ETL & Data cleaning Steps

1. **Data was downloaded from Kaggle and the row data was ingested into SQL in a staging schema. All the columns were assigned NVARCHAR(MAX) data type while staging**

```sql
CREATE SCHEMA staging;

-- Ingest raw data
-- Import tables using 'import flat files' into stagin schema

SELECT * FROM staging.olist_customers_dataset
SELECT * FROM staging.olist_geolocation_dataset
SELECT * FROM staging.olist_order_items_dataset
SELECT * FROM staging.olist_order_payments_dataset
SELECT * FROM staging.olist_order_reviews_dataset
SELECT * FROM staging.olist_orders_dataset
SELECT * FROM staging.olist_products_dataset
SELECT * FROM staging.olist_sellers_dataset
SELECT * FROM staging.product_category_name_translation
```

2. **Connected to SQL database with Python using the pyodbc library**

Importing libraries

```python
import numpy as np
import pandas as pd
import pyodbc
import unicodedata
import urllib
from sqlalchemy import create_engine
from sqlalchemy.types import VARCHAR, NVARCHAR, Integer, Float, DateTime, Boolean
```

Establishing connection with SQL database

```
conn_str = (
    r"DRIVER={ODBC driver 17 for sql server};"
    r"SERVER=DESKTOP-47IJHFR\AFTERERROR;"
    r"DATABASE=Olist;"
    r"Trusted_connection=yes;"

)
```

```
cnxn = pyodbc.connect(conn_str)
```

3. **Loaded the tables from SQL database into python**

```
tables = ["olist_geolocation_dataset",
          "olist_customers_dataset",
          "olist_sellers_dataset",
          "olist_products_dataset",
          "olist_orders_dataset",
          "olist_order_items_dataset",
          "olist_order_reviews_dataset",
          "olist_order_payments_dataset",
          "product_category_name_translation"
]
```

```
dfs = {}

for tb in tables:
    dfs[tb] = pd.read_sql(f"SELECT * FROM staging.{tb}", cnxn)
```

4. **Renaming the tables for simplicity**

```
geolocation = dfs["olist_geolocation_dataset"]
customers = dfs["olist_customers_dataset"]
sellers = dfs["olist_sellers_dataset"]
products = dfs["olist_products_dataset"]
orders = dfs["olist_orders_dataset"]
order_items = dfs["olist_order_items_dataset"]
order_payments = dfs["olist_order_payments_dataset"]
order_reviews = dfs["olist_order_reviews_dataset"]
cat_translation = dfs["product_category_name_translation"]
```

5. **Analyzing geolocation table**

In the geolocation table, we have duplicated values in the geolocation_city column due to some non-English characters

```
geolocation["geolocation_city"].unique()    # Unique values of city
```

```
array(['sao paulo', 'são paulo', 'sao bernardo do campo', ..., 'ciríaco',
       'estação', 'vila lângaro'], dtype=object)
```

Columns were normalized, all the non-english characters were removed and we had clean column

```
# Defining a function to normalize city names
def normalize_city(name):
    if pd.isna(name):
        return name

    nfkd = unicodedata.normalize('NFKD', name)      # Decompose unicode characters
    no_accents = ''.join(c for c in nfkd if not unicodedata.combining(c))
    return no_accents.strip().lower().title()
```

```
# Creating a normalized column of city names
geolocation["city_clean"] = geolocation["geolocation_city"].apply(normalize_city)
```

```python
geolocation["geolocation_city"] = geolocation["city_clean"]     # Normalizing the geolocation_city column

del geolocation["city_clean"]   # Deleting city_clean column as it is redundant
```

Data types of the columns where required were changed

```python
# Changing the data types
geolocation["geolocation_zip_code_prefix"] = geolocation["geolocation_zip_code_prefix"].astype(int)
geolocation["geolocation_lat"] = pd.to_numeric(geolocation["geolocation_lat"], errors="coerce")
geolocation["geolocation_lng"] = pd.to_numeric(geolocation["geolocation_lng"], errors="coerce")
```

Unique pairs of zipcode, city and state were created

```python
# Creating unique zip_code, city and state records
geo_grp = geolocation.groupby(
    ["geolocation_zip_code_prefix", "geolocation_city", "geolocation_state"]
).size().reset_index(name="count")

geo_grp.head()
```

	geolocation_zip_code_prefix	geolocation_city	geolocation_state	count
0	1001	Sao Paulo	SP	26
1	1002	Sao Paulo	SP	13
2	1003	Sao Paulo	SP	17
3	1004	Sao Paulo	SP	22
4	1005	Sao Paulo	SP	25

6. Analyzing Customers Table

```
customers.info()    # General observation
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 99441 entries, 0 to 99440
Data columns (total 5 columns):
 #   Column                    Non-Null Count  Dtype 
---  ------                    --------------  ----- 
 0   customer_id               99441 non-null  object
 1   customer_unique_id        99441 non-null  object
 2   customer_zip_code_prefix  99441 non-null  int64 
 3   customer_city             99441 non-null  object
 4   customer_state            99441 non-null  object
dtypes: int64(1), object(4)
memory usage: 3.8+ MB
```

Customers table was clean. AS a precautionary measure, white spaces were removed if any.

7. Analyzing Sellers Table

- Sellers table had some duplicate values in the seller_city column. This was because of non-English characters in some city names. The non-english characters were removed using the normalizae_city function created earlier

```
sellers["clean_city"] = sellers["seller_city"].apply(normalize_city)    # Creating a normalized column of city

sellers["clean_city"].nunique() # Checking unique counts of normalized city column

606

# Earlier there were 611 unique counts for seller_city and there are 606 unique counts for clean_city
# That means seller_city has duplicate values but not visibly duplicate because of non-english characters
# So will replace seller_city with clean_city

sellers["seller_city"] = sellers["clean_city"]  # Swappign seller_city with clean_city values

del sellers["clean_city"]   # Delete clean_city column as it is redundant
```

8. Analyzing Products Table

Two of the column headers had spelling errors, so they were renamed

```
# Renaming the columns
products.rename(
    columns={
        "product_name_lenght" : "product_name_length",
        "product_description_lenght": "product_description_length"
        }, inplace=True
    )
```

English names of product categories were brought into the products table

```
# Merging the English category names with the product table

products = products.merge(
    cat_translation,
    on = "product_category_name",
    how = "left"
)
```

Some null values were observed in the Products Table

```
products.isnull().sum() # Observing the null values
```

```
product_id                      0
product_category_name         610
product_name_length           610
product_description_length    610
product_photos_qty            610
product_weight_g                2
product_length_cm               2
product_height_cm               2
product_width_cm                2
product_category_name_english 623
dtype: int64
```

Analyzing these null values further, we observe that there are 74 distinct values in product_category_name while product_category_name_English has only 72. So these two missing English translation constitute 13 rows. English translations were imputed into these null values.

```
# So there are 13 instances where product_category_name is not null while it's English counterpart is null
# There are 74 unique counts of product_category_name in products table,
# while there are only 72 unique counts for product_category_name_English in cat_translation table
# So everything adds up
```

```
# Imputing null values in product_category_name_english column

products.loc[
    products["product_category_name"]=="pc_gamer",
    "product_category_name_english"
    ] = "pc_gamer"

products.loc[
    products["product_category_name"] == "portateis_cozinha_e_preparadores_de_alimentos",
    "product_category_name_english"
] = "Portable Kitchen Food Prepares"
```

```
products["product_category_name_english"].isna().sum()  # Checking null values
```

```
610
```

```
# The null values have reduced from 623 to 610 after imputation
```

Columns were converted into appropriate data types

```
# Converting columns to numerical data type
num_col = ['product_description_length', 'product_photos_qty', 'product_weight_g',
        'product_length_cm', 'product_height_cm', 'product_width_cm',]

for col in num_col:
    products[col] = pd.to_numeric(products[col])
```

Remaining null values in numerical columns were imputed with median or 0 and in categorical columns with ‘unknown’

```python
# Separating columns from num_col into another list
num_col_weight_dim = ['product_weight_g',
        'product_length_cm', 'product_height_cm', 'product_width_cm']
```

```python
# Imputing the null values with median
medians = products[num_col_weight_dim].median()
products[num_col_weight_dim] = products[num_col_weight_dim].fillna(medians)
```

```python
# Imputing categorical columns with null values and numerical columns with 0
products.fillna({
    "product_category_name": "unknown",
    "product_name_length" : 0,
    "product_description_length" : 0,
    "product_photos_qyt": 0,
    "product_category_name_english": "unknown"
    }, inplace=True
)
```

```python
# Spelling mistake has resulted in null values still being present in product_photos_qty
products["product_photos_qty"] = products["product_photos_qty"].fillna(0)
```

After imputing null values our table is clean

```python
products.isnull().sum()     #Checking null values
```

```
product_id                       0
product_category_name            0
product_name_length              0
product_description_length       0
product_photos_qty               0
product_weight_g                 0
product_length_cm                0
product_height_cm                0
product_width_cm                 0
product_category_name_english    0
dtype: int64
```

9. Analyzing Orders Table

- We have null values in 3 columns

```
orders.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 99441 entries, 0 to 99440
Data columns (total 8 columns):
 #   Column                         Non-Null Count  Dtype 
---  ------                         --------------  ----- 
 0   order_id                       99441 non-null  object
 1   customer_id                    99441 non-null  object
 2   order_status                   99441 non-null  object
 3   order_purchase_timestamp       99441 non-null  object
 4   order_approved_at              99281 non-null  object
 5   order_delivered_carrier_date   97658 non-null  object
 6   order_delivered_customer_date  96476 non-null  object
 7   order_estimated_delivery_date  99441 non-null  object
dtypes: object(8)
memory usage: 6.1+ MB
```

Before analyzing null values, we converted the data types of date columns into datetime format

```
# Converting date columns into datetime format
date_col = ['order_purchase_timestamp',
            'order_approved_at', 'order_delivered_carrier_date',
            'order_delivered_customer_date', 'order_estimated_delivery_date']

for col in date_col:
    orders[col] = pd.to_datetime(
        orders[col],
        errors = "coerce",
        infer_datetime_format = "True"
)
```

We calculated the time difference between order_purchase_timestamp & order_approved_at, order_purchase_timestamp & orderd_delivered_carrier_date, and order_purchase_timestamp & order_delivered_customer_date. The median of this time difference was used to impute the null values

```
# Calculating time difference between order_purchase_timestamp and order_approved_at
mask = orders["order_approved_at"].notna()
time_diff = (
    orders.loc[mask, "order_approved_at"]
    - orders.loc[mask, "order_purchase_timestamp"]
)
```

```
# Calculating the median time difference between order_purchase_time and order_approved_at
median_time_diff = time_diff.median()
median_time_diff
```

```
Timedelta('0 days 00:20:36')
```

```
# Imputing null values for order_approved_at
orders["order_approved_at"] = orders["order_approved_at"].fillna(
    orders["order_purchase_timestamp"] + median_time_diff)
```

```
orders["order_approved_at"].isnull().sum()  # Sanity check for null values
```

```
0
```

```
# Imputing null values for order_delivered_carrier_date column
mask_carrier = orders["order_delivered_carrier_date"].notna()
time_diff_carrier = (
    orders.loc[mask_carrier, "order_delivered_carrier_date"] -
    orders.loc[mask_carrier, "order_purchase_timestamp"]
)

median_time_diff_carrier = time_diff_carrier.median()

orders["order_delivered_carrier_date"] = orders["order_delivered_carrier_date"].fillna(
    orders["order_purchase_timestamp"] + median_time_diff_carrier)

# Imputing null values for order_delivered_customer_date column
mask_customer = orders["order_delivered_customer_date"].notna()
time_diff_customer = (
    orders.loc[mask_customer, "order_delivered_customer_date"] -
    orders.loc[mask_customer, "order_purchase_timestamp"]
)

median_time_diff_customer = time_diff_customer.median()

orders["order_delivered_customer_date"] = orders["order_delivered_customer_date"].fillna(
    orders["order_purchase_timestamp"] + median_time_diff_customer)
```

No null values remain after imputation

```
orders.isnull().sum()        # Sanity check for null values

order_id                         0
customer_id                      0
order_status                     0
order_purchase_timestamp         0
order_approved_at                0
order_delivered_carrier_date     0
order_delivered_customer_date    0
order_estimated_delivery_date    0
dtype: int64
```

10. Analyzing Order_reviews table

- There are null values in comment_tile and comment_message columns

```
order_reviews.info()    # General observation

<class 'pandas.core.frame.DataFrame'>
RangeIndex: 99224 entries, 0 to 99223
Data columns (total 7 columns):
 #   Column                   Non-Null Count  Dtype 
---  ------                   --------------  ----- 
 0   review_id                99224 non-null  object
 1   order_id                 99224 non-null  object
 2   review_score             99224 non-null  object
 3   review_comment_title     11566 non-null  object
 4   review_comment_message   40968 non-null  object
 5   review_creation_date     99224 non-null  object
 6   review_answer_timestamp  99224 non-null  object
dtypes: object(7)
memory usage: 5.3+ MB
```

We further found out that the same review id has been assigned to multiple order ids. This is a data integrity issue. Same review id cannot be assigned to multiple order ids. Furthermore, we also found that some of the orders have multiple reviews. For the sake of simplicity, we have only taken the latest review for a particular order

```
# Dedupe order_id, keep the latest review

order_reviews = order_reviews.sort_values("review_answer_timestamp")
order_reviews = order_reviews.drop_duplicates(subset="order_id", keep="last")
```

We created a Boolean column has_review. It will populate True if there is a comment else False

```
# Create a boolean flag column
order_reviews["has_reviews"] = order_reviews["review_comment_message"].notna()
```

Finally, we changed the data types of columns

```
# Changing the data types
order_reviews["review_score"] = order_reviews["review_score"].astype(int)
order_reviews["review_creation_date"] = pd.to_datetime(order_reviews["review_creation_date"], errors= "coerce")
order_reviews["review_answer_timestamp"] = pd.to_datetime(order_reviews["review_answer_timestamp"], errors= "coerce")
```

11. Analyzing order_items table

- Order_items table is clean

```
# General observation
order_items.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 112650 entries, 0 to 112649
Data columns (total 7 columns):
 #   Column               Non-Null Count   Dtype
---  ------               --------------   -----
 0   order_id             112650 non-null  object
 1   order_item_id        112650 non-null  object
 2   product_id           112650 non-null  object
 3   seller_id            112650 non-null  object
 4   shipping_limit_date  112650 non-null  object
 5   price                112650 non-null  object
 6   freight_value        112650 non-null  object
dtypes: object(7)
memory usage: 6.0+ MB
```

- Changing the data types where required

```
# Change the data types
order_items["order_item_id"] = order_items["order_item_id"].astype(int)
order_items["shipping_limit_date"] = pd.to_datetime(order_items["shipping_limit_date"], errors="coerce")
order_items["price"] = pd.to_numeric(order_items["price"], errors="coerce")
order_items["freight_value"] = pd.to_numeric(order_items["freight_value"], errors="coerce")
```

12. Analyzing order_payments table

- Order_payments table is clean. We just changed the data types

```
# Change the data types
order_payments["payment_sequential"] = order_payments["payment_sequential"].astype(int)
order_payments["payment_installments"] = order_payments["payment_installments"].astype(int)
order_payments["payment_value"] = order_payments["payment_value"].astype(float)
```

```
# General observation
order_payments.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 103886 entries, 0 to 103885
Data columns (total 5 columns):
 #   Column                Non-Null Count   Dtype
---  ------                --------------   -----
 0   order_id              103886 non-null  object
 1   payment_sequential    103886 non-null  int32
 2   payment_type          103886 non-null  object
 3   payment_installments  103886 non-null  int32
 4   payment_value         103886 non-null  float64
dtypes: float64(1), int32(2), object(2)
memory usage: 3.2+ MB
```

13. Exporting cleaned data back into SQL

- Engine was created using SQL Alchemy. Data types of all the columns were defined. A data frame of all the cleaned tables was created and finally the data was exported back into SQL in analytics schema

```python
# url encode and plug into sql alchemy
quoted = urllib.parse.quote_plus(conn_str)
engine = create_engine(f"mssql+pyodbc:///?odbc_connect={quoted}")
```

```python
# Defining data types
dtype_map = {
    'customers': {
        'customer_id': VARCHAR(length=50),
        'customer_unique_id': VARCHAR(length=50),
        'customer_zip_code_prefix': VARCHAR(length=10),
        'customer_city': NVARCHAR(length=100),
        'customer_state': VARCHAR(length=2),
    },
    'orders': {
        'order_id': VARCHAR(length=50),
        'customer_id': VARCHAR(length=50),
        'order_status': VARCHAR(length=20),
        'order_purchase_timestamp': DateTime(),
        'order_approved_at': DateTime(),
        'order_delivered_carrier_date': DateTime(),
        'order_delivered_customer_date': DateTime(),
        'order_estimated_delivery_date': DateTime(),
    },
    'order_items': {
        'order_id': VARCHAR(length=50),
        'order_item_id': Integer(),
        'product_id': VARCHAR(length=50),
        'seller_id': VARCHAR(length=50),
        'shipping_limit_date': DateTime(),
        'price': Float(),
        'freight_value': Float(),
    },
```

```
'order_payments': {
    'order_id': VARCHAR(length=50),
    'payment_sequential': Integer(),
    'payment_type': VARCHAR(length=20),
    'payment_installments': Integer(),
    'payment_value': Float(),
},
'order_reviews': {
    'order_id': VARCHAR(length=50),
    'review_score': Integer(),
    'review_comment_title': NVARCHAR(length='max'),
    'review_comment_message': NVARCHAR(length='max'),
    'review_creation_date': DateTime(),
    'review_answer_timestamp': DateTime(),
    'has_review': Boolean(),
},
'products': {
    'product_id': VARCHAR(length=50),
    'product_category_name': NVARCHAR(length=100),
    'product_name_length': Integer(),
    'product_description_length': Integer(),
    'product_photos_qty': Integer(),
    'product_weight_g': Float(),
    'product_length_cm': Float(),
    'product_height_cm': Float(),
    'product_width_cm': Float(),
    'product_category_name_english': NVARCHAR(length=100),
},
```

```
    'sellers': {
        'seller_id': VARCHAR(length=50),
        'seller_zip_code_prefix': VARCHAR(length=10),
        'seller_city': NVARCHAR(length=100),
        'seller_state': VARCHAR(length=2),
    },
    'geolocation': {
        'geolocation_zip_code_prefix': VARCHAR(length=10),
        'geolocation_lat': Float(),
        'geolocation_lng': Float(),
        'geolocation_city': NVARCHAR(length=100),
        'geolocation_state': VARCHAR(length=2),
    },
    'category_translation': {
        'product_category_name': NVARCHAR(length=100),
        'product_category_name_english': NVARCHAR(length=100),
    },
    'geo_grp' : {
        'geolocation_zip_code_prefix' : VARCHAR(length=10),
        'geolocation_city' : NVARCHAR(length=100),
        'geolocation_state' : VARCHAR(2),
    }
}
```

```
# Dictionary of cleaned data frames
clean_dfs = {
    "customers" : customers,
    "sellers" : sellers,
    "products" : products,
    "orders" : orders,
    "order_items" : order_items,
    "order_reviews" : order_reviews,
    "order_payments" : order_payments,
    "geolocation" : geolocation,
    "cat_translation" : cat_translation,
    "geo_grp" : geo_grp
}
```

```python
for name, table in clean_dfs.items():
    table.to_sql(
        name = name,
        schema = "analytics",
        con = engine,
        if_exists = "replace",
        index = False,
        dtype = dtype_map.get(name)
    )

    print(f"{name} : {len(table):,} rows exported to analytics.{name}")
```

```
customers : 99,441 rows exported to analytics.customers
sellers : 3,095 rows exported to analytics.sellers
products : 32,951 rows exported to analytics.products
orders : 99,441 rows exported to analytics.orders
order_items : 112,650 rows exported to analytics.order_items
order_reviews : 98,673 rows exported to analytics.order_reviews
order_payments : 103,886 rows exported to analytics.order_payments
geolocation : 1,000,163 rows exported to analytics.geolocation
cat_translation : 71 rows exported to analytics.cat_translation
geo_grp : 19,616 rows exported to analytics.geo_grp
```

```sql
CREATE SCHEMA analytics;


-- Insert cleaned data into analytics schema

SELECT * FROM analytics.cat_translation
SELECT * FROM analytics.customers
SELECT * FROM analytics.geo_grp
SELECT * FROM analytics.geolocation
SELECT * FROM analytics.order_items
SELECT * FROM analytics.order_payments
SELECT * FROM analytics.order_reviews
SELECT * FROM analytics.orders
SELECT * FROM analytics.products
SELECT * FROM analytics.sellers
```

14. Created table with full name of state

```
CREATE TABLE analytics.state (
    state_abbr VARCHAR(2) NOT NULL PRIMARY KEY,
    state_full VARCHAR(100) NOT NULL
)

ALTER TABLE analytics.state
ALTER COLUMN state_full NVARCHAR(100);
```

```
INSERT INTO analytics.state (state_abbr, state_full)
VALUES
  ('AC', 'Acre'),
  ('AL', 'Alagoas'),
  ('AP', 'Amapá'),
  ('AM', 'Amazonas'),
  ('BA', 'Bahia'),
  ('CE', 'Ceará'),
  ('DF', 'Distrito Federal'),
  ('ES', 'Espírito Santo'),
  ('GO', 'Goiás'),
  ('MA', 'Maranhão'),
  ('MT', 'Mato Grosso'),
  ('MS', 'Mato Grosso do Sul'),
  ('MG', 'Minas Gerais'),
  ('PA', 'Pará'),
  ('PB', 'Paraíba'),
  ('PR', 'Paraná'),
  ('PE', 'Pernambuco'),
  ('PI', 'Piauí'),
  ('RJ', 'Rio de Janeiro'),
  ('RN', 'Rio Grande do Norte'),
  ('RS', 'Rio Grande do Sul'),
  ('RO', 'Rondônia'),
  ('RR', 'Roraima'),
  ('SC', 'Santa Catarina'),
  ('SP', 'São Paulo'),
  ('SE', 'Sergipe'),
  ('TO', 'Tocantins');
```

15. Assigned Primary Keys and Foreign Keys

```sql
/* Assign Primary Key to customers table */

ALTER TABLE analytics.customers
ALTER COLUMN customer_id VARCHAR(50) NOT NULL;

ALTER TABLE analytics.customers
ADD CONSTRAINT pk_customer_id
PRIMARY KEY (customer_id)

/* Assign Primary key to products table */

ALTER TABLE analytics.products
ALTER COLUMN product_id VARCHAR(50) NOT NULL;

ALTER TABLE analytics.products
ADD CONSTRAINT pk_product_id
PRIMARY KEY (product_id);

/* Assign Primary Key to Sellers table */

ALTER TABLE analytics.sellers
ALTER COLUMN seller_id VARCHAR(50) NOT NULL;

ALTER TABLE analytics.sellers
ADD CONSTRAINT pk_seller_id
PRIMARY KEY (seller_id);
```

```sql
/* Assign Primary key to Orders table */

ALTER TABLE analytics.orders
ALTER COLUMN order_id VARCHAR(50) NOT NULL;

ALTER TABLE analytics.orders
ADD CONSTRAINT pk_order_id
PRIMARY KEY (order_id);

/* Assign Foreign key constraint to Orders table */

ALTER TABLE analytics.orders
ADD CONSTRAINT fk_customer_id
FOREIGN KEY (customer_id)
REFERENCES analytics.customers(customer_id)
```

```sql
/* Assign Foreign Key constraint to Order_items table */

ALTER TABLE analytics.order_items
ADD CONSTRAINT fk_order_id
FOREIGN KEY (order_id)
REFERENCES analytics.orders(order_id)

ALTER TABLE analytics.order_items
ADD CONSTRAINT fk_product_id
FOREIGN KEY (product_id)
REFERENCES analytics.products(product_id);

ALTER TABLE analytics.order_items
ADD CONSTRAINT fk_seller_id
FOREIGN KEY (seller_id)
REFERENCES analytics.sellers(seller_id);

/* Assign Foreign Key constraint to order_payments */

ALTER TABLE analytics.order_payments
ADD CONSTRAINT fk_order_id_payments
FOREIGN KEY (order_id)
REFERENCES analytics.orders (order_id);
```

```sql
/* Assign Foreign Key constraint to order_reviews table */

ALTER TABLE analytics.order_reviews
ADD CONSTRAINT fk_order_id_reviews
FOREIGN KEY (order_id)
REFERENCES analytics.orders(order_id);
```

The data was then imported into Power BI for analysis and reporting