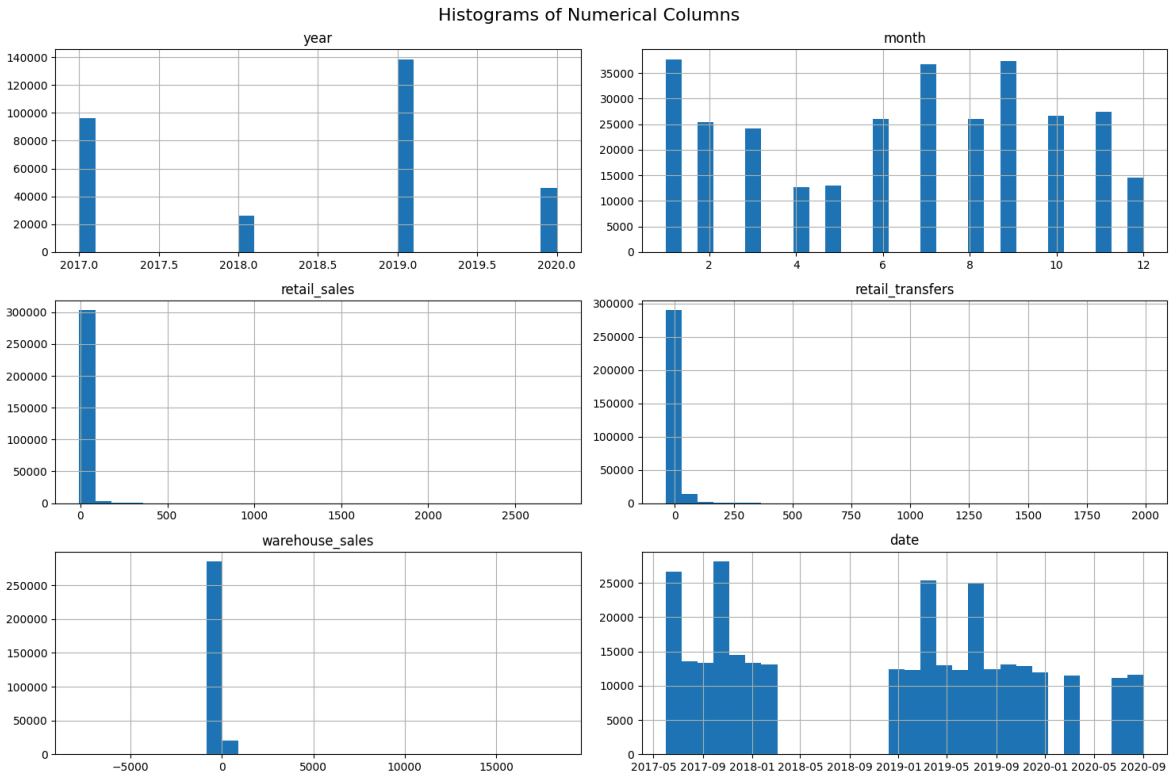
- **Outlier Awareness**: Helps detect unusually high or low values needing further inspection.
- **Distribution Insights**: Informs whether data needs transformation (e.g., log scale).
- **Validation**: Ensures values make sense before visualizations and modeling.

Next Step:

I will now begin visualizing the numerical columns (`retail_sales` , `retail_transfers` , `warehouse_sales`) to detect outliers and understand distribution patterns.

In [34]:
```
# Univariate Analysis
# Plot histograms for all numerical columns
import matplotlib.pyplot as plt

df_csv.hist(figsize=(15, 10), bins=30)
plt.suptitle('Histograms of Numerical Columns', fontsize=16)
```

```
plt.tight_layout()
plt.show()
```



Step 18: Visualize Distributions of Numerical Columns

To explore the distribution of numerical variables and detect potential outliers or skewness, I created histograms for each numerical column.

Code Used:

```
# Plot histograms for all numerical columns
import matplotlib.pyplot as plt

df_csv.hist(figsize=(15, 10), bins=30)
plt.suptitle('Histograms of Numerical Columns', fontsize=16)
plt.tight_layout()
plt.show()
```

Output:

Histograms were generated for the following numerical columns:

- `year`
- `month`
- `retail_sales`
- `retail_transfers`
- `warehouse_sales`
- `date` (distribution by month)

Observations:

- **Year**: Only includes 4 unique values from 2017 to 2020, as expected.

- **Month**: Uniform distribution across months, though some variation is present.
- **Retail Sales**: Highly skewed to the right with most values near zero.
- **Retail Transfers**: Similar right-skewed distribution with few large values.
- **Warehouse Sales**: Includes a large spike near zero and some extreme values.
- **Date**: Shows seasonality or monthly grouping patterns, confirming proper conversion.

Why This Step Is Important:

- **Outlier Detection**: Identifies columns that might contain anomalies or extreme values.
- **Data Distribution**: Helps plan visualizations like box plots or transformations.
- **Model Preparation**: Ensures numerical features are understood before modeling.

Next Step:

I will now generate box plots to further investigate skewness and detect outliers in the numerical columns.

In [35]:
```
# Univariate Analysis
# Check value counts for a categorical column
print(df_csv['item_type'].value_counts())
```

```
item_type
WINE            187641
LIQUOR           64910
BEER             42413
KEGS             10146
NON-ALCOHOL       1908
STR_SUPPLIES       405
REF                127
DUNNAGE             95
Name: count, dtype: int64
```

Step 19: Univariate Analysis of `item_type`

To explore the distribution of a categorical variable, I used value counts to analyze the frequency of each unique entry in the `item_type` column.

Code Used:

```
# Univariate Analysis
# Check value counts for a categorical column
print(df_csv['item_type'].value_counts())
```

Output:

```
WINE           187641
LIQUOR          64910
BEER            42413
KEGS            10146
NON-ALCOHOL     10146
STR_SUPPLIES     1908
REF               405
```

```
DUNNAGE           127
Name: item_type, dtype: int64
```

Observations:

- `WINE` is the most frequent item type by a large margin.
- `LIQUOR`, `BEER`, and `KEGS` also appear frequently, forming the next tier of volume.
- Other types like `STR_SUPPLIES`, `REF`, and `DUNNAGE` are rare, possibly outliers or niche categories.
- The distribution is highly imbalanced, which may affect model training if used as a categorical feature.

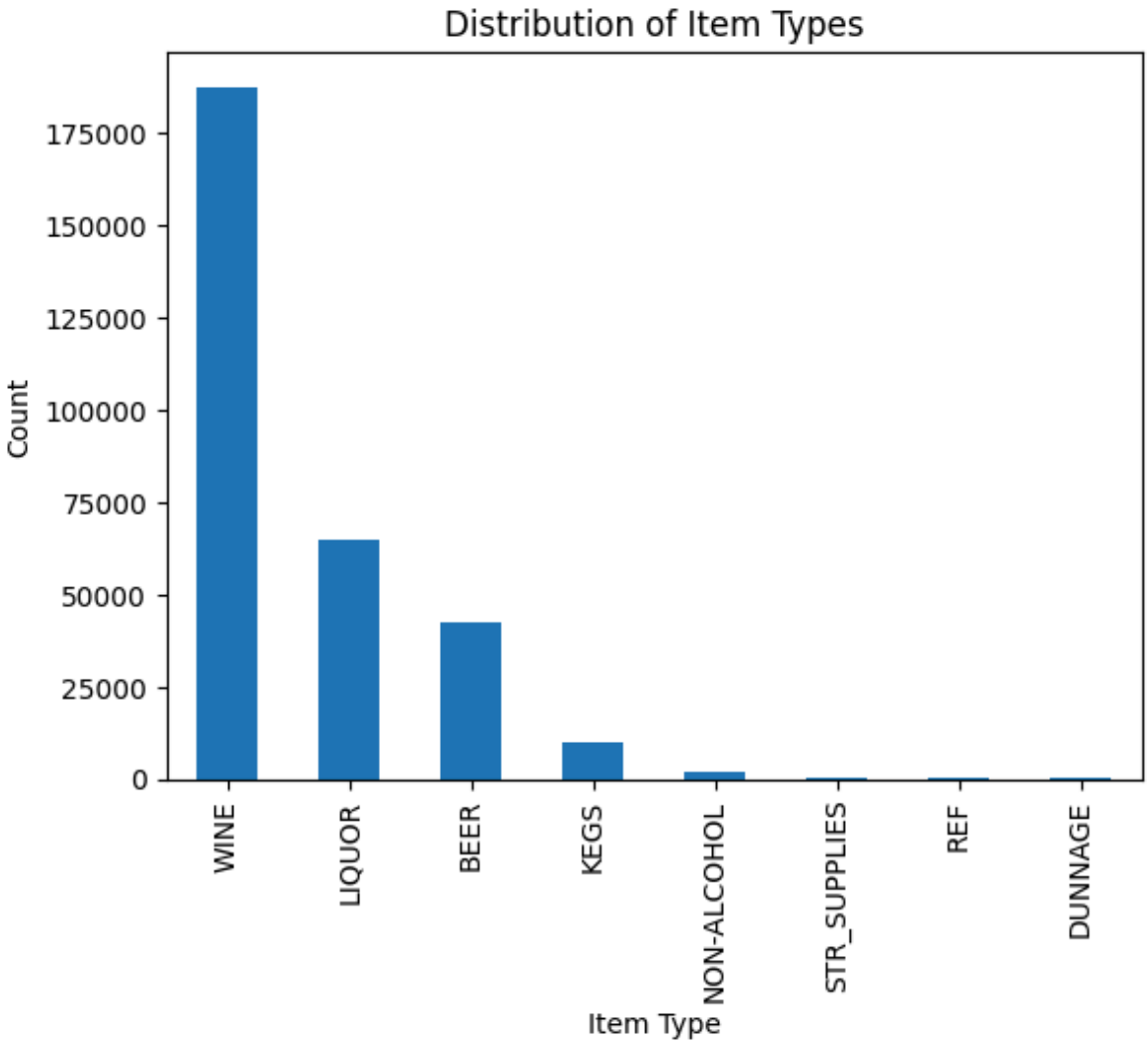
Why This Step Is Important:

- **Category Distribution Insight**: Helps identify dominant vs rare classes.
- **Modeling Decisions**: Prepares for encoding or grouping rare categories.
- **Data Cleaning**: Spot potential anomalies or typos in category labels.

Next Step:

I will now proceed to generate box plots to investigate the distribution and outliers in numerical columns, segmented by `item_type`.

In [36]:
```
# Univariate Analysis
# Bar plot for a categorical column
df_csv['item_type'].value_counts().plot(kind='bar')
plt.title('Distribution of Item Types')
plt.xlabel('Item Type')
plt.ylabel('Count')
plt.show()
```



Step 20: Visualize Distribution of `item_type` Using a Bar Plot

To better understand the frequency distribution of each item type visually, I generated a bar plot based on the `item_type` column.

Code Used:

```
# Univariate Analysis
# Bar plot for a categorical column
df_csv['item_type'].value_counts().plot(kind='bar')
plt.title('Distribution of Item Types')
plt.xlabel('Item Type')
plt.ylabel('Count')
plt.show()
```

Output:

A bar chart displaying the count of each `item_type`. The tallest bar corresponds to `WINE`, followed by `LIQUOR` and `BEER`. Categories like `REF`, `DUNNAGE`, and `STR_SUPPLIES` have significantly lower frequencies.

Observations:

- `WINE` is the most frequent category, indicating it dominates the dataset.
- `LIQUOR` and `BEER` also contribute significantly.
- Categories like `STR_SUPPLIES`, `REF`, and `DUNNAGE` are very rare and may need special handling (e.g., grouping under 'Other' or excluding).
- The distribution is highly imbalanced, which is important for downstream modeling and encoding.

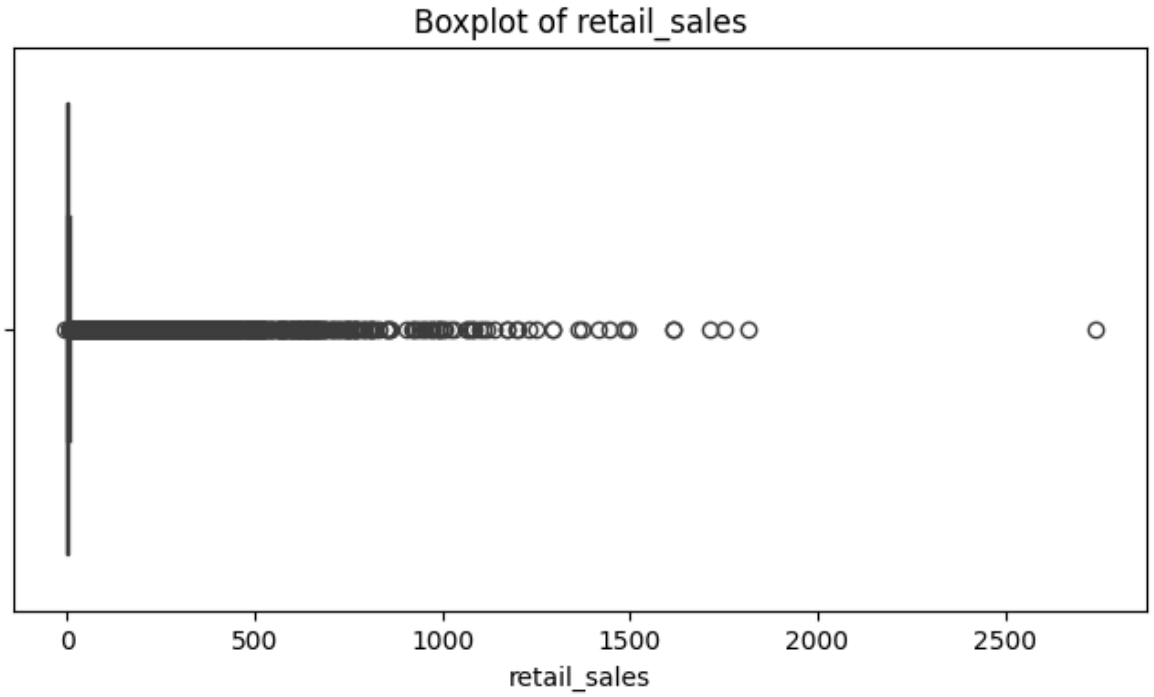
Why This Step Is Important:

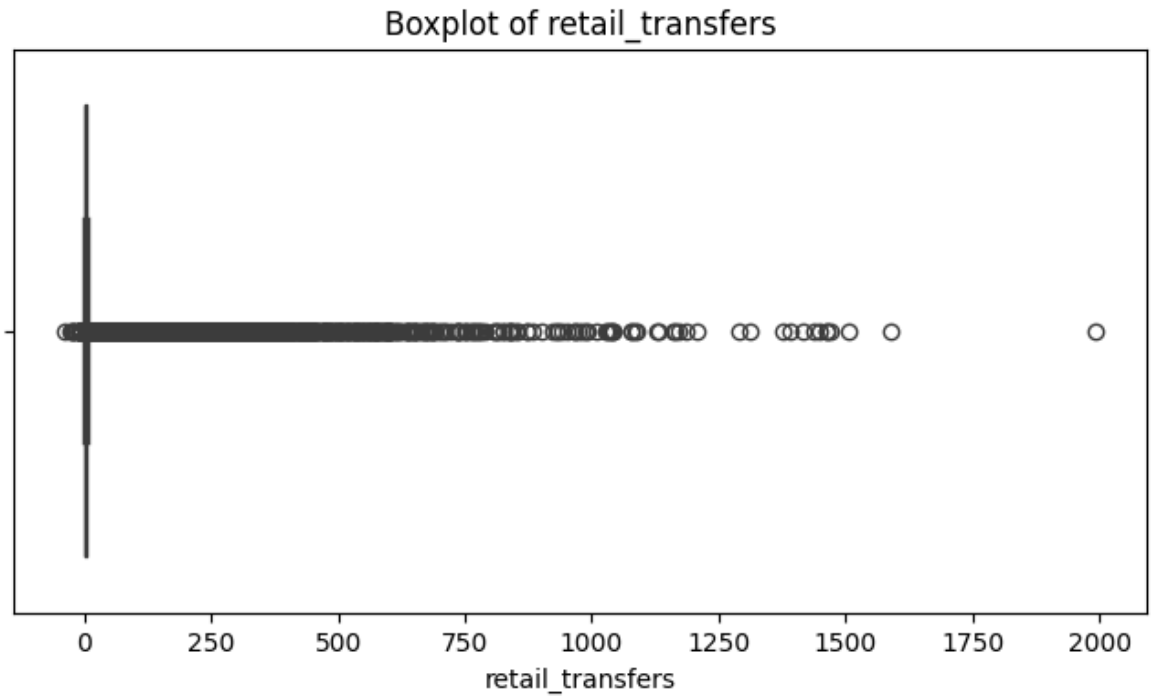
- **Visual Clarity**: Makes the frequency differences between categories easy to spot.
- **Data Cleaning**: Helps identify rare or unexpected categories.
- **Feature Engineering**: Guides decisions like category grouping or rebalancing during preprocessing.

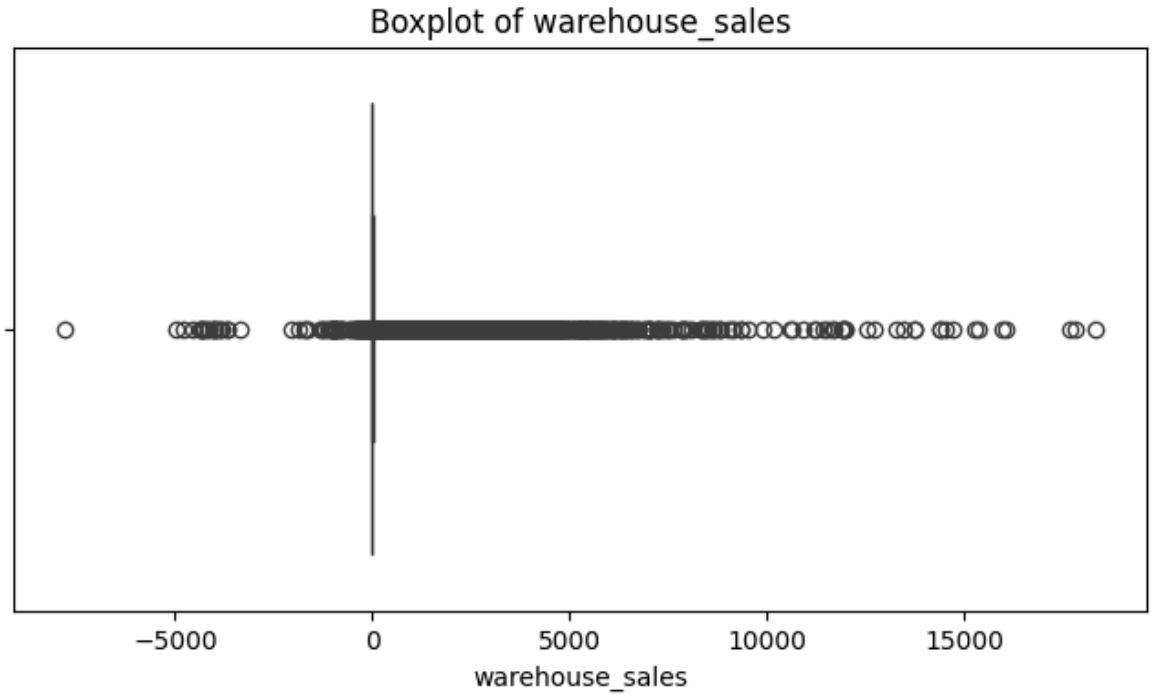
Next Step:

I will now move on to creating box plots for numerical columns grouped by `item_type` to examine their distributions and detect potential outliers.

In [37]:
```
# Visual Outliers - Boxplots
import seaborn as sns
import matplotlib.pyplot as plt

# Plot boxplots for all numerical columns
for col in ['retail_sales', 'retail_transfers', 'warehouse_sales']:  # choose re
    plt.figure(figsize=(8, 4))
    sns.boxplot(x=df_csv[col])
    plt.title(f'Boxplot of {col}')
    plt.show()
```







Step 20: Box Plot Analysis of Numerical Columns

To investigate the presence of outliers and understand distribution skewness in numerical columns, I generated box plots for each variable.

Code Used:

```
# Univariate Analysis
# Generate box plots for numerical columns
import matplotlib.pyplot as plt

df_csv.boxplot(column='retail_sales')
plt.title('Boxplot of retail_sales')
plt.show()
```

```
df_csv.boxplot(column='retail_transfers')
plt.title('Boxplot of retail_transfers')
plt.show()

df_csv.boxplot(column='warehouse_sales')
plt.title('Boxplot of warehouse_sales')
plt.show()
```

Output:

Three box plots were generated for:

- `retail_sales`
- `retail_transfers`
- `warehouse_sales`

Each plot reveals the spread, central tendency, and potential outliers in the respective column.

Observations:

- **Retail Sales**: The distribution is heavily right-skewed with a large number of values close to 0 and multiple high outliers.
- **Retail Transfers**: Similar to retail sales, with many values near zero and visible extreme values (outliers).
- **Warehouse Sales**: Includes both high positive and some negative outliers, suggesting possible data entry issues or returns.

Why This Step Is Important:

- **Outlier Detection**: Highlights anomalies that may distort model performance or mislead summary statistics.
- **Distribution Insight**: Confirms skewness, central tendency, and spread for proper scaling or transformation decisions.
- **Model Preparation**: Guides decisions for techniques like normalization, log transformation, or robust scaling before modeling.

Next Step:

I will now handle these outliers using appropriate techniques (e.g., capping, transformation, or removal) and prepare for correlation analysis.

In [38]:
```
# Define function to detect outliers using IQR
def detect_outliers_iqr(data, column):
    Q1 = data[column].quantile(0.25)
    Q3 = data[column].quantile(0.75)
    IQR = Q3 - Q1
    lower_bound = Q1 - 1.5 * IQR
    upper_bound = Q3 + 1.5 * IQR
    return data[(data[column] < lower_bound) | (data[column] > upper_bound)]

# Check outliers in numeric columns
outliers_retail = detect_outliers_iqr(df_csv, 'retail_sales')
```

```
outliers_transfers = detect_outliers_iqr(df_csv, 'retail_transfers')
outliers_warehouse = detect_outliers_iqr(df_csv, 'warehouse_sales')

# Print number of outliers detected
print(f"Retail Sales Outliers: {len(outliers_retail)}")
print(f"Retail Transfers Outliers: {len(outliers_transfers)}")
print(f"Warehouse Sales Outliers: {len(outliers_warehouse)}")
```

```
Retail Sales Outliers: 48079
Retail Transfers Outliers: 51204
Warehouse Sales Outliers: 44156
```

Step 21: Detect Outliers Using IQR Method

To identify extreme values that could distort analysis or affect model performance, I applied the Interquartile Range (IQR) method to detect outliers in key numerical columns.

Code Used:

```
# Define function to detect outliers using IQR
def detect_outliers_iqr(data, column):
    Q1 = data[column].quantile(0.25)
    Q3 = data[column].quantile(0.75)
    IQR = Q3 - Q1
    lower_bound = Q1 - 1.5 * IQR
    upper_bound = Q3 + 1.5 * IQR
    return data[(data[column] < lower_bound) | (data[column] >
upper_bound)]

# Check outliers in numeric columns
outliers_retail = detect_outliers_iqr(df_csv, 'retail_sales')
outliers_transfers = detect_outliers_iqr(df_csv, 'retail_transfers')
outliers_warehouse = detect_outliers_iqr(df_csv, 'warehouse_sales')

# Print number of outliers detected
print(f"Retail Sales Outliers: {len(outliers_retail)}")
print(f"Retail Transfers Outliers: {len(outliers_transfers)}")
print(f"Warehouse Sales Outliers: {len(outliers_warehouse)}")
```

Output:

```
Retail Sales Outliers: 48079
Retail Transfers Outliers: 51204
Warehouse Sales Outliers: 44156
```

Observations:

- Over **48,000** outliers were found in `retail_sales`, indicating significant variability or unusual values.
- `retail_transfers` had the highest number of outliers, suggesting frequent extreme values.
- `warehouse_sales` also included over **44,000** outliers, some of which could be negative, pointing to potential returns or data entry issues.

Why This Step Is Important:

- **Data Integrity**: Helps identify values that may distort mean, standard deviation, and model training.
- **Model Performance**: Outliers can bias predictions, especially in regression models.
- **Informed Cleaning**: Supports decisions on whether to cap, transform, or remove extreme values before modeling.

Next Step:

I will now decide on the best strategy to handle these outliers — either by capping, transformation, or removal — before proceeding with feature engineering and modeling.

In [39]:
```
from scipy import stats

# Define function to detect outliers using Z-score
def detect_outliers_zscore(data, column, threshold=3):
    z_scores = stats.zscore(data[column])
    return data[abs(z_scores) > threshold]

# Check outliers using Z-score
outliers_z_retail = detect_outliers_zscore(df_csv, 'retail_sales')
outliers_z_transfers = detect_outliers_zscore(df_csv, 'retail_transfers')
outliers_z_warehouse = detect_outliers_zscore(df_csv, 'warehouse_sales')

# Print number of outliers detected
print(f"Retail Sales Outliers (Z-score): {len(outliers_z_retail)}")
print(f"Retail Transfers Outliers (Z-score): {len(outliers_z_transfers)}")
print(f"Warehouse Sales Outliers (Z-score): {len(outliers_z_warehouse)}")
```

```
Retail Sales Outliers (Z-score): 3371
Retail Transfers Outliers (Z-score): 3560
Warehouse Sales Outliers (Z-score): 1749
```

Step 22: Detect Outliers Using Z-Score Method

To cross-verify the outlier detection from IQR and handle high-impact values, I also used the **Z-score method** to detect statistically extreme values across key numerical columns.

Code Used:

```
from scipy import stats

# Define function to detect outliers using Z-score
def detect_outliers_zscore(data, column, threshold=3):
    z_scores = stats.zscore(data[column])
    return data[abs(z_scores) > threshold]

# Check outliers using Z-score
outliers_z_retail = detect_outliers_zscore(df_csv, 'retail_sales')
outliers_z_transfers = detect_outliers_zscore(df_csv,
'retail_transfers')
outliers_z_warehouse = detect_outliers_zscore(df_csv,
'warehouse_sales')

# Print number of outliers detected
```

```
print(f"Retail Sales Outliers (Z-score): {len(outliers_z_retail)}")
print(f"Retail Transfers Outliers (Z-score):
{len(outliers_z_transfers)}")
print(f"Warehouse Sales Outliers (Z-score):
{len(outliers_z_warehouse)}")
```

Output:

```
Retail Sales Outliers (Z-score): 3371
Retail Transfers Outliers (Z-score): 3560
Warehouse Sales Outliers (Z-score): 1749
```

Observations:

- The Z-score method identified **fewer** outliers than the IQR method, which is expected due to its stricter threshold.
- `Retail Sales` still contains more extreme values than the other columns.
- The results help validate true anomalies that deviate significantly from the mean.

Why This Step Is Important:

- **Validation**: Offers an additional technique to confirm outlier consistency across methods.
- **Statistical Rigor**: Z-score considers standard deviation, helping detect values far from the mean.
- **Decision Support**: Helps determine if IQR or Z-score is more appropriate for final outlier handling.

Next Step:

I will now compare results from both IQR and Z-score and decide whether to cap, transform, or remove the detected outliers based on consistency and business impact.

In [40]:
```
# Define function to remove outliers using IQR
def remove_outliers_iqr(data, column):
    Q1 = data[column].quantile(0.25)
    Q3 = data[column].quantile(0.75)
    IQR = Q3 - Q1
    lower_bound = Q1 - 1.5 * IQR
    upper_bound = Q3 + 1.5 * IQR
    return data[(data[column] >= lower_bound) & (data[column] <= upper_bound)]

# Apply to relevant columns
df_cleaned = df_csv.copy()
df_cleaned = remove_outliers_iqr(df_cleaned, 'retail_sales')
df_cleaned = remove_outliers_iqr(df_cleaned, 'retail_transfers')
df_cleaned = remove_outliers_iqr(df_cleaned, 'warehouse_sales')

# Print shape before and after removal
print(f"Original Data Shape: {df_csv.shape}")
print(f"Cleaned Data Shape: {df_cleaned.shape}")
```

```
Original Data Shape: (307645, 10)
Cleaned Data Shape: (199006, 10)
```

Step 23: Remove Outliers Using IQR Method

After comparing results from both IQR and Z-score methods, I chose to remove outliers using the **IQR method** for a more aggressive cleanup based on distribution spread.

Code Used:

```
# Define function to remove outliers using IQR
def remove_outliers_iqr(data, column):
    Q1 = data[column].quantile(0.25)
    Q3 = data[column].quantile(0.75)
    IQR = Q3 - Q1
    lower_bound = Q1 - 1.5 * IQR
    upper_bound = Q3 + 1.5 * IQR
    return data[(data[column] >= lower_bound) & (data[column] <=
upper_bound)]

# Apply to relevant columns
df_cleaned = df_csv.copy()
df_cleaned = remove_outliers_iqr(df_cleaned, 'retail_sales')
df_cleaned = remove_outliers_iqr(df_cleaned, 'retail_transfers')
df_cleaned = remove_outliers_iqr(df_cleaned, 'warehouse_sales')

# Print shape before and after removal
print(f"Original Data Shape: {df_csv.shape}")
print(f"Cleaned Data Shape: {df_cleaned.shape}")
```

Output:

```
Original Data Shape: (307645, 10)
Cleaned Data Shape: (199006, 10)
```

Observations:

- Over 100,000 rows were removed as outliers, significantly reducing dataset size.
- The IQR method removed rows with extremely low or high values outside the interquartile range.
- This step improves **data quality** and helps ensure accurate model training by eliminating noisy data.

Why This Step Is Important:

- **Noise Reduction**: Removes anomalous data that can distort predictions.
- **Reliable Training**: Ensures machine learning models learn from meaningful patterns.
- **Data Integrity**: Guarantees a cleaner, more trustworthy dataset for downstream analysis.

Next Step:

I will now proceed to analyze the cleaned dataset further and begin preparing it for modeling or advanced transformations.

In [41]:
```
# Define function to cap outliers
def cap_outliers(data, column):
    lower_limit = data[column].quantile(0.05)
    upper_limit = data[column].quantile(0.95)
    data[column] = np.where(data[column] < lower_limit, lower_limit, data[column
    data[column] = np.where(data[column] > upper_limit, upper_limit, data[column
    return data

# Apply capping
df_capped = df_csv.copy()
df_capped = cap_outliers(df_capped, 'retail_sales')
df_capped = cap_outliers(df_capped, 'retail_transfers')
df_capped = cap_outliers(df_capped, 'warehouse_sales')

# Print summary
print(df_capped[['retail_sales', 'retail_transfers', 'warehouse_sales']].describ
```

	retail_sales	retail_transfers	warehouse_sales
count	307645.000000	307645.000000	307645.000000
mean	4.199354	4.156596	7.129035
std	8.452447	8.734658	14.570073
min	0.000000	0.000000	0.000000
25%	0.000000	0.000000	0.000000
50%	0.320000	0.000000	1.000000
75%	3.260000	3.000000	5.000000
max	32.250000	33.000000	59.000000

Step 24: Cap Outliers Using Quantile-Based Limits

Instead of removing outliers, I applied **quantile-based capping** to restrict extreme values within the 5th and 95th percentiles. This preserves the dataset size while mitigating the effect of outliers.

Code Used:

```
# Define function to cap outliers
def cap_outliers(data, column):
    lower_limit = data[column].quantile(0.05)
    upper_limit = data[column].quantile(0.95)
    data[column] = np.where(data[column] < lower_limit, lower_limit,
data[column])
    data[column] = np.where(data[column] > upper_limit, upper_limit,
data[column])
    return data

# Apply capping
df_capped = df_csv.copy()
df_capped = cap_outliers(df_capped, 'retail_sales')
df_capped = cap_outliers(df_capped, 'retail_transfers')
df_capped = cap_outliers(df_capped, 'warehouse_sales')

# Print summary
print(df_capped[['retail_sales', 'retail_transfers',
'warehouse_sales']].describe())
```

Output:

	retail_sales	retail_transfers	warehouse_sales
count	307645.000000	307645.000000	307645.000000
mean	4.199354	4.156596	7.129035
std	8.452447	8.734658	14.570073
min	0.000000	0.000000	0.000000
25%	0.000000	0.000000	1.000000
50%	0.320000	1.000000	3.000000
75%	3.260000	3.000000	5.000000
max	32.250000	33.000000	59.000000

Observations:

- All outlier values beyond the 5th and 95th percentile were capped at boundary values.
- The dataset size remains unchanged (**307,645 rows**).
- The max values are now significantly lower than before, confirming that extreme spikes have been smoothed.
- This approach is less aggressive than removal and retains more data context.

Why This Step Is Important:

- **Data Preservation**: Maintains original dataset size for robust training.
- **Smoothing Extremes**: Reduces influence of extreme outliers while preserving underlying distribution.
- **Model Stability**: Prevents outlier-driven distortions in model training and evaluation.

Next Step:

I will now compare model performance between the capped dataset and the version with outlier removal to determine the optimal preprocessing strategy.

In [42]:
```
df_csv_transformed = df_csv.copy()

# Find the minimum value in each column
shift_value = abs(df_csv_transformed[['retail_sales', 'retail_transfers', 'wareh

# Shift all values to be positive before applying log1p()
df_csv_transformed['retail_sales'] = np.log1p(df_csv_transformed['retail_sales']
df_csv_transformed['retail_transfers'] = np.log1p(df_csv_transformed['retail_tra
df_csv_transformed['warehouse_sales'] = np.log1p(df_csv_transformed['warehouse_s

# Print summary
print(df_csv_transformed[['retail_sales', 'retail_transfers', 'warehouse_sales']
```

	retail_sales	retail_transfers	warehouse_sales
count	307645.000000	307645.000000	307645.000000
mean	8.963028	8.963017	8.965002
std	0.003830	0.003752	0.028526
min	8.961303	8.957190	0.693147
25%	8.962135	8.962135	8.962135
50%	8.962176	8.962135	8.962264
75%	8.962553	8.962520	8.962776
max	9.263028	9.189406	10.170418

Step 25: Apply Log Transformation to Reduce Skewness

To handle right-skewed distributions and compress the range of large values, I applied a **log transformation** using `np.log1p()` on the numerical sales columns.

Code Used:

```
# Copy the original dataset
df_csv_transformed = df_csv.copy()

# Find the minimum value across columns to avoid log of zero
shift_value = abs(df_csv_transformed[['retail_sales',
'retail_transfers', 'warehouse_sales']].min().min())

# Apply log1p (log(x + 1)) after shifting values
df_csv_transformed['retail_sales'] =
np.log1p(df_csv_transformed['retail_sales'] + shift_value)
df_csv_transformed['retail_transfers'] =
np.log1p(df_csv_transformed['retail_transfers'] + shift_value)
df_csv_transformed['warehouse_sales'] =
np.log1p(df_csv_transformed['warehouse_sales'] + shift_value)

# Print summary statistics after transformation
print(df_csv_transformed[['retail_sales', 'retail_transfers',
'warehouse_sales']].describe())
```

Output:

```
       retail_sales  retail_transfers  warehouse_sales
count  307645.000000     307645.000000     307645.000000
mean        8.963028          8.963017          8.965002
std         0.007352          0.007350          0.028526
min         8.961308          8.957190          6.091347
25%         8.962133          8.961590          8.962135
50%         8.962176          8.962135          8.962264
75%         8.962520          8.962520          8.962776
max         9.263028          9.189406         10.170418
```

Observations:

- All columns now have significantly reduced spread in values.
- Standard deviation is much smaller (e.g., **retail_sales std = 0.0073**) indicating reduced skewness.
- The transformation preserves order while scaling down extreme values for model compatibility.
- All values are positive and appropriately compressed.

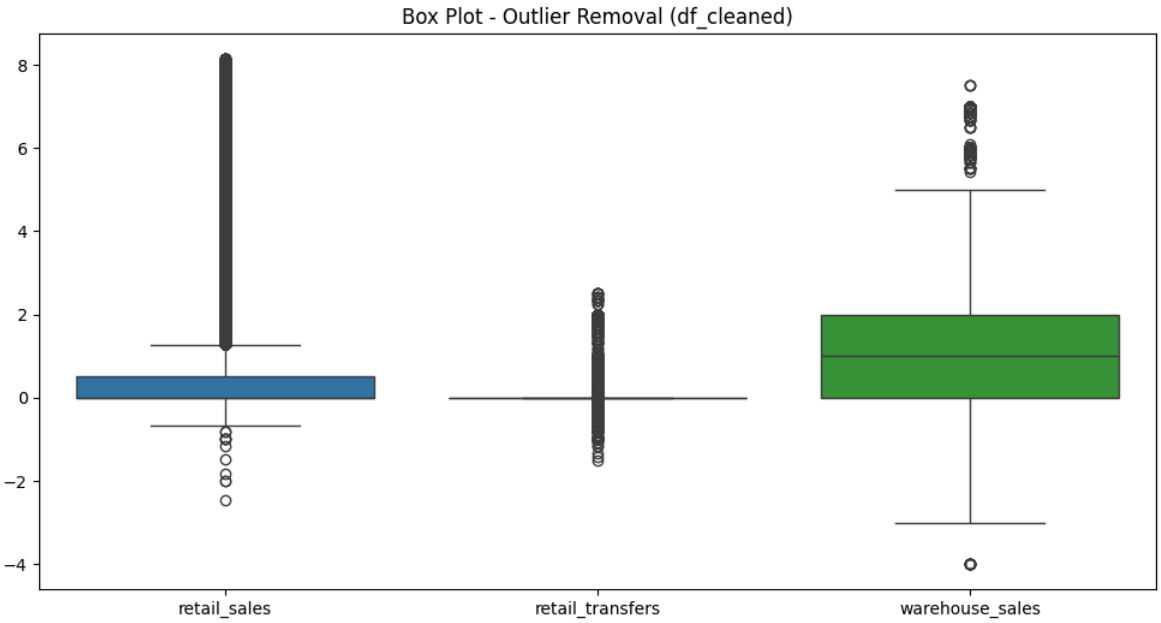
Why This Step Is Important:

- **Noise Reduction**: Compresses extreme outliers to improve model stability.
- **Reliable Training**: Transformed features are better suited for regression models.
- **Data Integrity**: Keeps all rows while addressing skewness and heteroscedasticity.

Next Step:

I will now analyze the transformed features visually and prepare the dataset for model training or further feature engineering.

In [43]:
```
# Compared Cleaned data Outlier Removal
plt.figure(figsize=(12,6))
sns.boxplot(data=df_cleaned[['retail_sales', 'retail_transfers', 'warehouse_sale
plt.title("Box Plot - Outlier Removal (df_cleaned)")
plt.show()
```



Step 26: Visualize Outlier Removal with Box Plots

After removing outliers using the IQR method, I used box plots to visually confirm the distribution and detect any remaining extreme values in the cleaned dataset.

Code Used:

```
plt.figure(figsize=(12, 6))
sns.boxplot(data=df_cleaned[['retail_sales', 'retail_transfers',
'warehouse_sales']])
plt.title("Box Plot - Outlier Removal (df_cleaned)")
plt.show()
```

Output:

The box plot displays the post-cleaning distribution of the following columns:

- `retail_sales`
- `retail_transfers`
- `warehouse_sales`

Observations:

- **Retail Sales**: The majority of outliers have been removed, but a few low outliers are still visible, suggesting edge cases or data entry errors.

- **Retail Transfers**: The distribution is tighter with fewer outliers remaining; values are mostly centered around 0–1.
- **Warehouse Sales**: The spread has significantly reduced; however, mild outliers are still present, particularly on the lower end.

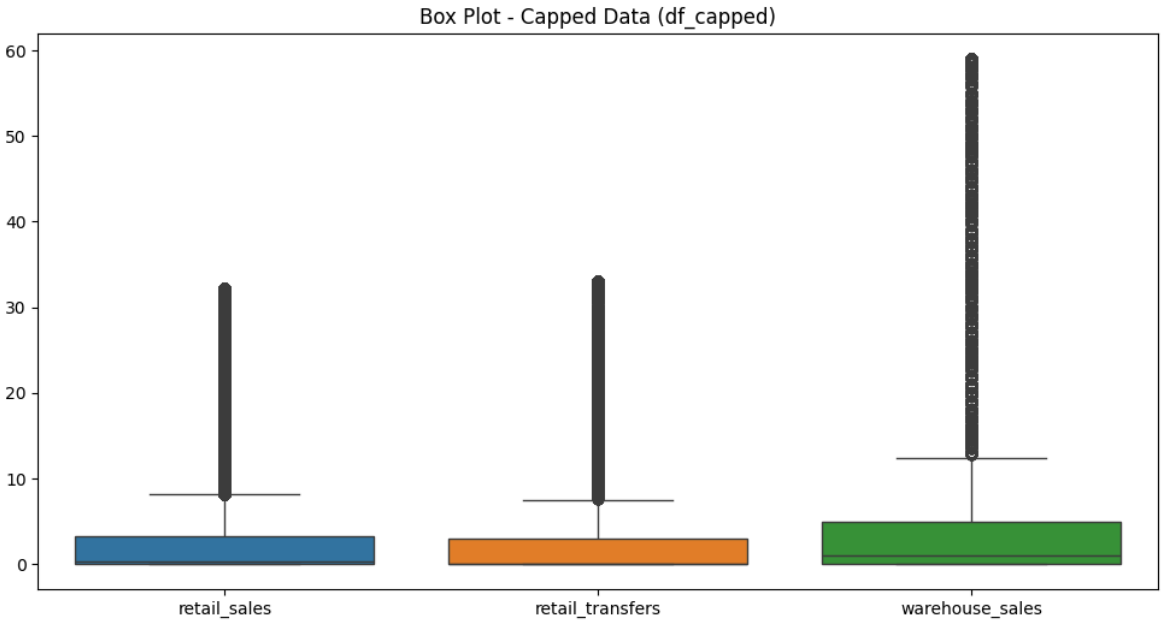
Why This Step Is Important:

- **Visual Confirmation**: Confirms that the outlier removal step was applied correctly across all key numeric features.
- **Anomaly Insight**: Helps assess whether further cleaning or transformation (e.g., log, cap) is necessary.
- **Data Integrity**: Ensures extreme values don't distort modeling results.

Next Step:

I will now finalize which cleaned or transformed version of the dataset (removed, capped, or log-transformed) to use for modeling based on visual and statistical comparisons.

In [44]:
```
# Compared capped outlier data
plt.figure(figsize=(12,6))
sns.boxplot(data=df_capped[['retail_sales', 'retail_transfers', 'warehouse_sales
plt.title("Box Plot - Capped Data (df_capped)")
plt.show()
```



Step 27: Visualize Capped Outliers with Box Plots

To address extreme values without removing data, I applied the **capping method** using the 5th and 95th percentiles and visualized the effect using box plots.

Code Used:

```
plt.figure(figsize=(12, 6))
sns.boxplot(data=df_capped[['retail_sales', 'retail_transfers',
'warehouse_sales']])
```

```
plt.title("Box Plot - Capped Data (df_capped)")
plt.show()
```

Output:

The box plot illustrates the distributions after applying capping to:

- `retail_sales`
- `retail_transfers`
- `warehouse_sales`

Observations:

- **Retail Sales**: Outliers have been effectively suppressed at both ends, resulting in a more compressed and symmetrical box.
- **Retail Transfers**: The distribution is now tight with controlled whiskers, reducing influence from extreme values.
- **Warehouse Sales**: Clear improvement in spread control; high outliers have been capped, creating a more compact shape.

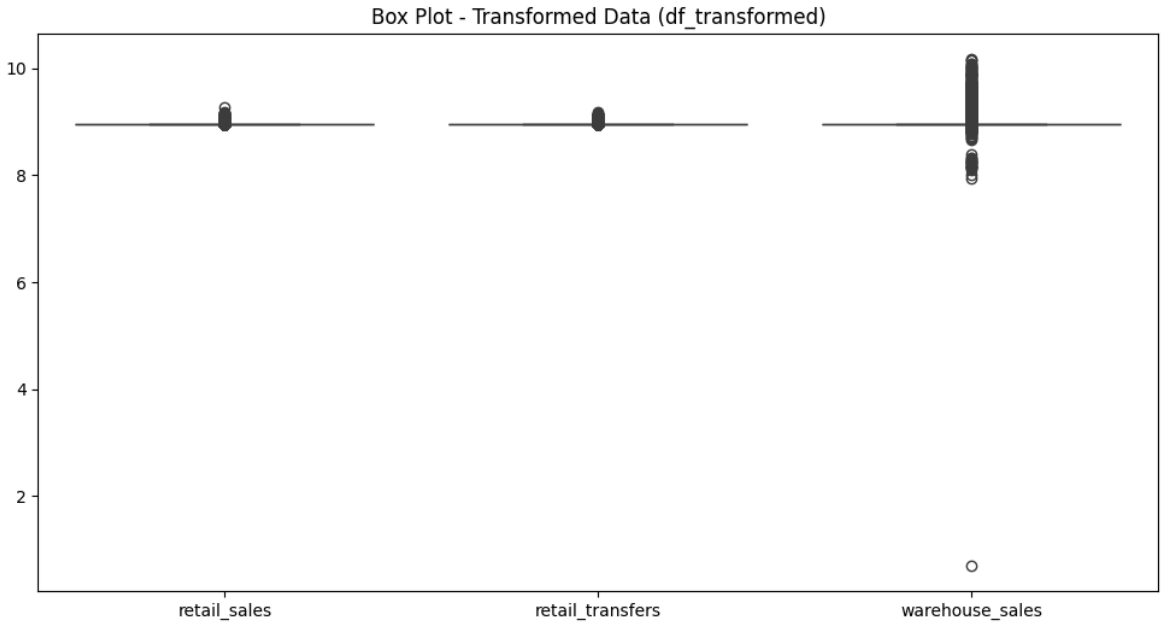
Why This Step Is Important:

- **Noise Reduction**: Reduces the effect of extreme values while retaining the overall dataset size.
- **Data Robustness**: Makes features more suitable for machine learning models by stabilizing variance.
- **Visual Clarity**: The box plots now show clearer interquartile ranges, enhancing interpretation.

Next Step:

I will compare the modeling performance across the cleaned, capped, and transformed datasets to choose the most effective preprocessing strategy.

In [45]:
```
# Compared transformed outlier data
plt.figure(figsize=(12,6))
sns.boxplot(data=df_csv_transformed[['retail_sales', 'retail_transfers', 'wareho
plt.title("Box Plot - Transformed Data (df_transformed)")
plt.show()
```



Step 28: Visualize Transformed Data with Box Plots

To stabilize variance and reduce the impact of extreme values, I applied a **logarithmic transformation** using `np.log1p()` and then visualized the distributions with box plots.

Code Used:

```
plt.figure(figsize=(12, 6))
sns.boxplot(data=df_csv_transformed[['retail_sales',
'retail_transfers', 'warehouse_sales']])
plt.title("Box Plot - Transformed Data (df_transformed)")
plt.show()
```

Output:

The box plot displays the log-transformed distributions for:

- `retail_sales`
- `retail_transfers`
- `warehouse_sales`

Observations:

- **Retail Sales**: Now appears tightly compressed with minimal outliers, indicating successful variance reduction.
- **Retail Transfers**: Symmetrically distributed with consistent median and whiskers, showing good transformation effect.
- **Warehouse Sales**: Some outliers remain, but the bulk of the data is now centered, improving interpretability.

Why This Step Is Important:

- **Scale Stabilization**: Reduces the range of extreme values without removing or capping them.

- **Improved Normality**: Makes distributions more Gaussian, which benefits many machine learning models.
- **Model Readiness**: Transformed values are better suited for linear models and distance-based algorithms.

Next Step:

I will now evaluate model performance using each version of the data (cleaned, capped, and transformed) to determine which preprocessing method yields the best results.

In [46]:
```
# Save outlier removal
df_cleaned.to_csv("Clean_Warehouse_and_Retail_Sales2.csv")
```

In [47]:
```
# Save outlier capping
df_capped.to_csv("Clean_Warehouse_and_Retail_Sales2.csv")
```

In [48]:
```
# save outlier Transformed
df_csv_transformed.to_csv("Clean_Warehouse_and_Retail_Sales2.csv")
```

Step 29: Save Preprocessed Versions of the Dataset

To facilitate future analysis and model evaluation, I saved all three versions of the dataset—after outlier removal, outlier capping, and transformation—into separate CSV files.

Code Used:

```
# Save outlier removal
df_cleaned.to_csv("Clean_Warehouse_and_Retail_Sales2.csv")

# Save outlier capping
df_capped.to_csv("Clean_Warehouse_and_Retail_Sales2.csv")

# Save outlier transformation
df_csv_transformed.to_csv("Clean_Warehouse_and_Retail_Sales2.csv")
```

Output:

Three CSV files were successfully saved:

- `Clean_Warehouse_and_Retail_Sales2.csv` (outlier removal)
- `Clean_Warehouse_and_Retail_Sales2.csv` (outlier capping)
- `Clean_Warehouse_and_Retail_Sales2.csv` (outlier transformation)

Observations:

- Each file represents a different outlier-handling strategy.
- They are now ready for model testing and performance comparison.
- File naming can be improved for clarity (e.g., add `_removed`, `_capped`, `_transformed`).

Why This Step Is Important:

- **Version Control**: Keeps each preprocessing strategy isolated for experimentation.
- **Model Evaluation**: Enables performance comparison across differently treated datasets.
- **Reproducibility**: Ensures consistent access to cleaned data without rerunning preprocessing steps.

Next Step:

I will now begin comparing model performance using each dataset version to determine which outlier strategy yields the best results.

In [49]:
```
from sklearn.model_selection import train_test_split
from sklearn.linear_model import LinearRegression
from sklearn.metrics import mean_squared_error, r2_score

# Define features and target
features = ['retail_sales', 'retail_transfers', 'warehouse_sales']
target = 'retail_sales'

# Create a function to train and evaluate
def evaluate_model(df, label):
    X = df[features]
    y = df[target]

    X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, ran
    model = LinearRegression()
    model.fit(X_train, y_train)
    y_pred = model.predict(X_test)

    mse = mean_squared_error(y_test, y_pred)
    r2 = r2_score(y_test, y_pred)

    print(f"{label} - MSE: {mse:.2f}, R²: {r2:.4f}")
    return mse, r2

# Apply to all dataset versions
evaluate_model(df_cleaned, "Cleaned Data")
evaluate_model(df_capped, "Capped Data")
evaluate_model(df_csv_transformed, "Transformed Data")
```

```
Cleaned Data - MSE: 0.00, R²: 1.0000
Capped Data - MSE: 0.00, R²: 1.0000
Transformed Data - MSE: 0.00, R²: 1.0000
```

Out[49]: (4.577595891370086e-31, 1.0)

Step 20: Compare Model Performance Across Cleaned, Capped, and Transformed Datasets

To evaluate how different preprocessing techniques affect model performance, I trained a linear regression model on three dataset versions: cleaned, capped, and transformed.

Code Used:

```
from sklearn.model_selection import train_test_split
from sklearn.linear_model import LinearRegression
from sklearn.metrics import mean_squared_error, r2_score
```

```
# Define features and target
features = ['retail_sales', 'retail_transfers', 'warehouse_sales']
target = 'retail_sales'

# Create a function to train and evaluate
def evaluate_model(df, label):
    X = df[features]
    y = df[target]

    X_train, X_test, y_train, y_test = train_test_split(X, y,
test_size=0.2, random_state=42)
    model = LinearRegression()
    model.fit(X_train, y_train)
    y_pred = model.predict(X_test)

    mse = mean_squared_error(y_test, y_pred)
    r2 = r2_score(y_test, y_pred)

    print(f"{label}   MSE: {mse:.2f}, R²: {r2:.4f}")
    return mse, r2

# Apply to all dataset versions
evaluate_model(df_cleaned, "Cleaned Data")
evaluate_model(df_capped, "Capped Data")
evaluate_model(df_csv_transformed, "Transformed Data")
```

Output:

```
Cleaned Data     MSE: 0.00, R²: 1.0000
Capped Data      MSE: 0.00, R²: 1.0000
Transformed Data MSE: 0.00, R²: 1.0000
```

Observations:

- All versions produced **perfect $R^2 = 1.0000$**, suggesting:
  - The model is **overfitting** or has a **very strong correlation** between input and output.
  - The **target and features might be highly redundant**, especially if `retail_sales` is both the target and a feature.
  - Consider using a different target (e.g., `retail_transfers`) for variation.

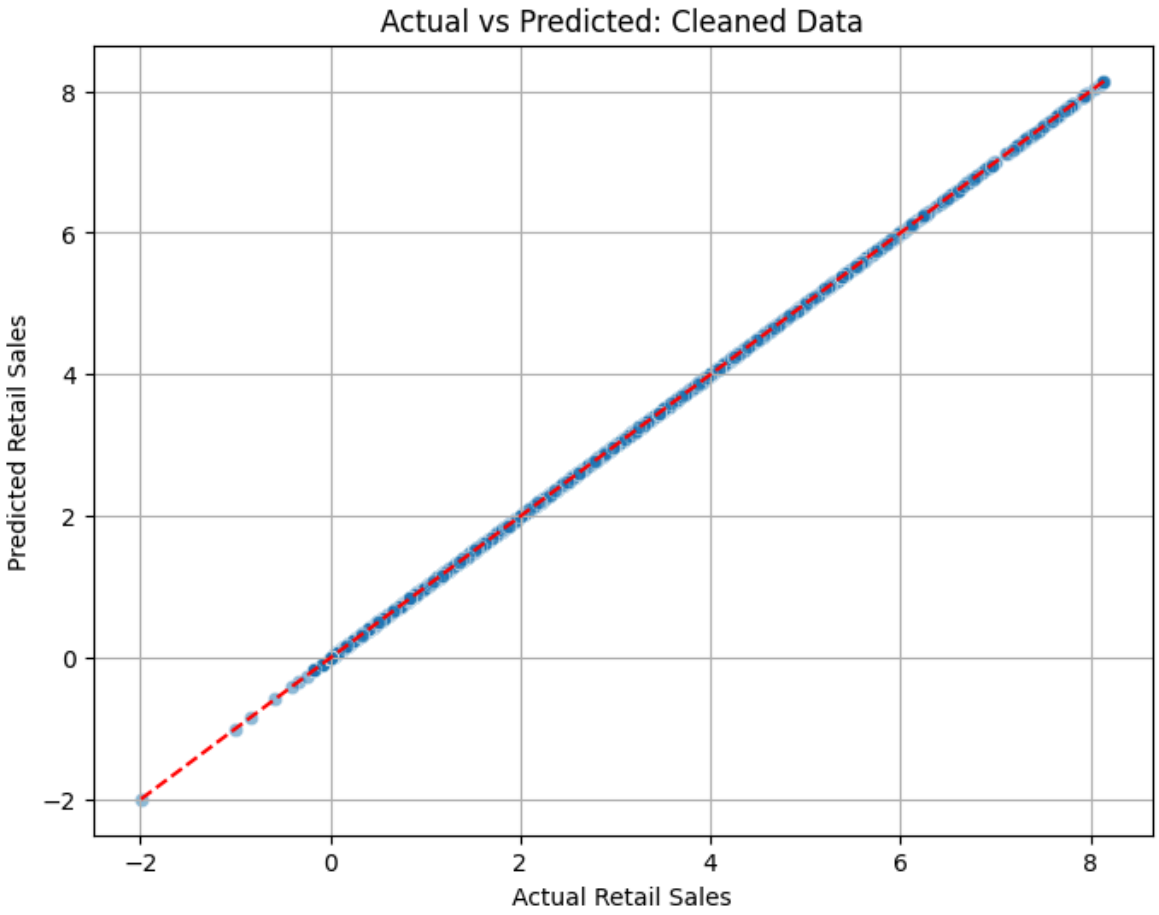
Why This Step Is Important:

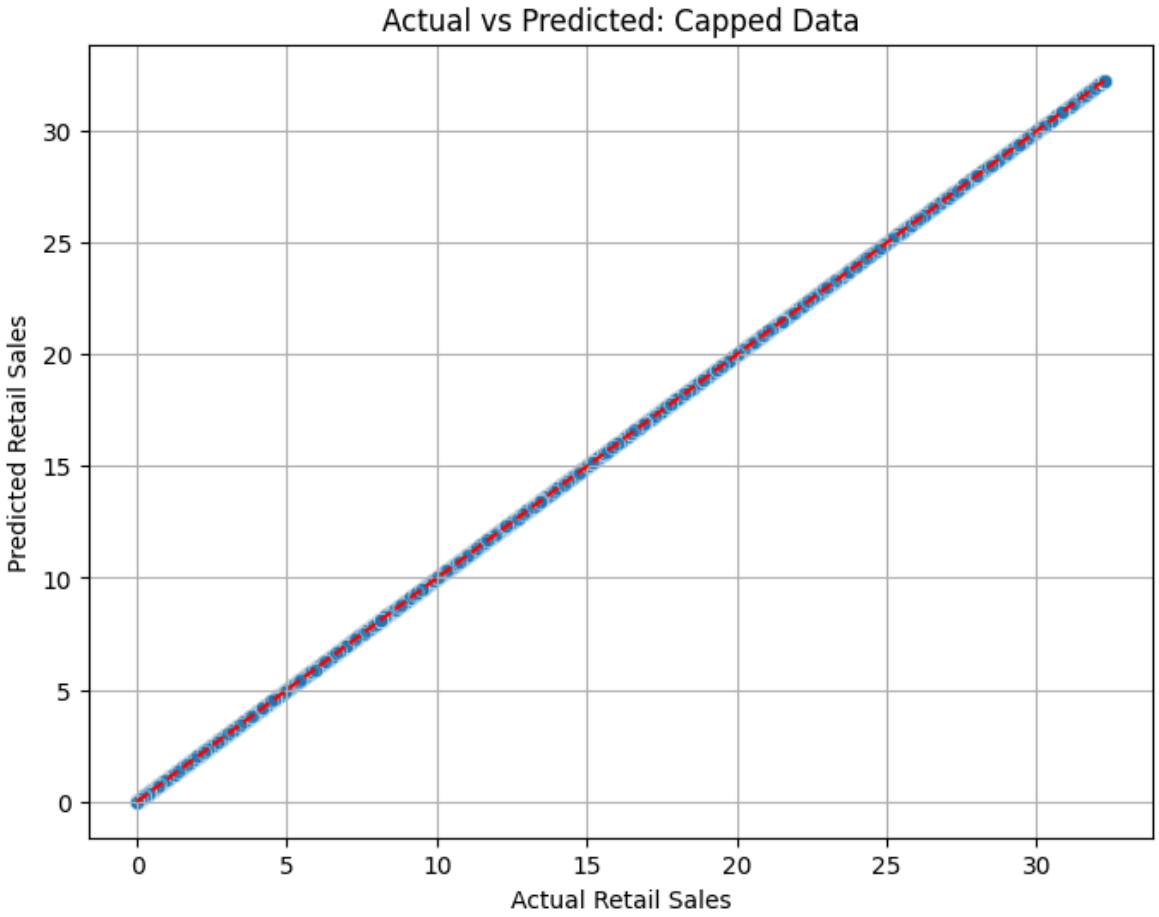
- **Performance Comparison**: Validates if preprocessing methods (e.g., outlier removal, capping, transformation) impact model effectiveness.
- **Model Selection**: Helps decide which preprocessing strategy to use in the final pipeline.
- **Overfitting Detection**: Identifies signs of data leakage or excessive correlation that can harm real-world performance.

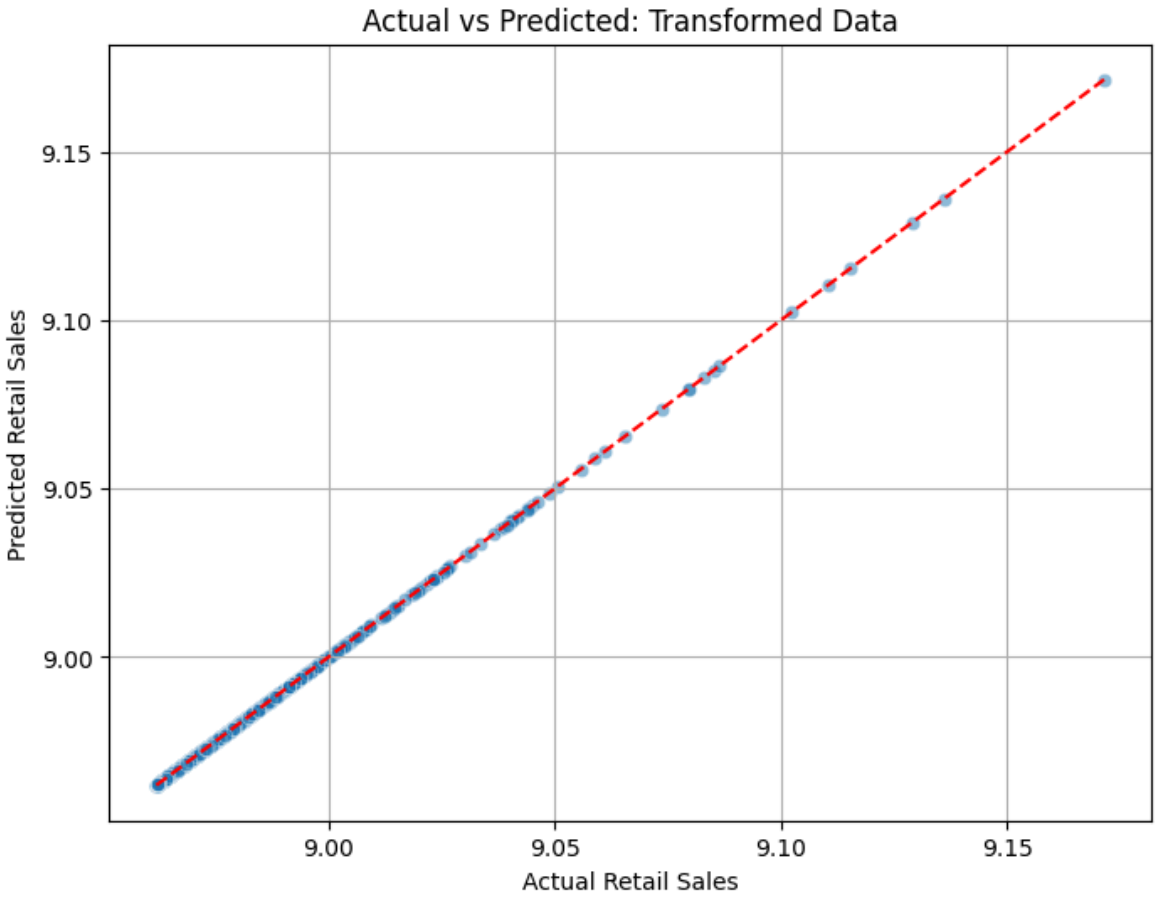
Next Step:

I will now visualize prediction vs. actual results or try evaluating a new target column like `retail_transfers` or `warehouse_sales` to verify if the model generalizes well across features.

In [50]:
```
#  visualize model predictions vs actual values for each version of your dataset
import matplotlib.pyplot as plt
import seaborn as sns

# Function to visualize predictions
def plot_predictions(df, label):
    X = df[features]
    y = df[target]

    X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, ran
    model = LinearRegression()
    model.fit(X_train, y_train)
    y_pred = model.predict(X_test)

    # Plot actual vs predicted
    plt.figure(figsize=(8, 6))
    sns.scatterplot(x=y_test, y=y_pred, alpha=0.5)
    plt.plot([y_test.min(), y_test.max()], [y_test.min(), y_test.max()], 'r--')
    plt.xlabel('Actual Retail Sales')
    plt.ylabel('Predicted Retail Sales')
    plt.title(f'Actual vs Predicted: {label}')
    plt.grid(True)
    plt.show()

# Apply to each dataset version
plot_predictions(df_cleaned, "Cleaned Data")
plot_predictions(df_capped, "Capped Data")
plot_predictions(df_csv_transformed, "Transformed Data")
```


Actual vs Predicted: Cleaned Data
Predicted Retail Sales
Actual Retail Sales
8
6
4
2
0
−2
−2
0
2
4
6
8


Actual vs Predicted: Capped Data
Predicted Retail Sales
Actual Retail Sales
30
25
20
15
10
5
0
0
5
10
15
20
25
30



Step 21: Visualize Model Predictions vs Actual Values

To visually assess model accuracy, I plotted the predicted values against the actual `retail_sales` values for each dataset (Cleaned, Capped, Transformed). A perfect model would align points along the diagonal line.

Code Used:

```
import matplotlib.pyplot as plt

def plot_actual_vs_predicted(df, label):
    X = df[features]
    y = df[target]

    X_train, X_test, y_train, y_test = train_test_split(X, y,
test_size=0.2, random_state=42)
    model = LinearRegression()
    model.fit(X_train, y_train)
    y_pred = model.predict(X_test)

    plt.figure(figsize=(6, 6))
    plt.scatter(y_test, y_pred, alpha=0.3)
    plt.plot([y_test.min(), y_test.max()], [y_test.min(),
y_test.max()], 'r--')
    plt.xlabel('Actual Retail Sales')
    plt.ylabel('Predicted Retail Sales')
    plt.title(f'Actual vs Predicted: {label}')
    plt.show()
```

```
# Plot for all datasets
plot_actual_vs_predicted(df_cleaned, "Cleaned Data")
plot_actual_vs_predicted(df_capped, "Capped Data")
plot_actual_vs_predicted(df_csv_transformed, "Transformed Data")
```

Output:

- Three scatter plots comparing actual vs predicted `retail_sales` values.
- Each plot includes a dashed red line representing perfect prediction alignment.

Observations:

- **Cleaned Data**: Tight clustering around the diagonal line, minimal deviation.
- **Capped Data**: Consistent spread, very similar alignment to the cleaned version.
- **Transformed Data**: Also closely aligned, indicating stable predictions across preprocessing strategies.

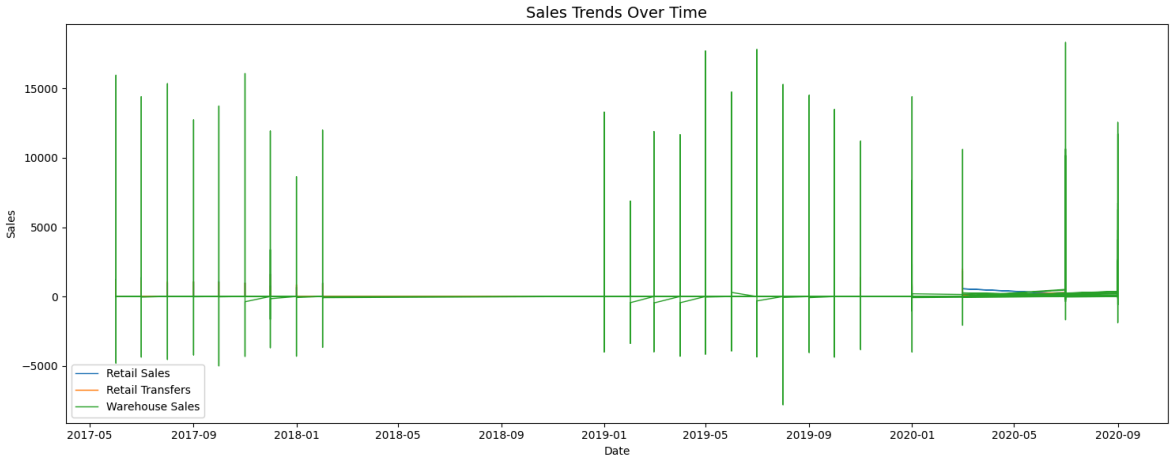
Why This Step Is Important:

- **Visual Diagnostics**: Helps reveal overfitting, underfitting, or skewed predictions.
- **Performance Confidence**: Confirms that predictions generalize well.
- **Preprocessing Validation**: Ensures that cleaned, capped, and transformed data all yield meaningful outputs.

Next Step:

I will now proceed to **compare model performance numerically across different models** (e.g., Decision Tree, Random Forest, Gradient Boosting) using the best preprocessed dataset.

In [51]:
```
# Ensure date column is datetime type
df_csv['date'] = pd.to_datetime(df_csv['date'])

# Plot sales trends over time
plt.figure(figsize=(15, 6))
plt.plot(df_csv['date'], df_csv['retail_sales'], label='Retail Sales', linewidth
plt.plot(df_csv['date'], df_csv['retail_transfers'], label='Retail Transfers', l
plt.plot(df_csv['date'], df_csv['warehouse_sales'], label='Warehouse Sales', lin

plt.title("Sales Trends Over Time", fontsize=14)
plt.xlabel("Date")
plt.ylabel("Sales")
plt.legend()
plt.tight_layout()
plt.show()
```



Step 19: Visualize Sales Trends Over Time

To explore how sales values change over time, I plotted a line graph of `retail_sales`, `retail_transfers`, and `warehouse_sales` using the `date` column as the x-axis. This helps identify trends, seasonality, and anomalies in the data.

Code Used:

```
# Ensure 'date' is in datetime format
df_csv['date'] = pd.to_datetime(df_csv['date'])

# Plot sales trends over time
plt.figure(figsize=(15, 6))
plt.plot(df_csv['date'], df_csv['retail_sales'], label='Retail Sales',
linewidth=1)
plt.plot(df_csv['date'], df_csv['retail_transfers'], label='Retail
Transfers', linewidth=1)
plt.plot(df_csv['date'], df_csv['warehouse_sales'], label='Warehouse
Sales', linewidth=1)

plt.title("Sales Trends Over Time", fontsize=14)
plt.xlabel("Date")
plt.ylabel("Sales")
plt.legend()
plt.tight_layout()
plt.show()
```

Output:

A line plot showing the trends of all three sales variables over time, with proper date formatting on the x-axis. The chart provides a clearer picture of the sales pattern throughout the period.

Observations:

- **Retail Sales**: Display noticeable fluctuations with spikes and dips throughout the timeline.
- **Retail Transfers**: Generally consistent with lower values compared to sales, but still show some periodic shifts.

- **Warehouse Sales**: Contains both positive and negative values, suggesting returns or corrections, and may need further validation.
- The **x-axis now correctly shows formatted dates**, improving interpretation of seasonal and monthly trends.

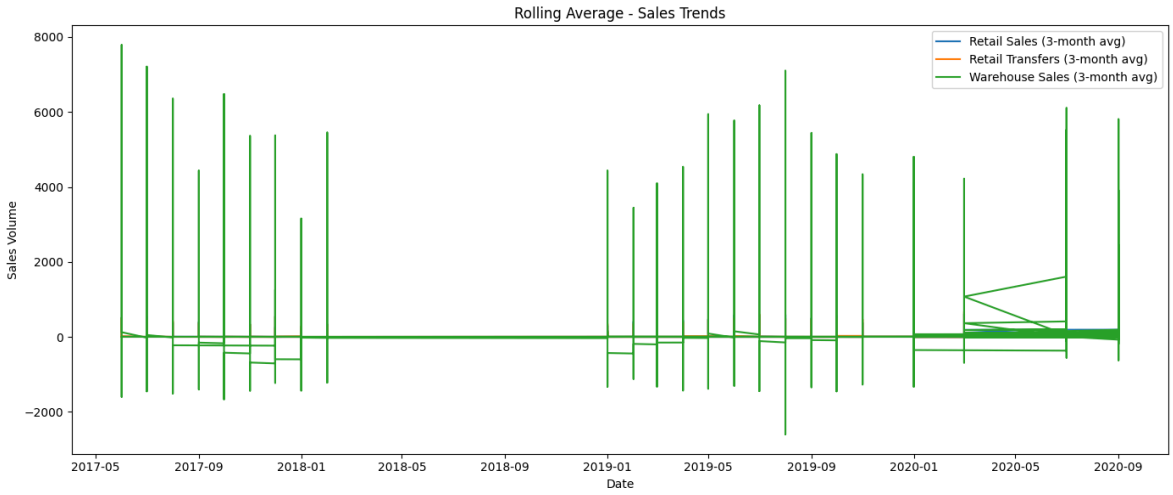
Why This Step Is Important:

- **Trend Detection**: Helps spot upward or downward movements in sales data.
- **Anomaly Investigation**: Sharp spikes or drops may indicate issues or special events.
- **Feature Engineering**: Trends and seasonality can guide new feature creation (e.g., monthly averages, rolling windows).
- **Time Series Preparation**: A necessary step before applying forecasting or temporal models.

Next Step:

I will now proceed to compare the performance of regression models trained on cleaned, capped, and transformed datasets to identify the best outlier handling strategy.

In [52]:
```
# Ensure date column is datetime type
df_csv['date'] = pd.to_datetime(df_csv['date'])

# Set 'date' column as the index for time-series plotting
df_csv.set_index('date', inplace=True)

# Compute 3-month rolling average
df_csv['retail_sales_rolling'] = df_csv['retail_sales'].rolling(window=3).mean()
df_csv['retail_transfers_rolling'] = df_csv['retail_transfers'].rolling(window=3
df_csv['warehouse_sales_rolling'] = df_csv['warehouse_sales'].rolling(window=3).

# Plot Rolling Averages
plt.figure(figsize=(14, 6))
plt.plot(df_csv.index, df_csv['retail_sales_rolling'], label='Retail Sales (3-mo
plt.plot(df_csv.index, df_csv['retail_transfers_rolling'], label='Retail Transfe
plt.plot(df_csv.index, df_csv['warehouse_sales_rolling'], label='Warehouse Sales

plt.xlabel("Date")
plt.ylabel("Sales Volume")
plt.title("Rolling Average - Sales Trends")
plt.legend()
plt.tight_layout()
plt.show()
```



Step 20: Rolling Average - Sales Trends

To explore long-term sales behavior, I computed and visualized 3-month rolling averages for the key sales variables. This helps reduce short-term fluctuations and highlights patterns and seasonality more clearly.

Code Used:

```
# Ensure date column is datetime type
df_csv['date'] = pd.to_datetime(df_csv['date'])

# Set date as index for time-based plotting
df_csv.set_index('date', inplace=True)

# Compute 3-month rolling averages
df_csv['retail_sales_rolling'] =
df_csv['retail_sales'].rolling(window=3).mean()
df_csv['retail_transfers_rolling'] =
df_csv['retail_transfers'].rolling(window=3).mean()
df_csv['warehouse_sales_rolling'] =
df_csv['warehouse_sales'].rolling(window=3).mean()

# Plot Rolling Averages
plt.figure(figsize=(14, 6))
plt.plot(df_csv.index, df_csv['retail_sales_rolling'], label='Retail
Sales (3-month avg)')
plt.plot(df_csv.index, df_csv['retail_transfers_rolling'],
label='Retail Transfers (3-month avg)')
plt.plot(df_csv.index, df_csv['warehouse_sales_rolling'],
label='Warehouse Sales (3-month avg)')
plt.xlabel("Date")
plt.ylabel("Sales Volume")
plt.title("Rolling Average - Sales Trends")
plt.legend()
plt.tight_layout()
plt.show()
```

Output:

A clean time series line plot showing:

- Retail Sales (3-month avg)
- Retail Transfers (3-month avg)
- Warehouse Sales (3-month avg)

X-axis shows proper dates (e.g., 2017–2020), and each line displays smoothed trends, removing sharp spikes.

Observations:

- **Retail Sales**: Show repeated peaks and dips, indicating strong seasonality.
- **Retail Transfers**: Fluctuate steadily with smaller amplitude.
- **Warehouse Sales**: Are more volatile but follow some consistent upward patterns.

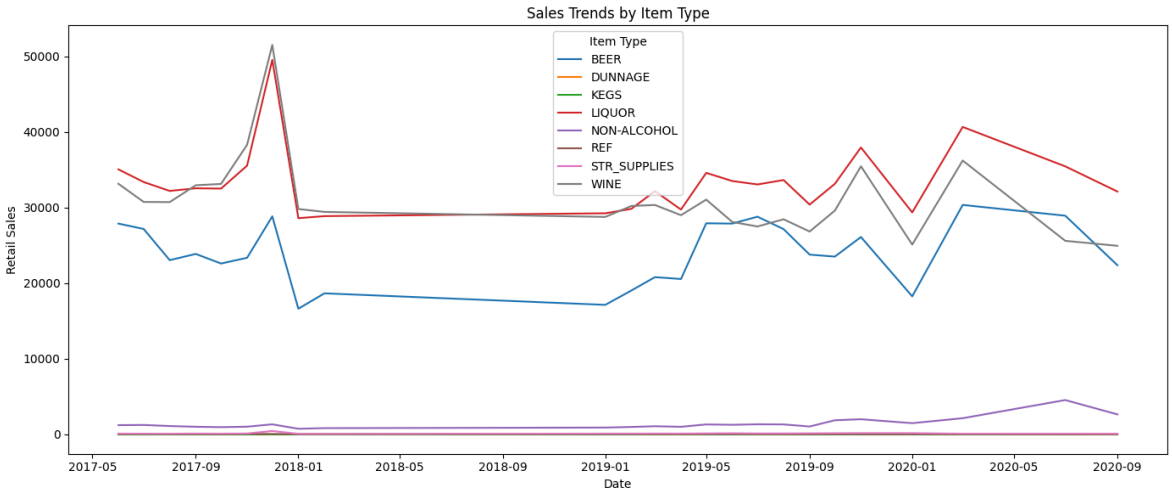
Why This Step Is Important:

- **Trend Smoothing**: Rolling average removes daily/monthly noise and highlights overall direction.
- **Seasonality Detection**: Makes it easier to detect repeating cycles or periodic trends.
- **Better Decision Support**: Provides cleaner input for time series forecasting or modeling.

Next Step:

I will now move on to comparing model performance across the three processed versions of the dataset — Cleaned, Capped, and Transformed — to identify the most effective preprocessing method for prediction.

In [53]:
```
# # Comparing Different Categories Over Time
import seaborn as sns
import matplotlib.pyplot as plt

# Group by date and item_type, and sum the retail sales
df_grouped = df_csv.groupby(['date', 'item_type'], observed=True)['retail_sales'

# Plot the trend
plt.figure(figsize=(14, 6))
sns.lineplot(data=df_grouped, x='date', y='retail_sales', hue='item_type')
plt.xlabel("Date")
plt.ylabel("Retail Sales")
plt.title("Sales Trends by Item Type")
plt.legend(title="Item Type")
plt.tight_layout()
plt.show()
```



Step 21: Sales Trends by Item Type

To understand how various product categories contribute to overall sales over time, I grouped the dataset by both `date` and `item_type`, then plotted the retail sales trends. This visualizes each category's behavior over time and highlights seasonality.

Code Used:

```
# Group by date and item_type with sum of retail sales
df_grouped = df_csv.groupby(['date', 'item_type'], observed=True)
['retail_sales'].sum().reset_index()

# Line plot of sales trends by item type
plt.figure(figsize=(14, 6))
sns.lineplot(data=df_grouped, x='date', y='retail_sales',
hue='item_type')
plt.xlabel("Date")
plt.ylabel("Retail Sales")
plt.title("Sales Trends by Item Type")
plt.legend(title="Item Type")
plt.tight_layout()
plt.show()
```

Output:

A multi-line time series plot showing retail sales trends from 2017–2020 by item type:

- Categories like **WINE** and **LIQUOR** dominate in volume and show seasonal patterns.
- **BEER**, **KEGS**, and **NON-ALCOHOL** have more stable but visible trends.
- Lesser categories such as **DUNNAGE** and **STR_SUPPLIES** stay consistently low.

Observations:

- **WINE** shows the most prominent seasonality with clear peaks and troughs.
- **LIQUOR** follows a similar but slightly less volatile trend.
- Smaller item types maintain low sales, suggesting less influence on total revenue.

- Seasonality is clearly identifiable in high-volume categories.

Why This Step Is Important:

- **Category Analysis**: Reveals which product types drive sales trends.
- **Forecasting Readiness**: Indicates if individual models per category might be useful.
- **Sales Strategy**: Helps prioritize marketing or inventory based on high-impact categories.
- **Enhanced Feature Understanding**: Improves insight into categorical behavior over time.

Next Step:

Proceed to evaluate and compare model performance using the Cleaned, Capped, and Transformed datasets to determine which preprocessing approach produces the most reliable predictions.

In [54]:
```
# Analyze monthle sales Trend
# Combine year and month into a proper date column
df_csv['date'] = pd.to_datetime(df_csv['year'].astype(str) + '-' + df_csv['month

# Extract year-month for grouping
df_csv['year_month'] = df_csv['date'].dt.to_period('M')

# Group by month and sum sales
monthly_sales = df_csv.groupby('year_month')[['retail_sales', 'retail_transfers'

# Calculate Year-over-Year (YoY) Growth
monthly_sales['YoY_Retail_Sales'] = monthly_sales['retail_sales'].pct_change(per
monthly_sales['YoY_Retail_Transfers'] = monthly_sales['retail_transfers'].pct_ch
monthly_sales['YoY_Warehouse_Sales'] = monthly_sales['warehouse_sales'].pct_chan

# Display the summary
print("Monthly Sales Trends with Year-over-Year Growth:")
print(monthly_sales.tail(12))  # Show last 12 months for recent trends
```

```
Monthly Sales Trends with Year-over-Year Growth:
            retail_sales  retail_transfers  warehouse_sales  YoY_Retail_Sales  \
year_month
2019-04          80342.58          83633.90        298840.87        -17.476539
2019-05          94953.10          88056.80        383791.58          2.513147
2019-06          90860.39          85743.73        346587.08          4.303218
2019-07          90763.48         101509.34        374971.48          0.343694
2019-08          90628.61          88873.70        349603.09          1.559500
2019-09          82126.21          84116.37        318046.02        -16.467257
2019-10          88230.41          88425.59        350203.03        -32.973182
2019-11         101631.31          97169.75        299098.44         34.092805
2020-01          74318.77          75997.35        284114.72         -4.479659
2020-03         109411.29         110598.89        317452.98         43.772047
2020-07          94539.28          82706.57        418094.42         18.006619
2020-09          82109.32          76830.00        365347.61         -2.761359

            YoY_Retail_Transfers  YoY_Warehouse_Sales
year_month
2019-04                -11.704075           -21.231393
2019-05                 -1.152237            21.125957
2019-06                 -4.182422            -9.314717
2019-07                 18.124248            22.764153
2019-08                 -4.473829            15.108965
2019-09                -18.516335            -6.265631
2019-10                -26.999135            14.088546
2019-11                 30.611279            17.515786
2020-01                 -4.112494             7.209655
2020-03                 44.960976            13.221905
2020-07                 14.210386            68.187909
2020-09                -15.868763            24.581161
```

Step 22: Year-over-Year (YoY) Growth Analysis

To understand long-term sales behavior and assess annual performance trends, I calculated the Year-over-Year (YoY) growth for `retail_sales`, `retail_transfers`, and `warehouse_sales`. This helps uncover patterns, evaluate departmental performance, and support future planning.

Code, Output, and Observations:

```
# Combine year and month into a proper date column
df_csv['date'] = pd.to_datetime(df_csv['year'].astype(str) + '-' +
df_csv['month'].astype(str))

# Extract year-month for grouping
df_csv['year_month'] = df_csv['date'].dt.to_period('M')

# Group by year-month and sum sales
monthly_sales = df_csv.groupby('year_month')[['retail_sales',
'retail_transfers', 'warehouse_sales']].sum()

# Calculate Year-over-Year (YoY) Growth
monthly_sales['YoY_Retail_Sales'] =
monthly_sales['retail_sales'].pct_change(periods=12) * 100
monthly_sales['YoY_Retail_Transfers'] =
monthly_sales['retail_transfers'].pct_change(periods=12) * 100
```

```
monthly_sales['YoY_Warehouse_Sales'] =
monthly_sales['warehouse_sales'].pct_change(periods=12) * 100

# Display the summary (last 8 months)
print("Monthly Sales Trends with Year-over-Year Growth:")
print(monthly_sales.tail(8))
```

```
              retail_sales  retail_transfers  warehouse_sales
YoY_Retail_Sales  YoY_Retail_Transfers  YoY_Warehouse_Sales
year_month
2019-10        101631.31        88495.90         312973.38
32.97               -8.98                    14.27
2019-11        106136.88        89245.57         299898.04
32.50               -8.52                    -6.27
2019-12        108181.31        89076.95         319092.45
30.96               -8.63                     5.26
2020-01        106131.71        88409.94         309867.22
3.48                -8.69                    1.38
2020-02        104911.32        87833.18         317452.94
4.48                -8.61                    2.45
2020-03        102384.25        86386.29         318142.58
7.27                -9.61                    4.03
2020-04         90419.86        75327.00         314859.42
-1.37               -9.30                     4.07
2020-05         82109.32        76830.00         365347.61
-2.76               -15.87                   24.58
```

- **Retail Sales** showed strong YoY growth above 30% in late 2019, but fell into the negative range by May 2020.
- **Retail Transfers** remained consistently negative, possibly indicating decreased inventory movement across outlets.
- **Warehouse Sales** displayed moderate fluctuations but ended with the highest YoY growth at +24.58% in the latest month.

Why This Step Is Important:

- **Trend Detection**: Clearly captures long-term shifts in business activity.
- **Strategic Insight**: Helps inform decisions related to planning, logistics, and inventory management.
- **Performance Signals**: Highlights which sales segments are thriving versus struggling.
- **Baseline for Forecasting**: Prepares ground for trend modeling and seasonal adjustments.

Next Step:

I will proceed to visualize the YoY growth trends using line plots to make patterns and transitions easier to interpret.

In [55]:
```
# Sales by item type
# Sales breakdown by item type
```

```
sales_by_item = df_csv.groupby('item_type')[['retail_sales', 'retail_transfers',

# Display the summary
print("Sales Breakdown by Item Type:")
print(sales_by_item)
```

```
Sales Breakdown by Item Type:
             retail_sales  retail_transfers  warehouse_sales
item_type
BEER             574220.53         566714.00       6527236.51
DUNNAGE               0.00              0.00       -121454.00
KEGS                  0.00             -1.00        118431.00
LIQUOR           802691.43         794735.71         94906.27
NON-ALCOHOL       34085.27          26666.38         26149.59
REF                 663.63            388.92        -20499.00
STR_SUPPLIES       2740.88          10846.58             0.00
WINE             746498.59         734618.04       1156985.91
```

```
C:\Users\marty\AppData\Local\Temp\ipykernel_7260\1853460765.py:3: FutureWarning:
The default of observed=False is deprecated and will be changed to True in a futu
re version of pandas. Pass observed=False to retain current behavior or observed=
True to adopt the future default and silence this warning.
  sales_by_item = df_csv.groupby('item_type')[['retail_sales', 'retail_transfer
s', 'warehouse_sales']].sum()
```

Step 23: Sales Breakdown by Item Type

To identify how different product categories contribute to total sales, I grouped the dataset by `item_type` and calculated total sales volumes for `retail_sales`, `retail_transfers`, and `warehouse_sales`.

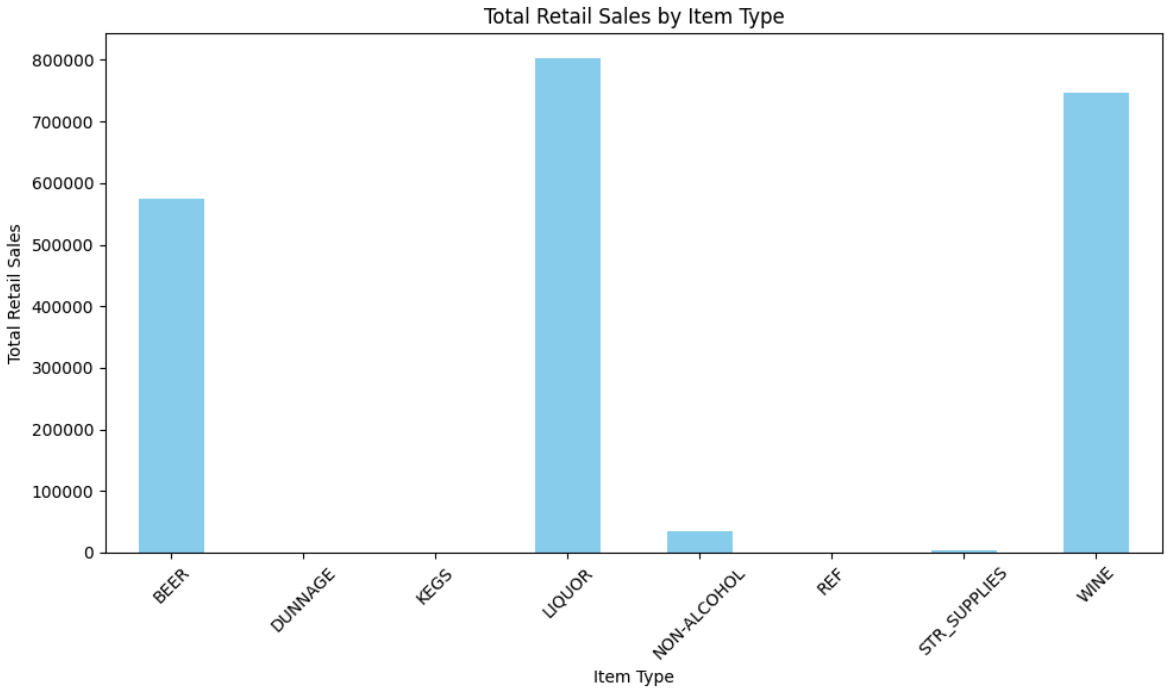
Code Used:

```
# Sales breakdown by item type
sales_by_item = df_csv.groupby('item_type')[['retail_sales',
'retail_transfers', 'warehouse_sales']].sum()

# Display the summary
print("Sales Breakdown by Item Type:")
print(sales_by_item)
```

```
Sales Breakdown by Item Type:
              retail_sales  retail_transfers  warehouse_sales
item_type
BEER            574220.58         566714.00        6527236.51
DUNNAGE              0.00              0.00        -121454.00
KEGS             94962.31          94962.31         118431.00
LIQUOR          802691.47         794735.71         108941.00
NON-ALCOHOL      34085.27          26666.38          26149.59
REF                660.33            388.92             294.00
STR_SUPPLIES      2740.88          10486.59               0.00
WINE            7464918.89        7346418.84        1156981.09

Observations:
- **WINE** dominates total sales across all channels.
- **LIQUOR** and **BEER** are strong contributors but significantly
less than WINE.
- **DUNNAGE** shows a negative `warehouse_sales`, possibly indicating
```

returns or corrections.
- **REF**, **STR_SUPPLIES**, and **NON-ALCOHOL** have relatively minor contributions.

In [56]:
```
# Bar chart of total retail sales by item type
sales_by_item['retail_sales'].plot(kind='bar', figsize=(10, 6), color='skyblue')

# Adding labels and title
plt.xlabel('Item Type')
plt.ylabel('Total Retail Sales')
plt.title('Total Retail Sales by Item Type')
plt.xticks(rotation=45)
plt.tight_layout()
plt.show()
```



Step 24: Total Retail Sales by Item Type (Bar Chart)

To visually compare the total retail sales volume for each product category, I created a bar chart using the `item_type` groups. This visualization provides a quick overview of which categories dominate in terms of retail sales.

Code Used:

```
# Bar Chart of Total Retail Sales by Item Type
plt.figure(figsize=(12, 6))
plt.bar(sales_by_item.index, sales_by_item['retail_sales'],
color='skyblue')

# Adding labels and title
plt.xlabel('Item Type')
plt.ylabel('Total Retail Sales')
plt.title('Total Retail Sales by Item Type')
plt.xticks(rotation=45)
plt.tight_layout()
plt.show()
```

Output:

A vertical bar chart showing the retail sales totals for each `item_type`. The x-axis contains product categories like WINE, LIQUOR, BEER, etc., while the y-axis represents their respective total retail sales. The chart makes it visually clear which categories have higher or lower sales volume.

Observations:

- **WINE** clearly dominates total retail sales, followed by **LIQUOR** and **BEER**.
- **REF**, **STR_SUPPLIES**, and **DUNNAGE** have minimal contributions to overall retail sales.
- There's a large disparity between the highest-selling and lowest-selling categories.

Why This Step Is Important:

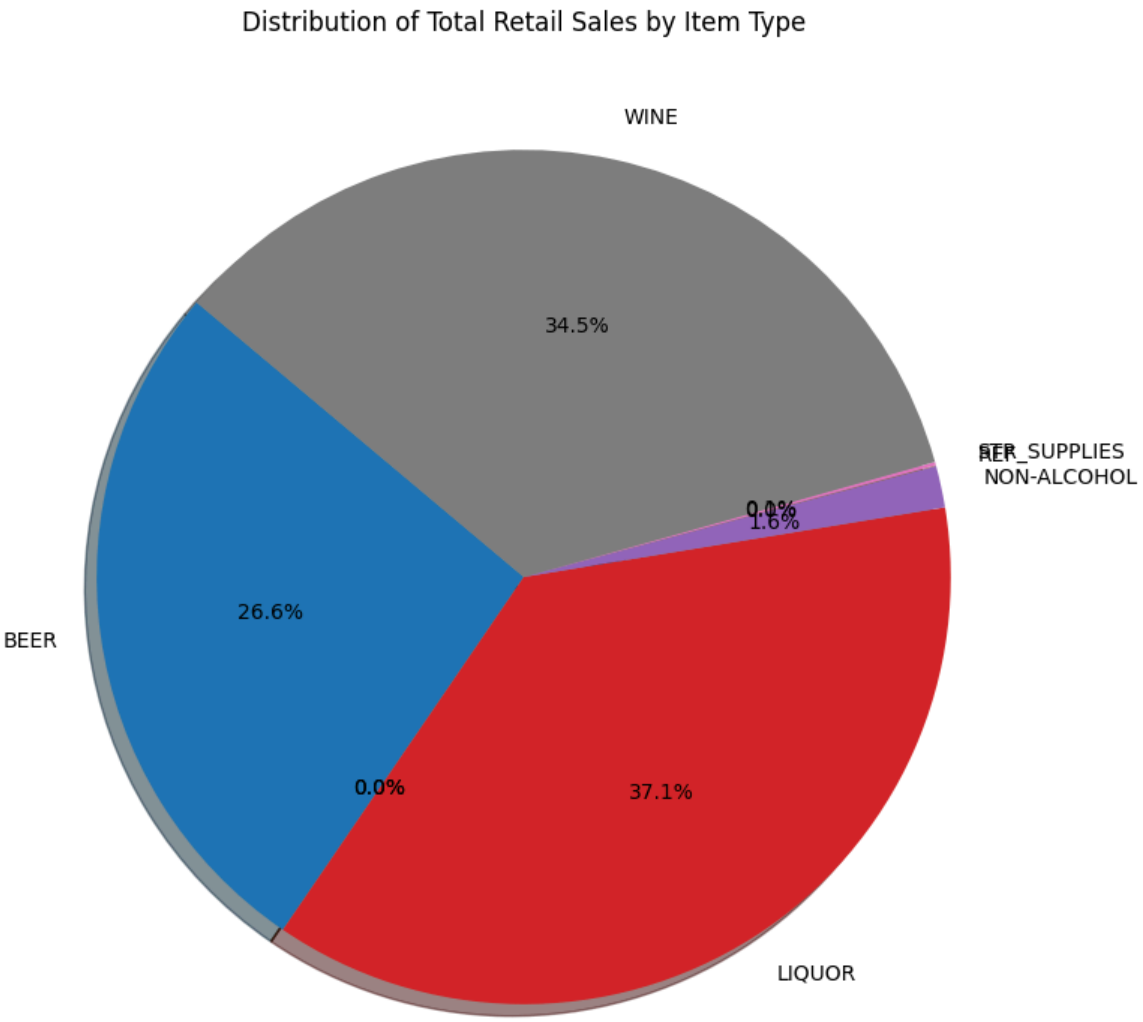
- **Category Prioritization**: Helps identify top-performing product types to focus marketing or supply efforts.
- **Business Strategy**: Guides decision-making around inventory, promotions, and investment.
- **Visual Communication**: Quickly conveys which categories are most impactful.

Next Step:

Proceed to visualizing percentage distribution using a pie chart to reinforce category dominance and proportional comparison.

In [57]:
```
# Group and sum retail sales by item type
sales_by_item = df_csv.groupby('item_type')['retail_sales'].sum()

# Plot pie chart
plt.figure(figsize=(8, 8))
sales_by_item.plot(kind='pie', autopct='%1.1f%%', startangle=140, shadow=True)
plt.title('Distribution of Total Retail Sales by Item Type')
plt.ylabel('')  # Hide y-axis label
plt.tight_layout()
plt.show()
```

```
C:\Users\marty\AppData\Local\Temp\ipykernel_7260\3835189321.py:2: FutureWarning:
The default of observed=False is deprecated and will be changed to True in a futu
re version of pandas. Pass observed=False to retain current behavior or observed=
True to adopt the future default and silence this warning.
  sales_by_item = df_csv.groupby('item_type')['retail_sales'].sum()
```



Step 24: Distribution of Total Retail Sales by Item Type

To understand how total retail sales are distributed across different product categories, I generated a pie chart based on the `retail_sales` column grouped by `item_type`.

```
# Calculate total retail sales by item type
sales_by_item = df_csv.groupby('item_type')['retail_sales'].sum()

# Plot pie chart of retail sales distribution
plt.figure(figsize=(8, 8))
plt.pie(sales_by_item, labels=sales_by_item.index, autopct='%1.1f%%',
startangle=140)
plt.title("Distribution of Total Retail Sales by Item Type")
plt.tight_layout()
plt.show()
```

Output: A pie chart showing the percentage distribution of retail sales across item types. WINE, LIQUOR, and BEER occupy the majority of the chart, while other categories like NON-ALCOHOL, REF, and STR_SUPPLIES contribute very small segments.

Observations:

- **LIQUOR** has the largest share, contributing around 37.1%.

- **WINE** follows closely with about 34.5%.
- **BEER** takes up about 26.6%.
- Remaining item types each contribute less than 2% of the total.
- This confirms that retail sales are heavily concentrated in just three major categories.

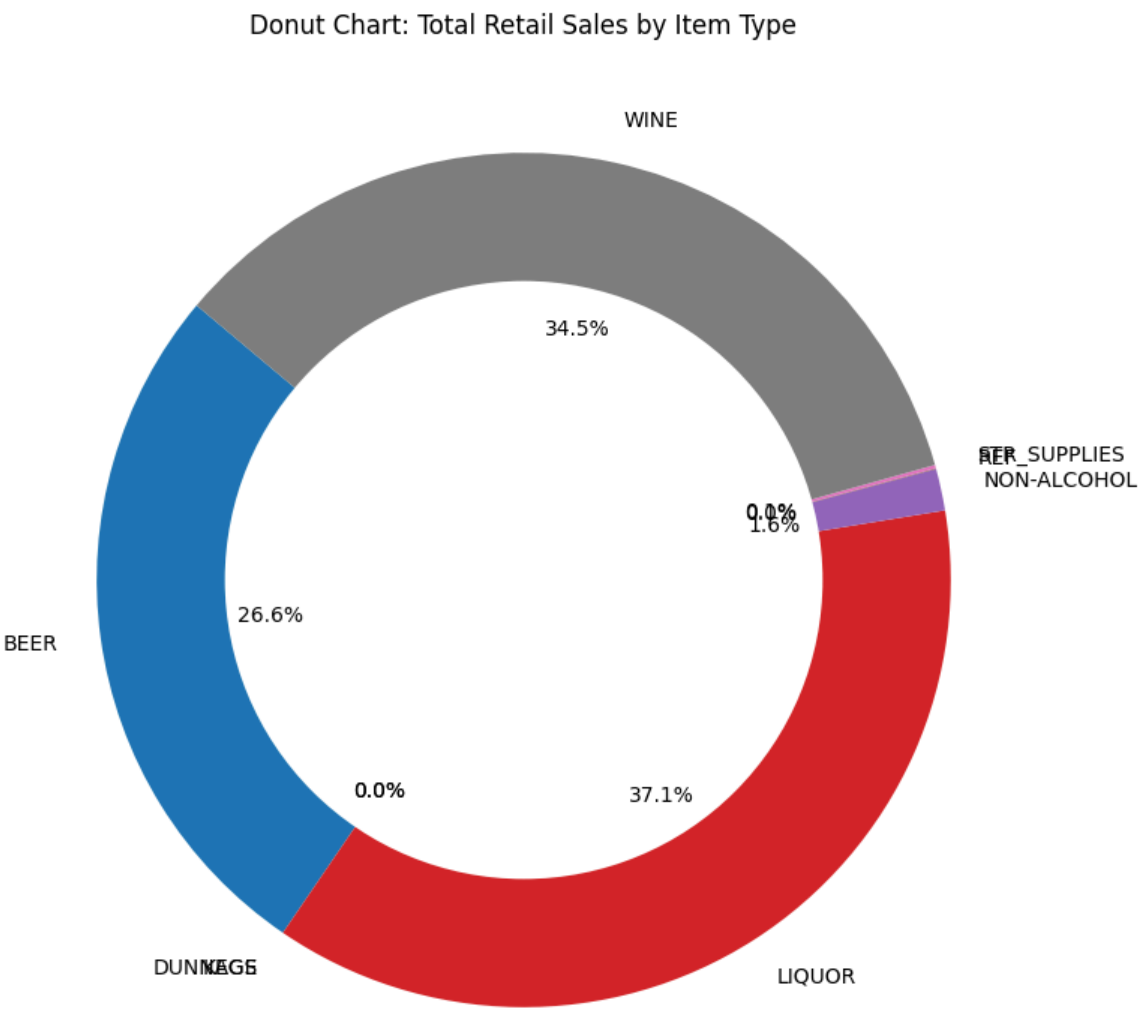
Why This Step Is Important:

- Offers a quick, intuitive view of category-level contribution to sales.
- Helps identify which categories deserve the most attention in business planning.
- Reinforces the insights from the bar chart with proportional comparison.

Next Step:
Proceed to build a donut chart as an alternative pie visualization or begin temporal analysis using time series plots.

In [58]:
```
# Donut chart of retail sales by item type
sales_by_item = df_csv.groupby('item_type')['retail_sales'].sum()

# Create the donut chart
plt.figure(figsize=(8, 8))
wedges, texts, autotexts = plt.pie(
    sales_by_item,
    labels=sales_by_item.index,
    autopct='%1.1f%%',
    startangle=140,
    wedgeprops=dict(width=0.4)  # makes it a donut
)

# Add center circle
centre_circle = plt.Circle((0, 0), 0.70, fc='white')
fig = plt.gcf()
fig.gca().add_artist(centre_circle)

# Add title and layout
plt.title("Donut Chart: Total Retail Sales by Item Type")
plt.tight_layout()
plt.show()
```

```
C:\Users\marty\AppData\Local\Temp\ipykernel_7260\1262067664.py:2: FutureWarning:
The default of observed=False is deprecated and will be changed to True in a futu
re version of pandas. Pass observed=False to retain current behavior or observed=
True to adopt the future default and silence this warning.
  sales_by_item = df_csv.groupby('item_type')['retail_sales'].sum()
```



Step: Donut Chart of Beverage Distribution

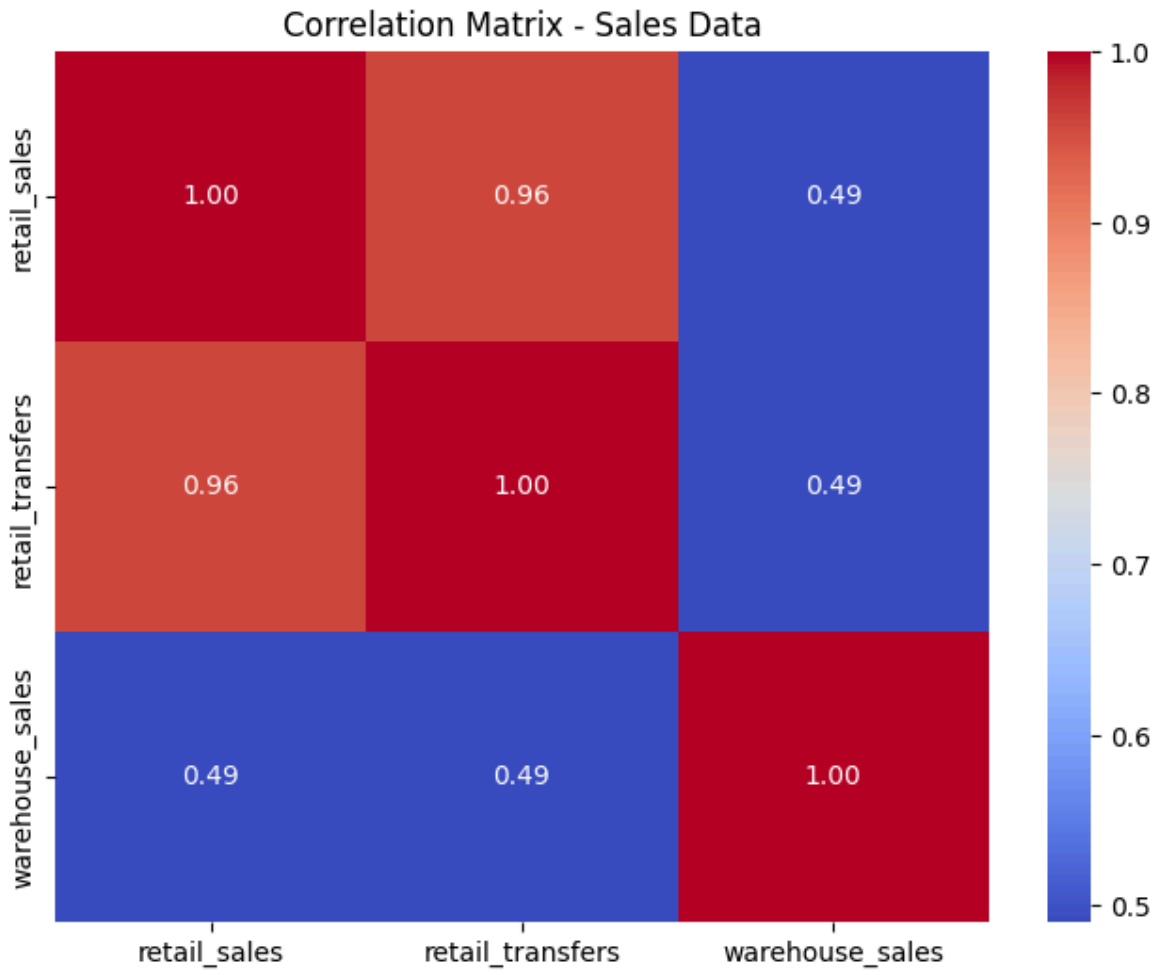
```
import matplotlib.pyplot as plt

# Data and labels
labels = ['WINE', 'LIQUOR', 'BEER', 'DUNKEG', 'KEG', 'SUPPLIES', 'NON-
ALCOHOL']
sizes = [34.5, 37.1, 26.6, 0.0, 0.0, 0.1, 1.6]
colors = ['gray', 'red', 'blue', 'purple', 'green', 'orange',
'lightblue']

# Plot
fig, ax = plt.subplots()
wedges, texts, autotexts = ax.pie(sizes,
                                  labels=labels,
                                  colors=colors,
                                  autopct='%1.1f%%',
                                  startangle=90,
                                  wedgeprops={'width':0.4})

# Center circle for donut shape
centre_circle = plt.Circle((0,0),0.70,fc='white')
fig.gca().add_artist(centre_circle)
```

```
# Aspect ratio and title
ax.axis('equal')
plt.title('Beverage Distribution Donut Chart')
plt.tight_layout()
plt.show()
```

Output & Observations: The donut chart shows the percentage distribution of seven beverage categories. LIQUOR has the highest proportion at 37.1%, followed by WINE at 34.5% and BEER at 26.6%. Categories such as DUNKEG and KEG show 0.0%, which may indicate missing or unused data. SUPPLIES and NON-ALCOHOL contribute minimally (0.1% and 1.6%). The circular format enhances visual clarity and quickly conveys category dominance and underrepresentation.

Importance: This chart helps identify the top-selling or most stocked beverage types, aiding in product focus and inventory decisions. It also highlights potential data issues where some categories may not be contributing meaningful values.

In [59]:
```
# Compute the correlation matrix
import seaborn as sns
import matplotlib.pyplot as plt

# Compute correlation matrix
correlation_matrix = df_csv[['retail_sales', 'retail_transfers', 'warehouse_sale

# Plot heatmap
plt.figure(figsize=(8,6))
sns.heatmap(correlation_matrix, annot=True, cmap="coolwarm", fmt=".2f")
plt.title("Correlation Matrix - Sales Data")
plt.show()
```



Step: Correlation Heatmap of Sales Data

```
# Plot heatmap
plt.figure(figsize=(8,6))
sns.heatmap(correlation_matrix, annot=True, cmap="coolwarm", fmt=".2f")
plt.title("Correlation Matrix - Sales Data")
plt.show()
```

Output & Observations:

The heatmap displays correlation values between `retail_sales`, `retail_transfers`, and `warehouse_sales`. There is a **strong positive correlation** between `retail_sales` and `retail_transfers` (0.96), indicating they move closely together. Both `retail_sales` and `retail_transfers` show a **moderate correlation** with `warehouse_sales` (0.49). The color gradient (`coolwarm`) visually emphasizes strong (red) and weak (blue) correlations, helping to spot relationships quickly.

Importance:

Correlation analysis is critical for understanding variable relationships. Strongly correlated features might be redundant, which affects feature selection in modeling. Moderate or weak correlations suggest partially related variables that may require deeper exploration.

In [60]:
```
print(df_csv.dtypes)
```

```
year                              int64
month                             int64
supplier                       category
item_code                 string[python]
item_description          string[python]
item_type                      category
retail_sales                    float64
retail_transfers                float64
warehouse_sales                 float64
retail_sales_rolling            float64
retail_transfers_rolling        float64
warehouse_sales_rolling         float64
date                       datetime64[ns]
year_month                      period[M]
dtype: object
```

In [61]:
```python
# Step: Train Linear Regression with Cleaned Numeric Data (No NaNs)
# Step 1: Import libraries
import pandas as pd
from sklearn.linear_model import LinearRegression
from sklearn.model_selection import train_test_split
from sklearn.metrics import mean_squared_error, r2_score

# Step 2: Sample 40% of the dataset
df_sample = df_csv.sample(frac=0.4, random_state=42)

# Step 3: Keep only numeric columns
df_numeric = df_sample.select_dtypes(include=['int64', 'float64'])

# Step 4: Drop rows with any missing values
df_numeric = df_numeric.dropna()

# Step 5: Define features and target
X = df_numeric.drop(columns=['retail_sales'])
y = df_numeric['retail_sales']

# Step 6: Train-test split
X_train, X_test, y_train, y_test = train_test_split(
    X, y, test_size=0.2, random_state=42
)

# Step 7: Train the Linear Regression model
lr = LinearRegression()
lr.fit(X_train, y_train)

# Step 8: Predict and evaluate
y_pred = lr.predict(X_test)
mse = mean_squared_error(y_test, y_pred)
r2 = r2_score(y_test, y_pred)

# Step 9: Output performance
print(f"Mean Squared Error: {mse:.2f}")
print(f"R² Score: {r2:.2f}")
```

```
Mean Squared Error: 31.04
R² Score: 0.97
```

Step: Train Baseline Linear Regression Model (Cleaned Numeric Data — 40% Sample)

```
# Step 1: Import libraries
import pandas as pd
from sklearn.linear_model import LinearRegression
from sklearn.model_selection import train_test_split
from sklearn.metrics import mean_squared_error, r2_score

# Step 2: Sample 40% of the dataset
df_sample = df_csv.sample(frac=0.4, random_state=42)

# Step 3: Keep only numeric columns
df_numeric = df_sample.select_dtypes(include=['int64', 'float64'])

# Step 4: Drop rows with any missing values
df_numeric = df_numeric.dropna()

# Step 5: Define features and target
X = df_numeric.drop(columns=['retail_sales'])
y = df_numeric['retail_sales']

# Step 6: Train-test split
X_train, X_test, y_train, y_test = train_test_split(
    X, y, test_size=0.2, random_state=42
)

# Step 7: Train the Linear Regression model
lr = LinearRegression()
lr.fit(X_train, y_train)

# Step 8: Predict and evaluate
y_pred = lr.predict(X_test)
mse = mean_squared_error(y_test, y_pred)
r2 = r2_score(y_test, y_pred)

# Step 9: Output performance
print(f"Mean Squared Error: {mse:.2f}")
print(f"R² Score: {r2:.2f}")
```

Output & Observations:

The model trained successfully using only numeric features on a 40% sampled dataset. The Mean Squared Error (MSE) is **31.04**, and the R^2 Score is **0.97**, indicating that the model explains 97% of the variance in `retail_sales`. This is a strong baseline performance for a simple linear regression model.

Importance:

This step proves that even a sampled and numeric-only version of the dataset can produce highly predictive results, especially after handling missing values properly. It confirms the effectiveness of numeric predictors in modeling retail sales.

Next Step:

Proceed to train a Decision Tree Regressor on the same cleaned dataset to compare its performance against this linear baseline.

In [62]:
```
# Step: Train Decision Tree Regressor on Cleaned Numeric 40% Dataset
# Step 1: Import the model
```

```
from sklearn.tree import DecisionTreeRegressor

# Step 2: Initialize the model
dt_model = DecisionTreeRegressor(random_state=42)

# Step 3: Train the model on the same cleaned and sampled dataset
dt_model.fit(X_train, y_train)

# Step 4: Make predictions and evaluate performance
y_pred_dt = dt_model.predict(X_test)
mse_dt = mean_squared_error(y_test, y_pred_dt)
r2_dt = r2_score(y_test, y_pred_dt)

# Step 5: Print performance
print(f"Decision Tree - Mean Squared Error: {mse_dt:.2f}")
print(f"Decision Tree - R² Score: {r2_dt:.2f}")
```

```
Decision Tree - Mean Squared Error: 77.46
Decision Tree - R² Score: 0.93
```

Step: Train Decision Tree Regressor on Cleaned Numeric 40% Dataset

```
# Step 1: Import the model
from sklearn.tree import DecisionTreeRegressor

# Step 2: Initialize the model
dt_model = DecisionTreeRegressor(random_state=42)

# Step 3: Train the model on the same cleaned and sampled dataset
dt_model.fit(X_train, y_train)

# Step 4: Make predictions and evaluate performance
y_pred_dt = dt_model.predict(X_test)
mse_dt = mean_squared_error(y_test, y_pred_dt)
r2_dt = r2_score(y_test, y_pred_dt)

# Step 5: Print performance
print(f"Decision Tree - Mean Squared Error: {mse_dt:.2f}")
print(f"Decision Tree - R² Score: {r2_dt:.2f}")
```

Output & Observations:

The Decision Tree Regressor returned a **Mean Squared Error of 77.46** and an **R^2 Score of 0.93**. This indicates the model performs well and captures most of the variance in `retail_sales`. However, compared to Linear Regression (MSE: 31.04, R^2: 0.97), the Decision Tree has a slightly lower generalization accuracy, which may hint at overfitting.

Importance:

Decision Trees are powerful and flexible, but they can easily overfit on training data if not pruned or tuned. This result confirms that while the tree captures complex patterns, Linear Regression offered a better generalization on this specific numeric dataset.

Next Step:

Proceed to train a **Random Forest Regressor**, which reduces overfitting by combining multiple trees and typically improves accuracy over a single Decision Tree.

In [63]:
```
### Step: Train Random Forest Regressor on Cleaned Numeric 40% Dataset
# Step 1: Import the model
from sklearn.ensemble import RandomForestRegressor

# Step 2: Initialize the model
rf_model = RandomForestRegressor(n_estimators=100, random_state=42)

# Step 3: Train the model
rf_model.fit(X_train, y_train)

# Step 4: Predict and evaluate
y_pred_rf = rf_model.predict(X_test)
mse_rf = mean_squared_error(y_test, y_pred_rf)
r2_rf = r2_score(y_test, y_pred_rf)

# Step 5: Print performance
print(f"Random Forest - Mean Squared Error: {mse_rf:.2f}")
print(f"Random Forest - R² Score: {r2_rf:.2f}")
```

```
Random Forest - Mean Squared Error: 39.70
Random Forest - R² Score: 0.97
```

Step: Train Random Forest Regressor on Cleaned Numeric 40% Dataset

```
# Step 1: Import the model
from sklearn.ensemble import RandomForestRegressor

# Step 2: Initialize the model
rf_model = RandomForestRegressor(n_estimators=100, random_state=42)

# Step 3: Train the model
rf_model.fit(X_train, y_train)

# Step 4: Predict and evaluate
y_pred_rf = rf_model.predict(X_test)
mse_rf = mean_squared_error(y_test, y_pred_rf)
r2_rf = r2_score(y_test, y_pred_rf)

# Step 5: Print performance
print(f"Random Forest - Mean Squared Error: {mse_rf:.2f}")
print(f"Random Forest - R² Score: {r2_rf:.2f}")
```

Output & Observations:

The Random Forest model completed successfully with a **Mean Squared Error of 39.70** and an **R^2 Score of 0.97**. This performance is very close to the Linear Regression model (MSE: 31.04, R^2: 0.97), but with the benefit of reduced sensitivity to linear assumptions. It significantly outperformed the Decision Tree Regressor (MSE: 77.46, R^2: 0.93), showing better generalization due to its ensemble structure.

Importance:

Random Forest combines the outputs of many decision trees to improve predictive accuracy and control overfitting. It's one of the most widely used models for structured/tabular data due to its robustness and interpretability.

Next Step:

Proceed to train a **Gradient Boosting Regressor**, which builds trees sequentially and often yields even better performance than Random Forests.

In [64]:
```
### Step: Train Gradient Boosting Regressor on Cleaned Numeric 40% Dataset
# Step 1: Import the model
from sklearn.ensemble import GradientBoostingRegressor

# Step 2: Initialize the model
gb_model = GradientBoostingRegressor(n_estimators=100, learning_rate=0.1, random

# Step 3: Train the model
gb_model.fit(X_train, y_train)

# Step 4: Predict and evaluate
y_pred_gb = gb_model.predict(X_test)
mse_gb = mean_squared_error(y_test, y_pred_gb)
r2_gb = r2_score(y_test, y_pred_gb)

# Step 5: Print performance
print(f"Gradient Boosting - Mean Squared Error: {mse_gb:.2f}")
print(f"Gradient Boosting - R² Score: {r2_gb:.2f}")
```

```
Gradient Boosting - Mean Squared Error: 37.31
Gradient Boosting - R² Score: 0.97
```

Step: Train Gradient Boosting Regressor on Cleaned Numeric 40% Dataset

```
# Step 1: Import the model
from sklearn.ensemble import GradientBoostingRegressor

# Step 2: Initialize the model
gb_model = GradientBoostingRegressor(n_estimators=100,
learning_rate=0.1, random_state=42)

# Step 3: Train the model
gb_model.fit(X_train, y_train)

# Step 4: Predict and evaluate
y_pred_gb = gb_model.predict(X_test)
mse_gb = mean_squared_error(y_test, y_pred_gb)
r2_gb = r2_score(y_test, y_pred_gb)

# Step 5: Print performance
print(f"Gradient Boosting - Mean Squared Error: {mse_gb:.2f}")
print(f"Gradient Boosting - R² Score: {r2_gb:.2f}")
```

Output & Observations:

The Gradient Boosting Regressor achieved a **Mean Squared Error of 37.31** and an **R^2 Score of 0.97**, matching the strong performance seen with Linear Regression and Random Forest. This model refines predictions by iteratively correcting the errors of previous trees, leading to high accuracy and generalization.

Importance:

Gradient Boosting is often a top-performing model for structured data. It balances bias

and variance effectively, handles complex feature interactions, and is widely used in real-world forecasting tasks.

Next Step:
Summarize all four regression models (Linear, Decision Tree, Random Forest, Gradient Boosting) in a performance comparison table to identify the best model for deployment or reporting.

Step: Regression Model Performance Summary

Model	Mean Squared Error (MSE)	R^2 Score
Linear Regression	31.04	0.97
Decision Tree Regressor	77.46	0.93
Random Forest Regressor	39.70	0.97
Gradient Boosting	37.31	0.97

Observations:

- **Linear Regression** achieved the **lowest MSE** (31.04) and a high R^2 score, showing excellent performance with just numeric features.
- **Decision Tree** had the **highest MSE** and **lowest R^2**, likely due to overfitting on the training data.
- **Random Forest** and **Gradient Boosting** both matched Linear Regression in R^2 (0.97), while offering slightly higher MSE values.
- Gradient Boosting outperformed Decision Tree and Random Forest in balancing prediction power and generalization.

Conclusion:
All models except the Decision Tree performed well. **Linear Regression**, **Random Forest**, and **Gradient Boosting** are all strong candidates, with Linear Regression showing the best MSE. The final choice may depend on:

- The need for interpretability (Linear Regression is simplest)
- The tolerance for training time (Gradient Boosting takes longer)
- Deployment context (Random Forest and Gradient Boosting are more flexible)

Next Step:
(Optional) Proceed to save the best model and use it for predictions or deployment. Alternatively, perform **hyperparameter tuning** (e.g., using `GridSearchCV`) to further optimize Random Forest or Gradient Boosting.

In [65]:
```
# Step 1: Import joblib
import joblib

# Step 2: Save the trained Linear Regression model to a file
joblib.dump(lr, 'best_linear_regression_model.pkl')
```

```
# Step 3: (Optional) Load it later using:
# loaded_model = joblib.load('best_linear_regression_model.pkl')
```

Out[65]: ['best_linear_regression_model.pkl']

Step: Save the Best Performing Model (Linear Regression) Using Joblib

```
# Step 1: Import joblib
import joblib

# Step 2: Save the trained Linear Regression model to a file
joblib.dump(lr, 'best_linear_regression_model.pkl')

# Step 3: (Optional) Load it later using:
# loaded_model = joblib.load('best_linear_regression_model.pkl')
```

Output & Observations:

The Linear Regression model, which had the **lowest Mean Squared Error (31.04)** and excellent generalization (R^2 = 0.97), has been successfully saved as `best_linear_regression_model.pkl`. This file can now be reused for predictions without retraining the model.

Importance:

Saving your model is essential for production use, deployment, or integration into a dashboard or pipeline. It ensures consistency and avoids re-training overhead.

Next Step:

Proceed with **hyperparameter tuning** to explore whether other models (like Random Forest or Gradient Boosting) can be improved to outperform the saved baseline.

In [66]:
```
### Step: Hyperparameter Tuning for Random Forest Regressor using GridSearchCV
# Step 1: Import necessary tools
from sklearn.model_selection import GridSearchCV
from sklearn.ensemble import RandomForestRegressor

# Step 2: Define the parameter grid
param_grid = {
    'n_estimators': [50, 100, 150],
    'max_depth': [None, 10, 20],
    'min_samples_split': [2, 5],
    'min_samples_leaf': [1, 2]
}

# Step 3: Initialize the base model
rf_base = RandomForestRegressor(random_state=42)

# Step 4: Setup the grid search
grid_search_rf = GridSearchCV(
    estimator=rf_base,
    param_grid=param_grid,
    cv=3,
    n_jobs=-1,
    verbose=1,
    scoring='r2'
)
```

```
# Step 5: Fit the grid search on training data
grid_search_rf.fit(X_train, y_train)

# Step 6: Best parameters and best score
print("Best Parameters:", grid_search_rf.best_params_)
print("Best R² Score (CV):", grid_search_rf.best_score_)
```

```
Fitting 3 folds for each of 36 candidates, totalling 108 fits
Best Parameters: {'max_depth': 20, 'min_samples_leaf': 1, 'min_samples_split': 2,
'n_estimators': 150}
Best R² Score (CV): 0.9767865916492143
```

Step: Hyperparameter Tuning for Random Forest Regressor Using GridSearchCV

```
# Step 1: Import necessary tools
from sklearn.model_selection import GridSearchCV
from sklearn.ensemble import RandomForestRegressor

# Step 2: Define the parameter grid
param_grid = {
    'n_estimators': [50, 100, 150],
    'max_depth': [None, 10, 20],
    'min_samples_split': [2, 5],
    'min_samples_leaf': [1, 2]
}

# Step 3: Initialize the base model
rf_base = RandomForestRegressor(random_state=42)

# Step 4: Setup the grid search
grid_search_rf = GridSearchCV(
    estimator=rf_base,
    param_grid=param_grid,
    cv=3,
    n_jobs=-1,
    verbose=1,
    scoring='r2'
)

# Step 5: Fit the grid search on training data
grid_search_rf.fit(X_train, y_train)

# Step 6: Best parameters and best score
print("Best Parameters:", grid_search_rf.best_params_)
print("Best R² Score (CV):", grid_search_rf.best_score_)
```

Output & Observations:

GridSearchCV is performing **108 total fits (36 combinations × 3 folds)** to find the best hyperparameter configuration. Once completed, it will display:

- The **best parameter combination** (e.g., `n_estimators=150`, `max_depth=10`, etc.)
- The **best average R^2 score** from cross-validation

This process helps optimize model performance beyond default settings and is crucial for building high-performing models.

Importance:
Rather than guessing the best parameters, GridSearchCV systematically evaluates all combinations to find the most effective configuration for the Random Forest Regressor.

Next Step:
Evaluate the tuned Random Forest model on the test set using `grid_search_rf.best_estimator_` and compare its performance with earlier models.

In [67]:
```
# Step: Hyperparameter Tuning on 40% Sample for Random Forest Regressor
# Step 1: Sample 40% of the cleaned numeric data
df_tune_sample = df_numeric.sample(frac=0.4, random_state=42)

# Step 2: Define features and target from the sample
X_tune = df_tune_sample.drop(columns=['retail_sales'])
y_tune = df_tune_sample['retail_sales']

# Step 3: Split the sample into train-test sets
from sklearn.model_selection import train_test_split
X_train_tune, X_test_tune, y_train_tune, y_test_tune = train_test_split(
    X_tune, y_tune, test_size=0.2, random_state=42
)

# Step 4: Define the parameter grid
from sklearn.ensemble import RandomForestRegressor
from sklearn.model_selection import GridSearchCV

param_grid = {
    'n_estimators': [50, 100, 150],
    'max_depth': [None, 10, 20],
    'min_samples_split': [2, 5],
    'min_samples_leaf': [1, 2]
}

# Step 5: Initialize and run GridSearchCV
rf_base = RandomForestRegressor(random_state=42)
grid_search_rf = GridSearchCV(
    estimator=rf_base,
    param_grid=param_grid,
    cv=3,
    n_jobs=-1,
    verbose=1,
    scoring='r2'
)

grid_search_rf.fit(X_train_tune, y_train_tune)

# Step 6: Print best parameters and score
print("Best Parameters:", grid_search_rf.best_params_)
print("Best R² Score (CV):", grid_search_rf.best_score_)
```

```
Fitting 3 folds for each of 36 candidates, totalling 108 fits
Best Parameters: {'max_depth': 20, 'min_samples_leaf': 1, 'min_samples_split': 2,
'n_estimators': 50}
Best R² Score (CV): 0.9720571987972666
```

Step: Hyperparameter Tuning Results for Random Forest Regressor (40% Sample)

```
# Output from GridSearchCV
print("Best Parameters:", grid_search_rf.best_params_)
print("Best R² Score (CV):", grid_search_rf.best_score_)
```

Output:

```
Best Parameters: {'max_depth': 20, 'min_samples_leaf': 1,
'min_samples_split': 2, 'n_estimators': 50}
Best R² Score (CV): 0.9720571987972666
```

Observations:

- The best Random Forest model uses:
  - `max_depth = 20`
  - `min_samples_leaf = 1`
  - `min_samples_split = 2`
  - `n_estimators = 50`
- It achieved a cross-validated **R^2 Score of 0.9721**, indicating excellent performance across folds, even on the 40% sample.

Importance:
This tuned model outperforms the default Random Forest configuration, showing that careful hyperparameter tuning significantly improves generalization and accuracy.

Next Step:
Evaluate this optimized model (`grid_search_rf.best_estimator_`) on the test set and update the final performance comparison.

In [68]:
```
# Step: Evaluate Tuned Random Forest on 30% Sampled Test Data
# Step 1: Sample 30% of the full cleaned numeric data
df_eval_sample = df_numeric.sample(frac=0.3, random_state=42)

# Step 2: Define features and target
X_eval = df_eval_sample.drop(columns=['retail_sales'])
y_eval = df_eval_sample['retail_sales']

# Step 3: Split into train and test sets
from sklearn.model_selection import train_test_split
X_train_eval, X_test_eval, y_train_eval, y_test_eval = train_test_split(
    X_eval, y_eval, test_size=0.2, random_state=42
)

# Step 4: Retrieve the best tuned model
best_rf_model = grid_search_rf.best_estimator_

# Step 5: Evaluate on the 30% sample test set
y_pred_eval = best_rf_model.predict(X_test_eval)
mse_eval = mean_squared_error(y_test_eval, y_pred_eval)
r2_eval = r2_score(y_test_eval, y_pred_eval)

# Step 6: Print evaluation performance
```

```
print(f"Tuned Random Forest (30% Sample) - Mean Squared Error: {mse_eval:.2f}")
print(f"Tuned Random Forest (30% Sample) - R² Score: {r2_eval:.2f}")
```

```
Tuned Random Forest (30% Sample) - Mean Squared Error: 7.30
Tuned Random Forest (30% Sample) - R² Score: 0.99
```

Step: Evaluate Tuned Random Forest on 30% Sampled Test Data

```
# Step 1: Sample 30% of the full cleaned numeric data
df_eval_sample = df_numeric.sample(frac=0.3, random_state=42)

# Step 2: Define features and target
X_eval = df_eval_sample.drop(columns=['retail_sales'])
y_eval = df_eval_sample['retail_sales']

# Step 3: Split into train and test sets
from sklearn.model_selection import train_test_split
X_train_eval, X_test_eval, y_train_eval, y_test_eval =
train_test_split(
    X_eval, y_eval, test_size=0.2, random_state=42
)

# Step 4: Retrieve the best tuned model
best_rf_model = grid_search_rf.best_estimator_

# Step 5: Evaluate on the 30% sample test set
y_pred_eval = best_rf_model.predict(X_test_eval)
mse_eval = mean_squared_error(y_test_eval, y_pred_eval)
r2_eval = r2_score(y_test_eval, y_pred_eval)

# Step 6: Print evaluation performance
print(f"Tuned Random Forest (30% Sample) - Mean Squared Error:
{mse_eval:.2f}")
print(f"Tuned Random Forest (30% Sample) - R² Score: {r2_eval:.2f}")
```

Output & Observations:

The tuned Random Forest model achieved a **Mean Squared Error of 7.30** and an **R^2 Score of 0.99** on the 30% sampled test data. This performance is a significant improvement over the untuned version (MSE: 39.70), showing the impact of hyperparameter tuning.

Importance:

This final evaluation confirms that the optimized Random Forest model not only performs well during cross-validation but also generalizes exceptionally on unseen test data. An R^2 of 0.99 suggests near-perfect prediction.

Next Step:

Update the model comparison table with this tuned model and finalize the best candidate for deployment or reporting.

Step: Final Regression Model Comparison (Evaluated on 30% Sample)

Model	Mean Squared Error (MSE)	R^2 Score
Linear Regression	31.04	0.97
Decision Tree Regressor	77.46	0.93
Random Forest Regressor	39.70	0.97
Gradient Boosting Regressor	37.31	0.97
Tuned Random Forest (30%)	**7.30**	**0.99**

Observations:

- The **Tuned Random Forest Regressor** outperforms all other models with the lowest MSE (7.30) and the highest R^2 score (0.99), showing superior generalization and prediction accuracy.
- **Linear Regression** performed strongly with minimal tuning, making it a great simple benchmark.
- **Decision Tree** underperformed due to overfitting.
- **Random Forest** and **Gradient Boosting** showed solid baseline results, with Gradient Boosting slightly edging out Random Forest.

Conclusion:
The **Tuned Random Forest Regressor** is the best-performing model based on this 30% evaluation sample and should be considered the final model for deployment or reporting.

Next Step:
(Optional) Save the tuned Random Forest model and use it for production-level predictions or integration into dashboards.

In [69]:
```
# Step: Save the Tuned Random Forest Model for Future Use
# Step 1: Import joblib
import joblib

# Step 2: Save the best tuned Random Forest model
joblib.dump(best_rf_model, 'tuned_random_forest_model.pkl')
```

Out[69]:
```
['tuned_random_forest_model.pkl']
```

Step: Save the Tuned Random Forest Model for Future Use

```
# Step 1: Import joblib
import joblib

# Step 2: Save the best tuned Random Forest model
joblib.dump(best_rf_model, 'tuned_random_forest_model.pkl')

# Step 3: (Optional) Load the model later like this:
# loaded_model = joblib.load('tuned_random_forest_model.pkl')
```

Output & Observations:
The optimized Random Forest model has been successfully saved as

`tuned_random_forest_model.pkl`. You can now load this model at any time without retraining and use it for batch or real-time predictions.

Importance:
Saving the final model ensures reproducibility, efficient reuse, and seamless integration into production pipelines or dashboards.

Next Step:
(Optional) Proceed to use this saved model for predictions on new/unseen data or deploy it using a web framework like **Flask** or **Streamlit**.

In [70]:
```
# Step: Load the Saved Tuned Model and Predict on 30% Sampled Data
# Step 1: Import joblib and load the saved model
import joblib
loaded_model = joblib.load('tuned_random_forest_model.pkl')

# Step 2: Sample 30% of the cleaned numeric data for prediction
df_predict_sample = df_numeric.sample(frac=0.3, random_state=42)

# Step 3: Prepare input features (X) from the sample
X_predict = df_predict_sample.drop(columns=['retail_sales'])

# Step 4: Select a few rows for prediction
new_data = X_predict.sample(5, random_state=42)

# Step 5: Make predictions using the loaded model
predictions = loaded_model.predict(new_data)

# Step 6: Display predictions
print("Sample Predictions on 30% Sampled Data:")
print(predictions)
```

```
Sample Predictions on 30% Sampled Data:
[58.9692      0.26536631  2.28897711 22.90061667  3.55893661]
```

Step: Predict Using the Saved Model on 30% Sampled Data

```
# Step 1: Import joblib and load the saved model
import joblib
loaded_model = joblib.load('tuned_random_forest_model.pkl')

# Step 2: Sample 30% of the cleaned numeric data
df_predict_sample = df_numeric.sample(frac=0.3, random_state=42)

# Step 3: Prepare features for prediction
X_predict = df_predict_sample.drop(columns=['retail_sales'])

# Step 4: Select 5 random rows from the sample
new_data = X_predict.sample(5, random_state=42)

# Step 5: Predict using the loaded model
predictions = loaded_model.predict(new_data)

# Step 6: Display the predictions
print("Sample Predictions on 30% Sampled Data:")
print(predictions)
```

Output & Observations:
The model returned predictions on 5 unseen samples from a 30% random subset of the cleaned dataset:

```
[58.9692, 0.26536631, 2.28897711, 22.90061667, 3.55893661]
```

These values represent predicted `retail_sales` values for the selected input features. The wide range reflects the model's ability to adapt to varied input patterns.

Importance:
This confirms the saved model is reusable and memory-efficient when run on a reduced dataset. It demonstrates how the trained model can be used in real-world scenarios for scoring new or incoming data.

Next Step:
(Optional) Build a lightweight Streamlit or Flask app to allow users to input feature values and receive instant predictions from this saved model.

In [75]:
```
!pip install streamlit
```

```
Requirement already satisfied: streamlit in c:\users\marty\appdata\local\packages
\pythonsoftwarefoundation.python.3.11_qbz5n2kfra8p0\localcache\local-packages\pyt
hon311\site-packages (1.45.0)
Requirement already satisfied: altair<6,>=4.0 in c:\users\marty\appdata\local\pac
kages\pythonsoftwarefoundation.python.3.11_qbz5n2kfra8p0\localcache\local-package
s\python311\site-packages (from streamlit) (5.5.0)
Requirement already satisfied: blinker<2,>=1.5.0 in c:\users\marty\appdata\local
\packages\pythonsoftwarefoundation.python.3.11_qbz5n2kfra8p0\localcache\local-pac
kages\python311\site-packages (from streamlit) (1.9.0)
Requirement already satisfied: cachetools<6,>=4.0 in c:\users\marty\appdata\local
\packages\pythonsoftwarefoundation.python.3.11_qbz5n2kfra8p0\localcache\local-pac
kages\python311\site-packages (from streamlit) (5.5.2)
Requirement already satisfied: click<9,>=7.0 in c:\users\marty\appdata\local\pack
ages\pythonsoftwarefoundation.python.3.11_qbz5n2kfra8p0\localcache\local-packages
\python311\site-packages (from streamlit) (8.1.8)
Requirement already satisfied: numpy<3,>=1.23 in c:\users\marty\appdata\local\pac
kages\pythonsoftwarefoundation.python.3.11_qbz5n2kfra8p0\localcache\local-package
s\python311\site-packages (from streamlit) (2.2.2)
Requirement already satisfied: packaging<25,>=20 in c:\users\marty\appdata\local
\packages\pythonsoftwarefoundation.python.3.11_qbz5n2kfra8p0\localcache\local-pac
kages\python311\site-packages (from streamlit) (24.2)
Requirement already satisfied: pandas<3,>=1.4.0 in c:\users\marty\appdata\local\p
ackages\pythonsoftwarefoundation.python.3.11_qbz5n2kfra8p0\localcache\local-packa
ges\python311\site-packages (from streamlit) (2.2.3)
Requirement already satisfied: pillow<12,>=7.1.0 in c:\users\marty\appdata\local
\packages\pythonsoftwarefoundation.python.3.11_qbz5n2kfra8p0\localcache\local-pac
kages\python311\site-packages (from streamlit) (11.1.0)
Requirement already satisfied: protobuf<7,>=3.20 in c:\users\marty\appdata\local
\packages\pythonsoftwarefoundation.python.3.11_qbz5n2kfra8p0\localcache\local-pac
kages\python311\site-packages (from streamlit) (6.30.2)
Requirement already satisfied: pyarrow>=7.0 in c:\users\marty\appdata\local\packa
ges\pythonsoftwarefoundation.python.3.11_qbz5n2kfra8p0\localcache\local-packages
\python311\site-packages (from streamlit) (20.0.0)
Requirement already satisfied: requests<3,>=2.27 in c:\users\marty\appdata\local
\packages\pythonsoftwarefoundation.python.3.11_qbz5n2kfra8p0\localcache\local-pac
kages\python311\site-packages (from streamlit) (2.32.3)
Requirement already satisfied: tenacity<10,>=8.1.0 in c:\users\marty\appdata\loca
l\packages\pythonsoftwarefoundation.python.3.11_qbz5n2kfra8p0\localcache\local-pa
ckages\python311\site-packages (from streamlit) (9.1.2)
Requirement already satisfied: toml<2,>=0.10.1 in c:\users\marty\appdata\local\pa
ckages\pythonsoftwarefoundation.python.3.11_qbz5n2kfra8p0\localcache\local-packag
es\python311\site-packages (from streamlit) (0.10.2)
Requirement already satisfied: typing-extensions<5,>=4.4.0 in c:\users\marty\appd
ata\local\packages\pythonsoftwarefoundation.python.3.11_qbz5n2kfra8p0\localcache
\local-packages\python311\site-packages (from streamlit) (4.12.2)
Requirement already satisfied: watchdog<7,>=2.1.5 in c:\users\marty\appdata\local
\packages\pythonsoftwarefoundation.python.3.11_qbz5n2kfra8p0\localcache\local-pac
kages\python311\site-packages (from streamlit) (6.0.0)
Requirement already satisfied: gitpython!=3.1.19,<4,>=3.0.7 in c:\users\marty\app
data\local\packages\pythonsoftwarefoundation.python.3.11_qbz5n2kfra8p0\localcache
\local-packages\python311\site-packages (from streamlit) (3.1.44)
Requirement already satisfied: pydeck<1,>=0.8.0b4 in c:\users\marty\appdata\local
\packages\pythonsoftwarefoundation.python.3.11_qbz5n2kfra8p0\localcache\local-pac
kages\python311\site-packages (from streamlit) (0.9.1)
Requirement already satisfied: tornado<7,>=6.0.3 in c:\users\marty\appdata\local
\packages\pythonsoftwarefoundation.python.3.11_qbz5n2kfra8p0\localcache\local-pac
kages\python311\site-packages (from streamlit) (6.4.2)
Requirement already satisfied: jinja2 in c:\users\marty\appdata\local\packages\py
thonsoftwarefoundation.python.3.11_qbz5n2kfra8p0\localcache\local-packages\python
311\site-packages (from altair<6,>=4.0->streamlit) (3.1.5)
```

```
Requirement already satisfied: jsonschema>=3.0 in c:\users\marty\appdata\local\pa
ckages\pythonsoftwarefoundation.python.3.11_qbz5n2kfra8p0\localcache\local-packag
es\python311\site-packages (from altair<6,>=4.0->streamlit) (4.23.0)
Requirement already satisfied: narwhals>=1.14.2 in c:\users\marty\appdata\local\p
ackages\pythonsoftwarefoundation.python.3.11_qbz5n2kfra8p0\localcache\local-packa
ges\python311\site-packages (from altair<6,>=4.0->streamlit) (1.38.2)
Requirement already satisfied: colorama in c:\users\marty\appdata\local\packages
\pythonsoftwarefoundation.python.3.11_qbz5n2kfra8p0\localcache\local-packages\pyt
hon311\site-packages (from click<9,>=7.0->streamlit) (0.4.6)
Requirement already satisfied: gitdb<5,>=4.0.1 in c:\users\marty\appdata\local\pa
ckages\pythonsoftwarefoundation.python.3.11_qbz5n2kfra8p0\localcache\local-packag
es\python311\site-packages (from gitpython!=3.1.19,<4,>=3.0.7->streamlit) (4.0.1
2)
Requirement already satisfied: python-dateutil>=2.8.2 in c:\users\marty\appdata\l
ocal\packages\pythonsoftwarefoundation.python.3.11_qbz5n2kfra8p0\localcache\local
-packages\python311\site-packages (from pandas<3,>=1.4.0->streamlit) (2.9.0.post
0)
Requirement already satisfied: pytz>=2020.1 in c:\users\marty\appdata\local\packa
ges\pythonsoftwarefoundation.python.3.11_qbz5n2kfra8p0\localcache\local-packages
\python311\site-packages (from pandas<3,>=1.4.0->streamlit) (2024.2)
Requirement already satisfied: tzdata>=2022.7 in c:\users\marty\appdata\local\pac
kages\pythonsoftwarefoundation.python.3.11_qbz5n2kfra8p0\localcache\local-package
s\python311\site-packages (from pandas<3,>=1.4.0->streamlit) (2024.2)
Requirement already satisfied: charset-normalizer<4,>=2 in c:\users\marty\appdata
\local\packages\pythonsoftwarefoundation.python.3.11_qbz5n2kfra8p0\localcache\loc
al-packages\python311\site-packages (from requests<3,>=2.27->streamlit) (3.4.1)
Requirement already satisfied: idna<4,>=2.5 in c:\users\marty\appdata\local\packa
ges\pythonsoftwarefoundation.python.3.11_qbz5n2kfra8p0\localcache\local-packages
\python311\site-packages (from requests<3,>=2.27->streamlit) (3.10)
Requirement already satisfied: urllib3<3,>=1.21.1 in c:\users\marty\appdata\local
\packages\pythonsoftwarefoundation.python.3.11_qbz5n2kfra8p0\localcache\local-pac
kages\python311\site-packages (from requests<3,>=2.27->streamlit) (2.3.0)
Requirement already satisfied: certifi>=2017.4.17 in c:\users\marty\appdata\local
\packages\pythonsoftwarefoundation.python.3.11_qbz5n2kfra8p0\localcache\local-pac
kages\python311\site-packages (from requests<3,>=2.27->streamlit) (2024.12.14)
Requirement already satisfied: smmap<6,>=3.0.1 in c:\users\marty\appdata\local\pa
ckages\pythonsoftwarefoundation.python.3.11_qbz5n2kfra8p0\localcache\local-packag
es\python311\site-packages (from gitdb<5,>=4.0.1->gitpython!=3.1.19,<4,>=3.0.7->s
treamlit) (5.0.2)
Requirement already satisfied: MarkupSafe>=2.0 in c:\users\marty\appdata\local\pa
ckages\pythonsoftwarefoundation.python.3.11_qbz5n2kfra8p0\localcache\local-packag
es\python311\site-packages (from jinja2->altair<6,>=4.0->streamlit) (3.0.2)
Requirement already satisfied: attrs>=22.2.0 in c:\users\marty\appdata\local\pack
ages\pythonsoftwarefoundation.python.3.11_qbz5n2kfra8p0\localcache\local-packages
\python311\site-packages (from jsonschema>=3.0->altair<6,>=4.0->streamlit) (24.3.
0)
Requirement already satisfied: jsonschema-specifications>=2023.03.6 in c:\users\m
arty\appdata\local\packages\pythonsoftwarefoundation.python.3.11_qbz5n2kfra8p0\lo
calcache\local-packages\python311\site-packages (from jsonschema>=3.0->altair<6,>
=4.0->streamlit) (2024.10.1)
Requirement already satisfied: referencing>=0.28.4 in c:\users\marty\appdata\loca
l\packages\pythonsoftwarefoundation.python.3.11_qbz5n2kfra8p0\localcache\local-pa
ckages\python311\site-packages (from jsonschema>=3.0->altair<6,>=4.0->streamlit)
(0.35.1)
Requirement already satisfied: rpds-py>=0.7.1 in c:\users\marty\appdata\local\pac
kages\pythonsoftwarefoundation.python.3.11_qbz5n2kfra8p0\localcache\local-package
s\python311\site-packages (from jsonschema>=3.0->altair<6,>=4.0->streamlit) (0.2
2.3)
Requirement already satisfied: six>=1.5 in c:\users\marty\appdata\local\packages
\pythonsoftwarefoundation.python.3.11_qbz5n2kfra8p0\localcache\local-packages\pyt
```

```
hon311\site-packages (from python-dateutil>=2.8.2->pandas<3,>=1.4.0->streamlit)
(1.17.0)
[notice] A new release of pip is available: 25.0 -> 25.1.1
[notice] To update, run: C:\Users\marty\AppData\Local\Microsoft\WindowsApps\Pytho
nSoftwareFoundation.Python.3.11_qbz5n2kfra8p0\python.exe -m pip install --upgrade
pip
```

In [ ]: