FLIP1 handover jason - System Architecture / Development / Maintainment

1. Context

What we've heard from the MCH Demand Planning team:

- MCH DP team requests a tool to enhance their productivity and improve accuracy in their work processes. Currently, they rely on Excel and Power Query for operations, which can be time-consuming and have limitations in terms of computing complexity and forecasting capabilities
- The existing forecasting approach is limited to Channel (GT/MT) and Region (North, South, Central) on a monthly basis for the next 12 months and weekly basis for the next 4-6 weeks. This restricts their ability to perform more granular analyses (province, weekly)
- Furthermore, the team manually maintains the database, which is time-intensive and can be difficult to track. They'd like a more structured and auto-synchronized system to handle the database more effectively

The objective of the new tool is to address these challenges by providing a faster, more accurate, and more granular solution.

2. System Architecture

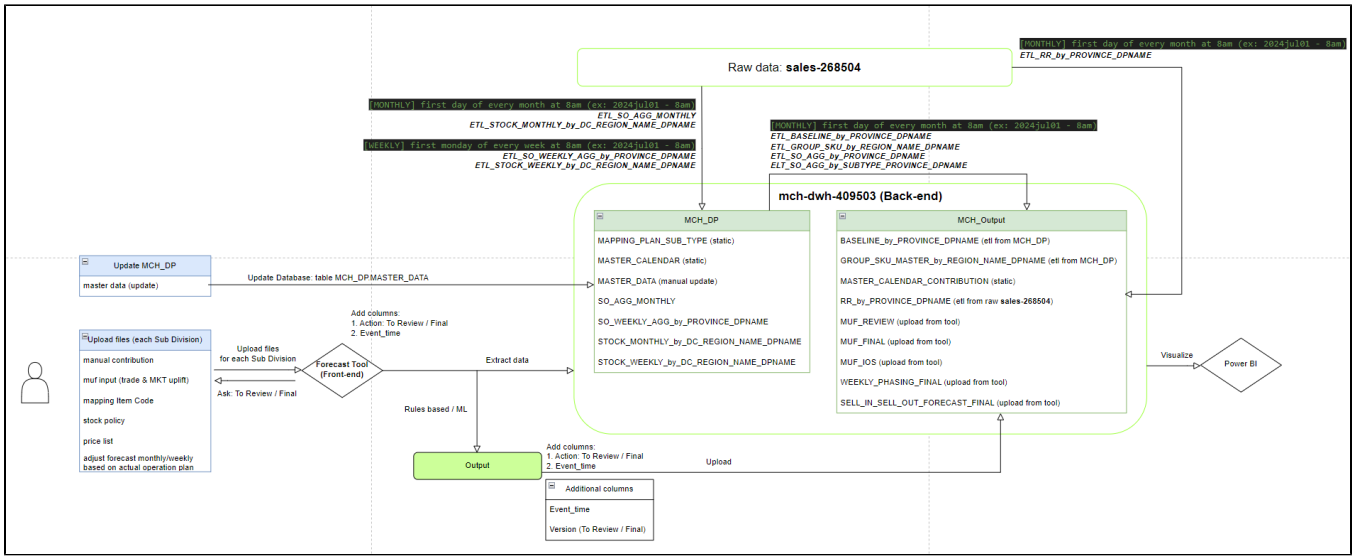
Tool is packaged as a desktop app that allows users to upload assumption data.

It extracts raw data from the main database (currently BigQuery - Google Cloud Service).

The backend server then processes the 2 dataset (input assumption & raw data), generates forecast results, which are stored back to BigQuery or display the simulation to users.

Features:

- Simulate forecast based on input assumptions for any period of time
- Store final forecast Sell Out on database -> automatically convert to Sell In weekly and send emails to Operation involved team PIC (Trade, RSM, SO, IT, DP, SP, DRP, MPO, Production)
- Report tracking forecast performance (compare actual vs forecast, Accuracy, Bias,...)



3. Development

The app is developed under Functional Programing convention. The main reason to choose this approach rather than OOP (Object-oriented Programming) is that each calculation step is a component and output from each step is dependent on each other, this makes FP more suitable in this case to create small, reusable components.

3.1. Code explaination

All the calculation logic and functions used in the app is packed inside this file ***functions.py***. All the functions/objects used are explained below:

3.1.1. Collect & Prepare data

- ***service account key:*** mch-dwh-409503-0b23ccc41bb8.json
- ***client:*** to interact with the Back-end (BigQuery) we need to authenticate by a service account and the client object is generated from credentials info in the service account key
- ***etl_collect_data(client):*** this function uses the ***client*** constructed to retrieve all data needed from Back-end Database then return several dataframe objects (***df_baseline, df_master_calendar, df_groupSKU_master, df_groupSKU_byProvince, df_groupSKU_bySubTypeProvince, df_master_date, df_forecast_week, df_week, df_RR_by_PROVINCE_DPNAME, df_SO_weekly_last_5w, df_stock_weekly_last_5w, df_stock_monthly_last_2m, df_past_innovation, df_actual_so***)
- ***etl_baseline(...):*** this function takes two dataframes ***df_baseline, df_master_calendar*** as input and return the forecast baseline/baseline_adjustment /seasonality building blocks for the next 12 months
- ***province_contribution_byDPName(...) / province_contribution_byGroupSKU(...) / province_contribution_bySubDivision(...) / province_contribution_byDefault(...):*** these functions takes the dataframe ***df_groupSKU_byProvince*** as input and return contributions of each province at different channel in national wide for different level (DP Name, Group SKU, Sub Division Name, Default) => to split from national wide level in assumption input to province level.
- ***planSubType_contribution_byDPName(...) / planSubType_contribution_byGroupSKU(...) / planSubType_contribution_bySubDivision(...) / planSubType_contribution_byDefault(...):*** these functions takes the dataframe ***df_groupSKU_bySubTypeProvince*** as input and return contributions of each sub_type in a channel for different level (DP Name, Group SKU, Sub Division Name, Default)

3.1.2. Process Input data

Users need to upload assumption data. The template and format of all files can be found here: input files - template. These input files need pre-processing in order to use for calculation.

- ***etl_price(...):*** this function takes the input price list (in week) ***df_price_list*** by users and convert to monthly price for each product at each region/channel. The output is a dataframe named ***df_price_list_month***
- ***etl_clean_transform_contribution_input(...):*** this function takes two input files ***df_contribution, df_region_contribution*** and pre-clean to use in next steps. The output includes two dataframes ***df_contribution_melted, df_region_contribution_melted***
- ***etl_manual_groupSKU(...):*** this function takes two pre-processed dataframes ***df_contribution_melted, df_region_contribution_melted*** and ***df_groupSKU_master*** collected from step 3.1.1 then calculate the manual contribution to over-write the default contribution. The output is a dataframe ***df_manual_groupSKU***
- ***etl_default_groupSKU(...):*** this function takes the pre-processed dataframe ***df_contribution_melted*** and ***df_groupSKU_master*** collected from step 3.1.1 then calculate the default contribution. The output is a dataframe ***df_default_groupSKU***
- ***etl_clean_transform_muf_input(...):*** this function takes the input file of ***Monthly Forecast Assumption for other building blocks*** except (Baseline, Baseline_Adjustment, Seasonality which are collected directly from Back-end Database) and pre-clean to use in next steps. The output is a dataframe ***df_muf_input_melted***
- ***etl_stock_policy(...):*** this function takes the input file of ***Stock Policy*** defined by team and transform to use in next steps. The output is a dataframe ***df_stock_policy_byProvince_DPName***
- ***etl_s1(...):*** this function takes the input file of current month operation ***df_s1*** and pre-clean to use in next steps. The output is a dataframe ***df_muf_s1***

3.1.3. Calculation steps and Output from app

- ***etl_muf(...):*** this function takes the the pre-processed ***df_muf_input_melted*** and some other dataframes to simulate forecast results. The output is a dataframe ***df_muf***
- ***etl_actualization(...):*** this function takes the output of simulated forecast from last step ***df_muf*** then return a dataframe for **ACTUALIZATION** - ***df_actualization*** that can be exported to check
- ***etl_muf_adjustment(...):*** this function takes the output of simulated forecast from last step ***df_muf*** and adjusted forecast file ***df_muf_adjustment_new_template*** *(if there is any adjustment)* then adjust the ***df_muf*** accordingly to return a new dataframe ***df_muf_adjustment_final***
- ***etl_MUF_withDC(...):*** this function takes either the output of simulated forecast ***df_muf*** or adjusted forecast ***df_muf_adjustment_final*** *(if there is any adjustment)* then mapping the Distribution Center (DC) that serves for the region/channel. The output is a dataframe ***df_MUF_withDC***
- ***etl_muf_ios(...):*** this function takes the output of simulated forecast mapping to DC ***df_MUF_withDC,*** current month operation ***df_muf_s1*** to simulate operation (In-Out Stock). It calculates the monthly Sell In based on monthly Forecasted Sell Out and Stock Policy. It is a recursive problem when we know the first period t=0 then we need to calculate SI(t) and Close(t). The rule is defined below, for more details on how to implement please check the source code.

```
# AVAILABLE: so(0), close(0), si(0)
# RULE: so(t) + close(t) = si(t) + close(t-1)
# if so(t) + close_policy(t) < close(t-1):
#     => si(t) = 0
#     => close(t) = close(t-1) - so(t)
# else:
#     => si(t) = so(t) + close_policy(t) - close(t-1)
#     => close(t) = close_policy(t)
```

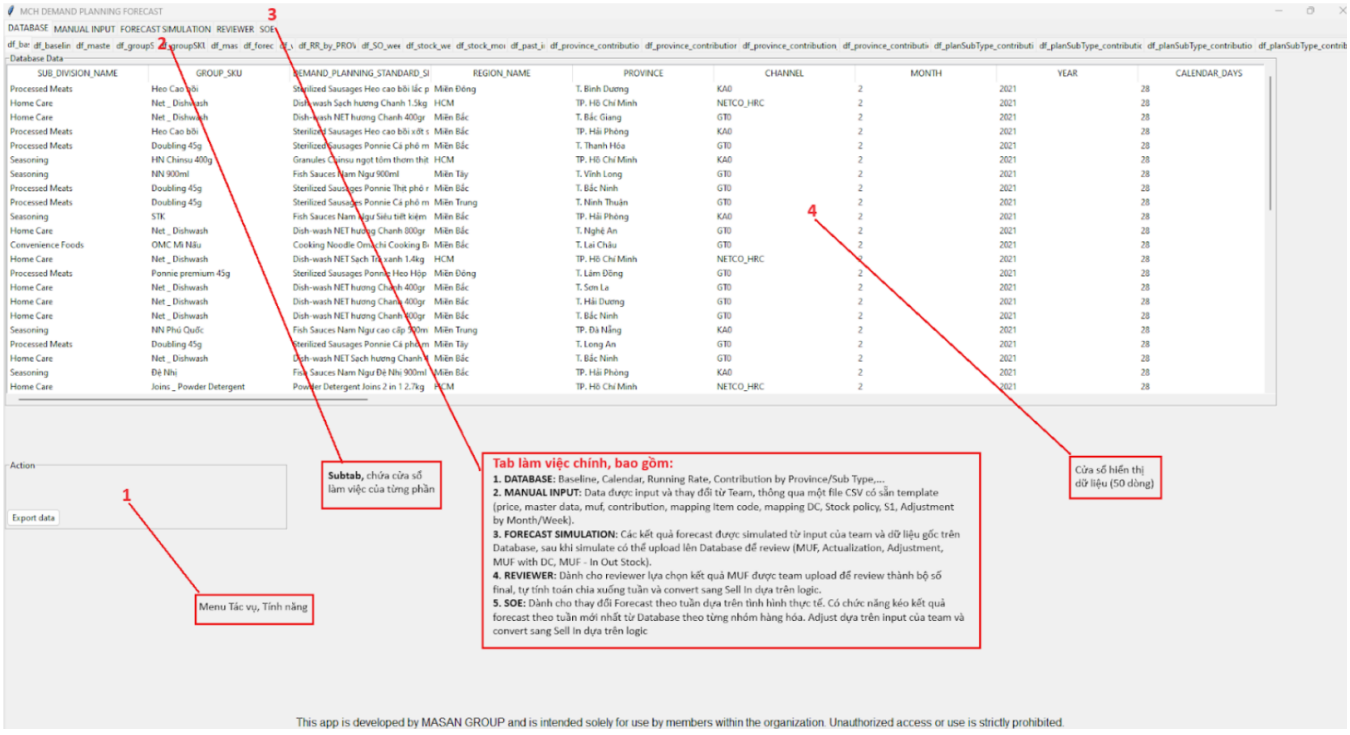
- ***etl_weeklyPhasing(…):*** this function takes the output of simulated forecast mapping to DC ***df_MUF_withDC*** then split to weekly forecast as defined logic. The output is a weekly forecast dataframe ***df_weeklyPhasing_agg***
- ***etl_conversion_si(…):*** this function takes the output of weekly forecast ***df_weeklyPhasing_agg*** and stock policy ***df_stock_policy_byProvince_DPName*** to convert from Sell Out to Sell In based on Stock Policy, It is similar to function ***etl_muf_ios(…)*** but in different level (week instead of month)
- ***etl_soe(…):*** this function takes the output of weekly forecast ***df_weeklyPhasing_agg*** and actual Sell Out last 5 weeks ***df_SO_weekly_last_5w*** and return a dataframe as requested from users for them to track actual operation last 5 week and forecast, and adjust forecast (if needed) for upcoming weeks as actual operation. The output is a dataframe ***df_soe***
- ***etl_soe_adjustment(…):*** this function takes the latest of weekly forecast ***df_weeklyPhasing_agg*** from database and adjusted file template input by users ***df_soe_adjustment*** then adjust the forecast as defined logic. The output is a new weekly forecast for upcoming weeks then it will be uploaded and stored back to Back-end database.

3.2. User Interface & flow

The UI design and all features are packed inside this file ui.py. All the functions/objects used are explained below:

Step 1: DATABASE

Run the app => Trigger all functions in ***3.1.1.*** and store all output dataframes in a dictionary object named ***database_data***. Then display each element in this dictionary object in the tab DATABASE

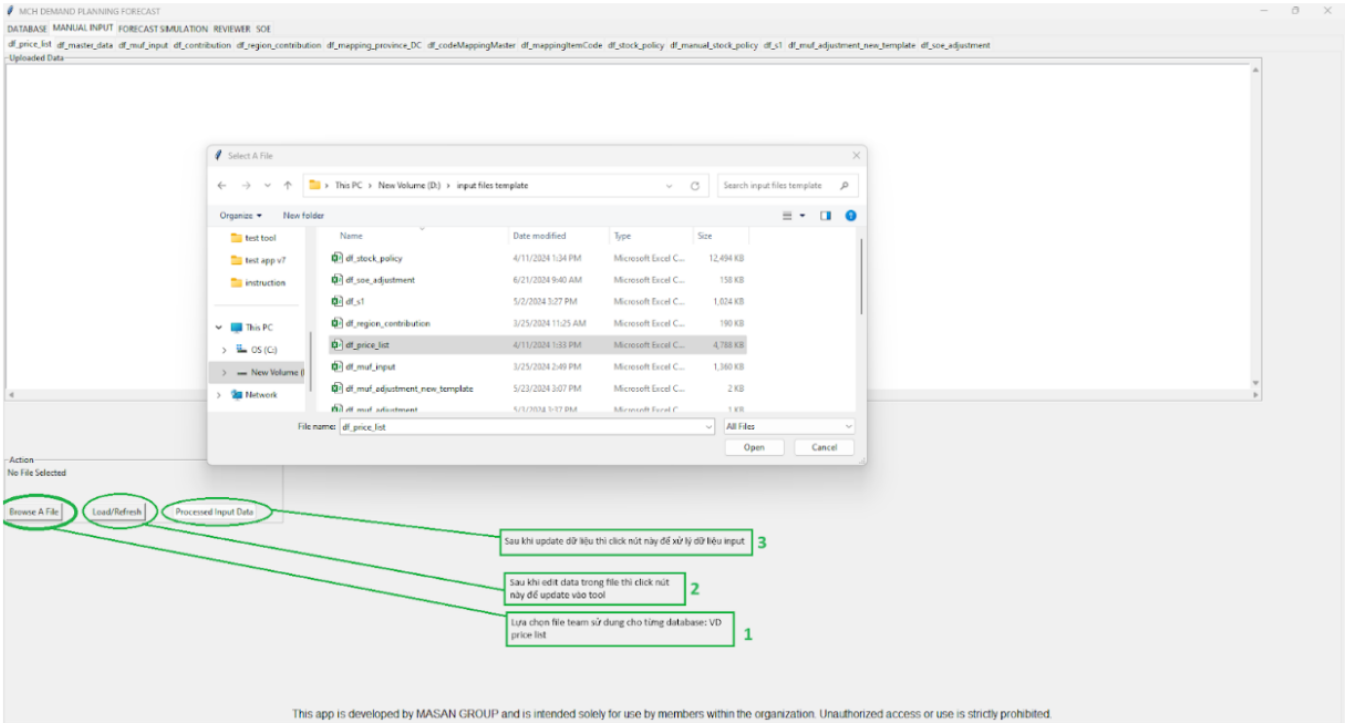


Step 2: MANUAL INPUT

Browse the file path to each files template, the button Load/Refresh will store the file path and current data in a dictionary object named ***upload_file_paths.*** Then display each element in this dictionary object in the tab MANUAL INPUT.

The reason to keep the file path is that users require to edit the input data directly from excel and the app needs to capture any changes. => The file path will let the app knows where it has to go and reload the new edited data (user finishes editing just needs to save the file, no need to closes it and goes back to the app hits refresh).

The button Process Input Data is packed inside all the function in ***3.1.2.*** Everytime user makes a change they need to click the button and trigger processing input data. The output will be stored in a dictionary object named ***processed_input_data***



Step 3: SIMULATION

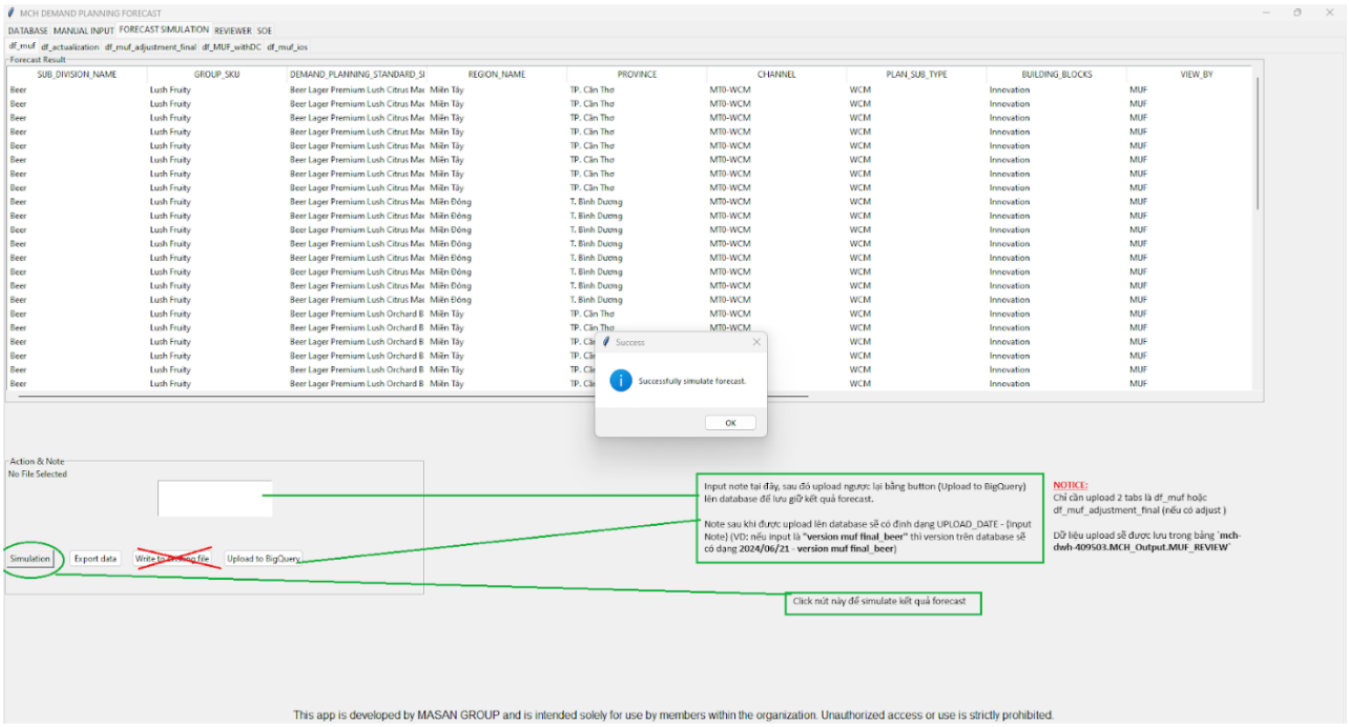
After uploading all files requested => we have the tab Simulation which holds all the simulated forecast results from current working session.

At first each simulated output is created as an empty dataframes and stored in the ***output_data*** dictionary. The button simulation in each tab will be packed within the function generating the output for each tab (ex: the simulation button in tab **df_muf** will trigger the function ***etl_muf(..)*** ; the simulation button in tab ***df_muf_ios*** will trigger the function ***etl_muf_ios(..)***)

The **Upload to BigQuery** button in each tab (if availabe) has the functionality to upload the simulated forecast (review version) to store in our Back-end Database. 2 tabs that require storing output are df_muf (df_muf_adjustment_final if there is any adjust) and df_muf_ios.

The destination table in Database for each tab is:

- **df_muf**: mch-dwh-409503.MCH_Output.MUF_REVIEW
- **df_muf_adjustment_final** : mch-dwh-409503.MCH_Output.MUF_REVIEW
- **df_muf_ios**: mch-dwh-409503.MCH_Output.MUF_IOS



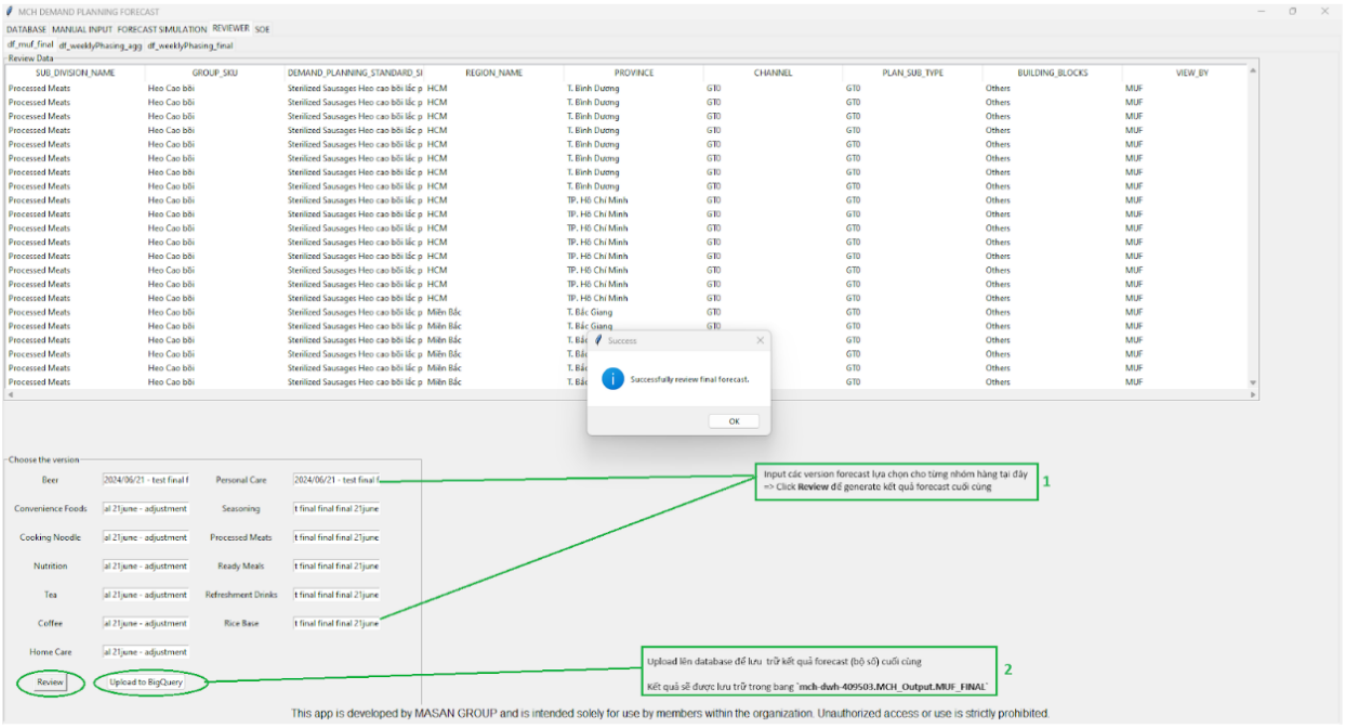
Step 4: REVIEW

Outputs of this tab are firstly created as empty object and stored in a dictionary object **final_output_data** including ***df_muf_final, df_weeklyPhasing_agg, df_weeklyPhasing_final***

After reviewing the simulated forecast uploaded by team members, the reviewer decides on the final version for each Sub Division, then input the version in each box in the app.

The review button will trigger a Query to table **mch-dwh-409503.MCH_Output.MUF_REVIEW** and retrieve the data.

The Upload to Bigquery button inside this tab will upload this version as final forecast which will be used for operation to Back-end Database (table: **mch-dwh-409503.MCH_Output.MUF_FINAL**)

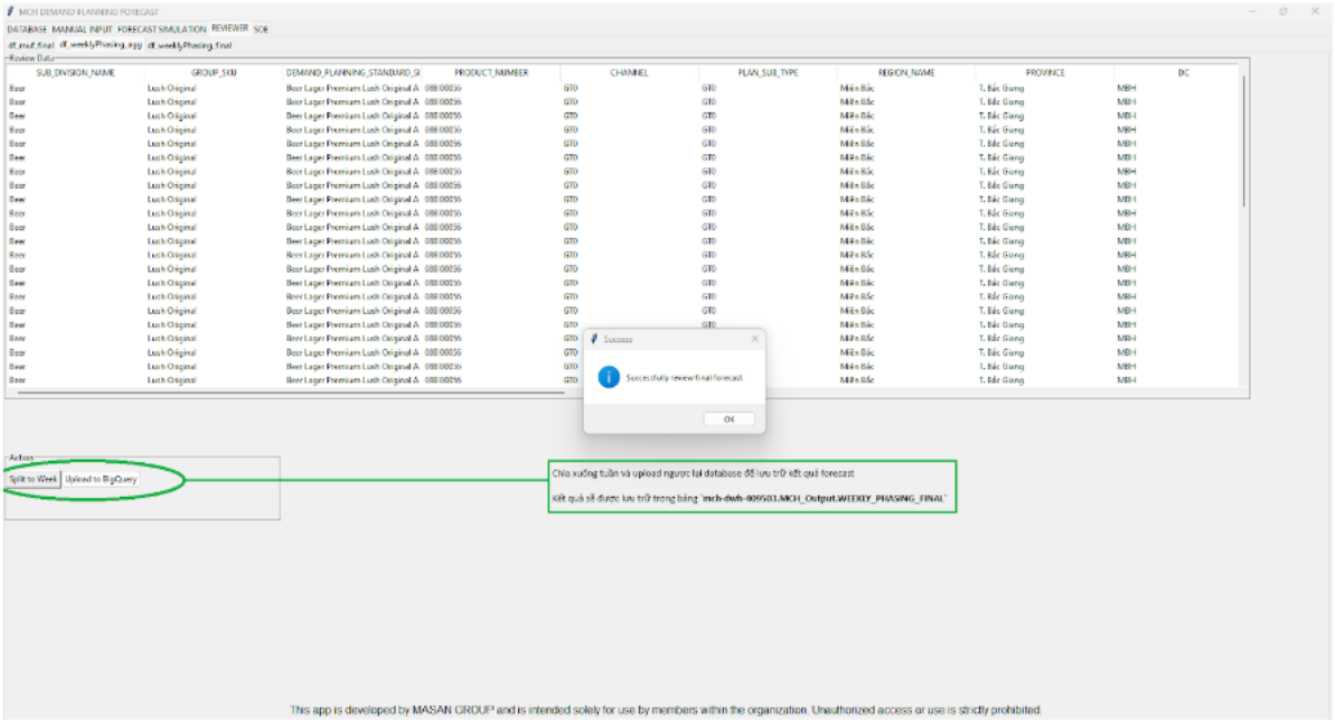


The next two tab has the functionality to split from Monthly Forecast to Weekly Forecast and Convert to Sell In. Two functions ***etl_weeklyPhasing(..)*** and ***etl_conversion_si(..)*** are used in these tabs.

Again there are upload outputs to Database.

The destination table in Database for each tab is:

- **df_muf_final**: mch-dwh-409503.MCH_Output.MUF_FINAL
- **df_weeklyPhasing_agg**: mch-dwh-409503.MCH_Output.WEEKLY_PHASING_FINAL
- **df_weeklyPhasing_final**: mch-dwh-409503.MCH_Output.SELL_IN_SELL_OUT_FORECAST_FINAL



Step 5: SOE - adjust weekly forecast based on current operation

This tab allows users to retrieve weekly forecast and plan based on current situation.

The drop down box allows users to choose the Sub Division, and the button **Retrieve latest Weekly Forecast** will trigger a query to table **mch-dwh-409503.MCH_Output.WEEKLY_PHASING_FINAL** (which is the forecast final).

The app will adjust this final forecast by using the template to adjust ***df_soe_adjustment*** then again trigger two functions ***etl_soe_adjustment(..)*** to adjust the weekly forecast and ***etl_conversion_si(..)*** to convert to Sell-In.

After adjustment, the button Upload to BigQuery again upload back to our main database.

The destination table in Database for each tab is:

- **df_weeklyPhasing_soe_adjustment**: mch-dwh-409503.MCH_Output.WEEKLY_PHASING_FINAL
- **df_weeklyPhasing_soe_adjustment_final**: mch-dwh-409503.MCH_Output.SELL_IN_SELL_OUT_FORECAST_FINAL