# DSE3101 Technical Documentation

The Trio: Nguyen Duc Thinh, Tan Win Yi, Li Yu

2025-04-19

## Contents

# I. Introduction

## 1. Project overview

Macroeconomic forecasting plays a critical role in shaping national policy, but it faces a major challenge where real-time GDP data is often revised over time, making forecasting decisions difficult. Research has shown that forecasts using real-time data tend to be less accurate than those using revised data whereby traditional models may not reflect the actual economic situation.

Our project addresses this issue by developing a web-based interactive application that allows users to benchmark various time-series forecasting models; using real-time and revised GDP data. Our application enables users to visualize and compare the accuracy and robustness of different models across data vintages and forecast horizons.

Our main goal is to help policymakers and economists evaluate how forecasts would have performed in real-time, quantify the impact of data revisions, and identify which models remain reliable under changing economic conditions.

## 2. Overall design

Our application consist of two main tabs: Dataset and Model.

The Dataset Tab is where users choose to either work with our provided sample dataset or upload their own. Here user will also set up their interest forecasting period. We also provide a preview of the cleaned data from the starting period, alongside with visualizations to help users to examine the differences in current and vintage data.

The Model tab allows users to select forecasting models for evaluation and customize them by specifying key parameters and features. This interactive setup ensures that users can experiment with different models and directly observe their impact on forecasting performance. Error metrics and visualization will be provided to assess the results.

## 3. Data and methodology

Our primary data source is available at https://www.philadelphiafed.org/surveys-and-data/real-time-data-research/routput.
We also use the FRED API to source additional economic data for model enhancement.

In this project, we evaluate model performance using two different data settings: vintage data (data available at the time of forecasting) and latest vintage data (revised data). This comparison helps quantify the impact of data revisions on forecast accuracy.

We focus on evaluating three models:

- Autoregressive (AR): A benchmark model that uses only past values of GDP growth to generate forecasts.
- Autoregressive Distributed Lag (ADL): Extends the AR model by incorporating additional economic indicators.
- K-Nearest Neighbors (KNN): A non-parametric machine learning model

Model performance is assessed using two standard forecasting error metrics: Root Mean Squared Error (RMSE) and Mean Absolute Error (MAE). These metrics allow us to rank models consistently based on both accuracy and robustness across different data vintages.

Before moving into building the application, it is necessary to go through our logic. The framework of how our forecast works is as follow:

- Current Vintage Prediction: For each point we want to forecast, we will use the vintage data at that point to train the model and generate forecast. For instance, a forecast for 2000Q1 GDP growth will use the 2000Q1 vintage data to generate forecast. This is the real-time data that we have in 2000Q1 about all other dates.
- Latest Vintage Prediction: We will use the latest vintage data for training our model (2025Q1), however, we use only use data up until the time of prediction. Thus, a 2000Q1 forecast with the latest data will only use GDP Growth up until 1999Q4.
- Evaluation: We will then generate two sequences of forecasts, and calculate the error metrics of the sequences with the real value in the latest vintage data. The reason is that this most recent data is what closest to the truth by constantly going through revision.

As the project progressed, we chose to focus exclusively on quarterly vintage data. This decision simplifies the modeling process while still aligning with our primary goal: quantifying the impact of data revisions. With that being said, it is completely possible to extend our model to accompany monthly vintage data as the structure would be very similar to how we handle quarterly data.

## II. Application Backend

### 1. Data Preparation

For data selection, users can either work with our provided sample dataset or upload their own. For sample dataset, we are using the real-time dataset available here for download: https://www.philadelphiafed.org/surveys-and-data/real-time-data-research/routput

For upload option, we currently support uploads in csv, xlsx, or json formats. Additionally, the uploaded dataset must follow a structure similar to the FRED data format to ensure compatibility with our processing.

Our initial plan was to incorporate both quarter vintages and monthly vintages. In order to do this, we need to detect the frequency of the dataset that users chose. This is where we define a function that can detect the frequency of dataset as below.

```r
detect_frequency <- function(data) {
    # Get first vintage column name
    first_col <- names(data)[2]

    if (str_detect(first_col, "M\\d+$")) {
      return("monthly")
    } else if (str_detect(first_col, "Q\\d$")) {
      return("quarterly")
    } else {
      stop("Unsupported Uploaded File!")
    }
  }
```

After we have select our data, we start our data cleaning process. We begin by cleaning the column names, which are in the format "ROUTPUT65Q1", to extract the correct year and quarter. For each subsequent column, we compare the last two digits of the year with the previous one: If the new year is greater than or equal to the previous, we keep the current prefix.

If it's smaller, this signals a rollover into the next century, so we increment the prefix by 1 (e.g., from "19" to "20").

For example, from "98" to "99", we stay in the 1900s. When it shifts from "99" to "00", we move to the 2000s.

```
## Get the right century for the year
  clean_columns <- function(data) {

    data_cols <- names(data)[-1] %>%
      str_remove(pattern = "ROUTPUT")

    yy <- str_sub(data_cols, start = 1, end = 2)
    prev_yy <- yy[1]
    century = 19
    complete_year <- c()
    for (i in 1:length(yy)) {
      cur_yy <- yy[i]
      if (as.numeric(cur_yy) < as.numeric(prev_yy)) {
        century = century + 1
      }
      complete_year[i] <- paste0(century, cur_yy)
      prev_yy <- cur_yy
    }
    return(complete_year)
  }
```

This function will get us the right prefix for each column. As a result, we can use this function and concatenate the prefix with the year and quarter/month and get a better column name ,for instance, "ROUTPUT65Q1" to "1965Q1".

Next, we clean the dataset by reshaping into long format, allowing us to extract and organize the columns into year, quarter, v_year (vintage year), v_quarter/v_month (vintage quarter/month) and the corresponding GDP value. This structure makes it easier to filter for the corresponding vintage in future analysis.

```
clean.data <- function(data, vintage_freq = "quarterly") {

    q <- names(data)[-1] %>% str_remove(pattern = "ROUTPUT") %>% str_sub(start = 1)
    clean_cols <- paste0(clean_columns(data), q)
    names(data)[-1] <- clean_cols
    total_col <- length(names(data))
    ## Clean the data
    clean_data <- data %>%
      mutate(across(2:total_col, as.numeric)) %>%
      pivot_longer(cols = -1, names_to = "vintage",
                   values_to = "current_vintage") %>%
      mutate(year = str_sub(DATE, 1,4),
             quarter = str_sub(DATE, 7,7),
             v_year = str_sub(vintage, 1,4),
             log_current_vintage = log(current_vintage)) %>%
      drop_na()

    if (vintage_freq == "quarterly") {
      final_data <- clean_data %>%
        mutate(v_quarter = str_extract(vintage, pattern = "(?<=Q).*$")) %>%
        select(year, quarter, v_year, v_quarter,
               current_vintage, log_current_vintage) %>%
        mutate(across(1:6, as.numeric)) # Turn all value to numeric
    }
    else {
      final_data <- clean_data %>%
        mutate(v_month = str_extract(vintage, pattern = "(?<=M).*$")) %>%
        select(year, quarter, v_year, v_month, current_vintage, log_current_vintage) %>%
        mutate(across(1:6, as.numeric))
    }


    return(final_data)
  }
```

Moving on, we define a filtering function to extract data for a specific vintage, based on the selected vintage year and quarter. This function will take in the vintage year and vintage quarter and output the filter table with the same structure, with the additional columns that we need such as current vintage gdp level (current_vintage) and vintage growth (current_growth)

```
filter_function <- function(v_year1, v_quarter1) {
    cleaned_data() %>% filter(v_year == v_year1,
                              v_quarter == v_quarter1) %>%

      mutate(lag_current_vintage = lag(current_vintage,1),
             log_lag_current_vintage = log(lag_current_vintage),
             current_growth = 400*(log_current_vintage - log_lag_current_vintage))

  }
```

Due to missing data from earlier years in some vintages, we begin all analyses from 1965, which is also the first available vintage in our dataset. However, this introduces a challenge: if a user selects a forecast starting point close to 1965, the model will have limited historical data to train on. This lack of training data may negatively impact model performance and forecast reliability. One solution would be to restrict the range of vintages that users can choose to allow a certain level of training size. This would be a potential area for further research to determine the suitable range.

## 2. Model Construction

### 2.1. Autoregressive (AR)

We choose our baseline model to be the AR model. The AR model serves well as the baseline model because it is a relatively intuitive model. The idea behind the choice is that past growths (lags) of GDP could be used to predict GDP growth in the next period.

The explanation for the function is as follow: we create a matrix named aux that involves all the lags needed according to the user input by embedding Y p+h times. The first column is then extracted into a separate variable y as the dependent variable, while the lags needed are put into a separate matrix named X. These serve as the training data. The next part of the function determines what lags are used as actual predictors for the GDP growth that we want. If we are doing one-step forecast (h=1), we retrieve the last p observations as predictors. Otherwise, we delete h-1 columns before retrieving the last p observations so that the appropriate lags are used as predictors. These are stored in a matrix named X.out. Then, we run OLS regression with the dependent variable y and independent variable X and extract the coefficients. This will then be used to determine the prediction.

```r
fitARp=function(Y,p,h){

    #Inputs: Y- predicted variable,  p - AR order, h -forecast horizon
    aux=embed(Y,p+h) #create p lags + forecast horizon shift (=h option)
    y=aux[,1] #  Y variable aligned/adjusted for missing data due to lags
    X=as.matrix(aux[,-c(1:(ncol(Y)*h))]) # lags of Y corresponding to forecast horizon
    if(h==1){
      X.out=tail(aux,1)[1:ncol(X)] #retrieve last p observations if one-step forecast
    }else{
      X.out=aux[,-c(1:(ncol(Y)*(h-1)))] #delete first (h-1) columns of aux,
      X.out=tail(X.out,1)[1:ncol(X)] #last p observations to predict T+1
    }

    model=lm(y~X) #estimate direct h-step AR(p) by OLS
    coef=coef(model) #extract coefficients
    #make a forecast using the last few observations: a direct h-step forecast.
    pred=c(1,X.out)%*%coef

    return(list("pred"=pred))
  }
```

After we have implement our function, we define another function to run our forecast procedure through all the forecasting time point.

```r
run_ar <- function(h, p) {
    # A dataframe to store the result
    results <- data.frame('year' = numeric(0),
                          'quarter' = numeric(0),
                          'cur_pred' = numeric(0),
                          'latest_pred' = numeric(0),
                          'latest_growth' = numeric(0))


    for (i in 1:nrow(forecast_list())) {
      item <- forecast_list()[i,]
      year_f <- item$year
      quarter_f <- item$quarter
      # Filter for correct vintage data
      final_data <- filter_function(v_year1 = year_f, v_quarter1 = quarter_f) %>%
        select(year, quarter, current_growth) %>%
        right_join(latest_vintage_data(), by = c('year', 'quarter')) %>%
        filter(year >= 1965, year <= year_f) %>% drop_na() %>%
        select(year,quarter, current_growth, latest_growth)
      # Data process to input into model
      Y_cur <- as.matrix(final_data %>% pull(current_growth))
      Y_latest <- as.matrix(final_data %>% pull(latest_growth))
      # Fitting the model
      model_cur <- fitARp(Y = Y_cur, p = p, h = h) # Current Vintage
      model_latest <- fitARp(Y = Y_latest, p = p, h = h) # Latest Vintage
      pred_cur <- model_cur$pred[1]
      pred_latest <- model_latest$pred[1]
      # Assemble the result
      result_df <- data.frame("year" = year_f, "quarter" = quarter_f,
                              "cur_pred" = pred_cur, "latest_pred" = pred_latest)
      result_df <- result_df %>% left_join(actual_data(), by = c("year", "quarter")) %>%
        rename("latest_growth" ="value")
      results <- rbind(results, result_df)
    }
    return(results)
  }
```

### 2.2. Autoregressive Distributed Lag (ADL)

The workflow for building the ADL model as follow:

First, we will construct a dataframe for the chosen vintage that contains GDP growth, GDP growth lags, accompany with the features data following their own lags, this will be built using a get_adl_data function. To view how we construct our dataframe for the adl model, please view the Appendix.

Next, we define a function that will actually do the prediction using the vintage data and the current data. The prediction_adl function here attaches lags for the GDP growth of the real-time vintage, the GDP growth most recent vintage as well as each of the features data. Then, similar to the AR model, we choose the relative lags according to the user input of p and forecast horizon h. There are two sets of predictors. X_var_cur uses lags of the real_time vintage and the lags of the feature data, while X_var_latest uses lags of the latest vintage and lags of the feature data. The reason for this is because originally we want to use the corresponding vintage for the feature data as well. But since we have decided to use the latest available feature data across all analysis, the X_var_cur and X_var_latest here is basically the same. Then we run two separate OLS regressions to get the two separate prediction results.

```
prediction_adl <- function(h, data, features, lags, lag_y, v_year1, v_quarter1) {
    start_idx <- 1965*4
    end_idx <- v_year1 * 4 + (v_quarter1 - 1)
    n <- end_idx - start_idx + 1
    for (i in 1:lag_y) {
      y_lag <- lag(data$current_growth, h-1+i)

      y_lag_df <- data.frame('year' = data$year, 'quarter' = data$quarter,
                             setNames(data.frame(c(y_lag)), paste0("current_growth_lag", h-1+i
      data <- data %>% full_join(y_lag_df, by = c("year", "quarter"))
    }
    for (i in 1:lag_y) {
      y_lag_latest <- lag(data$latest_growth, h-1+i)

      y_lag_latest_df <- data.frame('year' = data$year, 'quarter' = data$quarter,
                             setNames(data.frame(c(y_lag_latest)), paste0("latest_growth_lag",
      data <- data %>% full_join(y_lag_latest_df, by = c("year", "quarter"))
    }
    for (i in 1:length(features)) {
      feature <- features[i]
      lag <- lags[i]

      for (j in 1:lag) {


        latest_lag_feature <- lag(data[[paste0(feature)]], h-1+j)
        latest_lag_feature_df <- data.frame('year' = data$year, 'quarter' = data$quarter,
                                            setNames(data.frame(c(latest_lag_feature)), paste0
        data <- data %>% full_join(latest_lag_feature_df, by = c('year', 'quarter'))
      }

    }



    data <- data %>% filter(year >= 1965) %>% mutate
    data <- data[1:n,]
```

```r
    # Current Prediction
    x_var <- names(data)
    x_var_cur <- x_var[c(5:(4+lag_y), (5+2*lag_y):length(x_var))]
    x_var_cur <- x_var_cur[x_var_cur %in% names(data)]
    x_var_latest <- x_var[c(5+lag_y:length(x_var))]
    x_var_latest <- x_var_latest[x_var_latest %in% names(data)]


    formula1 <- as.formula(paste('current_growth', "~", paste(x_var_cur, collapse = " + ")))
    model1 <- lm(data = data %>% drop_na(), formula = formula1)
    latest_input1 <- data %>% filter(year == v_year1, quarter == v_quarter1) %>% select(!!!syms
    pred1 <- predict(model1, newdata = latest_input1)

    # Latest Prediction

    formula2 <- as.formula(paste('latest_growth', "~", paste(x_var_latest, collapse = " + ")))
    model2 <- lm(data = data %>% drop_na(), formula = formula2)
    latest_input2 <- data %>% filter(year == v_year1, quarter == v_quarter1) %>% select(!!!syms
    pred2 <- predict(model2, newdata = latest_input2)


    result <- data.frame('year' = v_year1, 'quarter' = v_quarter1, 'cur_pred' = pred1, 'latest_
    result <- result %>% left_join(actual_data(), by = c('year', 'quarter'))

    return(result)
  }
```

Lastly, we define a run_adl_model to generate all the forecast for the forecasting period, similar to how we do in the AR section

```r
run_adl_model <- function(h,features, lags, lag_y, units) {
    results <- data.frame('year' = numeric(0),
                          'quarter' = numeric(0),
                          'cur_pred' = numeric(0),
                          'latest_pred' = numeric(0),
                          'latest_growth' = numeric(0))
    for (i in 1:nrow(forecast_list())) {
      item <- forecast_list()[i,]
      year_f <- item$year
      quarter_f <- item$quarter
      dat <- get_adl_data(features = features,v_year2 = year_f,
                          v_quarter2 = quarter_f, units = units)
      result_df <- prediction_adl(h = h, data = dat, features = features, lags = lags,lag_y = l
                                  v_quarter1 = quarter_f)
      results <- rbind(results, result_df)

    }
    return(results)
  }
```

### 2.3. K Nearest Neighbours (KNN)

Our third model is K-Nearest Neighbors (KNN). KNN regression has been widely studied for time series forecasting, and research has consistently shown it to perform well in capturing nonlinear patterns in the data (Lora et al., 2007; Zhang et al., 2017) We choose KNN to explore the self-explanatory power of GDP growth and compare this with parameter tuning models. We utilized the tsfknn package for using KNN regression for time-series forecast. More information on the package can be found here: https://github.com/franciscomartinezdelrio/tsfknn

To generate an h-step ahead forecast, we use data only up to time T-h. This approach ensures that our model does not unintentionally peek into the future. We construct lag-based features by using the first four lags of the target variable as autoregressive inputs. Since we are forecasting multiple steps ahead (rather than just the next time point), we also need to specify a multi-step ahead strategy. In our case, we use the Multiple Input Multiple Output (MIMO) method, which allows the model to generate all future predictions in one step. This avoids the compounding error problem commonly encountered in recursive forecasting. Another parameter that is needed to be specified is the cf, the method to aggregate the target from their nearest neighbors. Some

Then we extract the latest forecast which will be at time T for our purpose. The loop will run through all forecast point in the forecast list, generate forecast and assemble the result for evaluation.

```
run_prediction_knn <- function(h, k, cf) {
    results = data.frame(year = numeric(0), quarter = numeric(0), cur_forecast = numeric(0), la
    for (i in 1:nrow(forecast_list())) {
      item = forecast_list()[i,]
      year_f = item$year
      quarter_f = item$quarter
      data <- filter_function(v_year1 = year_f, v_quarter1 = quarter_f)
      %>% left_join(latest_vintage_data(), by = c("year", "quarter")) %>%
        select(year, quarter, current_growth, latest_growth) %>% filter(year >= 1965)
      data <- data[1:(nrow(data)-h+1),] # Extract data up until T - h
      train_data_cur <- data %>% select(-latest_growth)
      train_data_latest <- data %>% select(-current_growth)
      ts_train_cur <- ts_transform(train_data_cur)
      ts_train_latest <- ts_transform_latest(train_data_latest)
      model_current <- knn_forecasting(ts_train_cur, h = h,
                                       lags = 1:4, k = k,
                                       msas = "MIMO", cf = cf)
      cur_pred <- tail(model_current$prediction, 1)
      model_latest <-  knn_forecasting(ts_train_latest, h = h,
                                       lags = 1:4, k = k,
                                       msas = "MIMO", cf = cf)
      latest_pred <- tail(model_current$prediction, 1)
      results[i, 'year'] = year_f
      results[i, 'quarter'] = quarter_f
      results[i, 'cur_forecast'] = cur_pred
      results[i, 'latest_forecast'] = latest_pred

    }
    return(results)
  }
```

### 3. Visualization

We mainly use ggplot2 as our visualization tool.

Since we previously split the year and quarter for filtering, we now need to recombine them into a date format for meaningful plotting. To do this, we use the zoo package's yearqtr() function, which converts a year-quarter string into a proper quarterly object. We then convert it into a date object for visualization purposes. An example of this conversion for x aesthetic mapping in ggplot.

```
x = as.Date(zoo::as.yearqtr(paste0(df$year, " Q", df$quarter))
```

### Team contribution

Model Construction: Nguyen Duc Thinh, Lu Yi
Technical Video: Nguyen Duc Thinh, Lu Yi
Report Writing: Nguyen Duc Thinh, Lu Yi

## III. Application Frontend

The layout of our application will consist of two component: Dataset and Model. We chose the Journal theme for a clean and professional look, with a neutral gray tone that keeps the focus on the content. Fonts of text are choosen to make everything easy to read and visually appealing, helping users navigate the app more comfortably. We also avoid having two much content on each tab as not overload user with too much information.

### 1. Dataset Tab

This is where user choose their dataset as well specifying the range of forecast period they are interested in.

#### 1.1 Side Panel

To ensure meaningful forecasting results, we want the forecasting period to include a sufficient number of out-of-sample observations. In this case, we set a minimum of 30 out-of-sample points. Thus, we define a function that can determine the minimum ending period given the chosen starting period. Then can dynamically update the ending period that is allowed based on user input for the starting period. The code for this two function can be found in the Appendix

### 1.2 Main Panel

The main panel will preview the vintage data from the starting period as demonstration the structure of the cleaned data as well a visualization of the current vintage and latest vintage data. User can switch to the growth rate by clicking on the Growth button in the setting sidebar

## 2. Model Tab

This is where user can set up their models and features and view the evaluation results.

### 2.1 Side Panel

**Model Setup** In order to get the result for right model, we assign an id to each model to keep track of the chosen model.

```
#Server code
output$all_models_ui <- renderUI({
    lapply(model_ids(), function(i) {
      wellPanel(
        h4(paste("Model", i)),

        # Model type selection
        selectInput(paste0("model_type_", i), "Choose a model:",
                    choices = c("KNN", "AR", "ADL")),
        # Placeholder for model-specific parameters
        uiOutput(paste0("model_params_", i)),
        # Delete button
        actionButton(paste0("delete_model_", i), "Delete", class = "btn-danger")
      )
    }) %>% tagList()
  })
```

We also need to the parameters to be provided to the correct model by assigning it with the corresponding model id.

```
# Server code
observe({
    lapply(model_ids(), function(i) {
      output[[paste0("model_params_", i)]] <- renderUI({
        model_type <- input[[paste0("model_type_", i)]]
        req(model_type)

        if (model_type == "KNN") {
          tagList(textInput(paste0("knn_k_",i), "Enter a number or a comma-separated vector:",
                  selectInput(paste0("knn_cf_", i), "CF", choices = c("mean","median", "weight
        } else if (model_type == "AR") {
          tagList(numericInput(paste0("ylag_ar_",i), "Enter how many lag for Y", value = 1, mi
        } else if (model_type == "ADL") {
          tagList(numericInput(paste0("ylag_adl_",i), "Enter how many lag for Y", value = 2, mi
        }
      })
    })
  })
```

This id will be later be used to pull the correct results with corresponding model

### 2.2 Main Panel

The most important thing for our application is to run the model. We will create placeholders to store the model, prediction result and prediction performance when we loop through the model_id list and run the corresponding model.

```
modeltype = reactiveVal(list())
results = reactiveVal(list())
performance = reactiveVal(list())
```

For AR model and KNN model, since both model does not required additional features to be tune in, we only need to retrieve the parameters for each model and return the result. An example for retrieving the result for the KNN model can demonstrated as follow

```
# Server code

if (model_type == "KNN") {
        k_input <- input[[paste0("knn_k_", i)]]
        k <- as.numeric(unlist(strsplit(as.character(k_input), ",")))
        cf <- input[[paste0('knn_cf_', i)]]

        knn_result_df <- run_prediction_knn(h = h, k = k, cf = cf) %>%
          left_join(latest_vintage_data(), by = c('year', 'quarter')) %>%
          select(year,quarter, cur_forecast, latest_forecast, latest_growth) %>%
          rename("cur_pred" = "cur_forecast",
                 "latest_pred" = "latest_forecast")

        knn_performance_df <- knn_result_df %>%
            summarize(cur_mae = round(mean(abs(cur_pred - latest_growth)),2),
                      cur_rmse = round(sqrt(mean((cur_pred - latest_growth)^2)),2),
                      latest_mae = round(mean(abs(latest_pred - latest_growth)),2),
                     latest_rmse = round(sqrt(mean((latest_pred - latest_growth)^2)),2))
        model_lst <- append(model_lst, list(model_type))
        res_lst <- append(res_lst, list(knn_result_df))
        per_lst <- append(per_lst, list(knn_performance_df))
```

For ADL model, we will need to retrieve the feature as well its lags and transformation method. For more information on how features is retrieved, please refer to Appendix.

```r
# Server code
else if (model_type == "ADL") {
      ylag_input_adl <- input[[paste0("ylag_adl_", i)]]
      if (length(feature_series_ids()) == 0) {
      showNotification("Please add a feature for ADL or use AR model instead."
                           , type = "error")
          return()
        }
      invalid_ids <- feature_series_ids()[!sapply(feature_series_ids(),
                                                  is_valid_series)]

      # Warning when user input an invalid feature id
      if (length(invalid_ids) > 0) {
        showNotification(paste("The following series IDs are invalid:", paste(invalid_ids, coll
          return()
        }
      features <- c()
      feature_lags <- c()
      feature_transforms <- c()
      for (feature in feature_series_ids()) {
        features <- c(features,feature)
      }
      for (feature_lag in feature_lag_series()) {
         feature_lags <- c(feature_lags, feature_lag)
      }
      for (feature_transform in feature_transform_series()) {
        feature_transforms <- c(feature_transforms, feature_transform)
      }

      adl_result_df <- run_adl_model(h = h, features = features, lag_y = ylag_input_adl, lags =
          rename("latest_growth" = "value")
      adl_per_df <- adl_result_df %>%
        summarize(cur_mae = round(mean(abs(cur_pred - latest_growth)),2),
                  latest_mae = round(mean(abs(latest_pred - latest_growth)),2),
                  cur_rmse = round(sqrt(mean((cur_pred - latest_growth)^2)),2),
                  latest_rmse =round(sqrt(mean((latest_pred - latest_growth)^2)),2))
      model_lst <- append(model_lst, list(model_type))
      res_lst <- append(res_lst, list(adl_result_df))
      per_lst <- append(per_lst, list(adl_per_df))
      }
```

Now that we have all the results we need, the remaining work is to output the results with the correct id. Please refer to Appendix for more information. Here we have the Forecast tab (for prediction result), Comparison tab (for compare the performance between models) and a Visualization tab (for displaying the prediction with the real gdp growth).

**Team Contribution**

UI development and integration: Nguyen Duc Thinh
UI design: Nguyen Duc Thinh, Tan Win Yi
Pitch Video: Nguyen Duc Thinh, Tan Win Yi
Report Writing: Nguyen Duc Thinh, Tan Win Yi

## IV. Challenges and Future improvement

We want to highlight three improvements that we want to address for our application. ## 1. Development Tool
One key limitation of our current application is the limited range of forecasting models available, especially as forecasting techniques continue to evolve rapidly. Rather than attempting to include every possible model, our next step is to introduce a Development Tab, allowing users to integrate their own models by following a simple input-output structure.
## 2. Forecast
Additionally, while model evaluation is important, the ultimate goal of any forecasting model is to generate forecasts. Therefore, we plan to extend the workflow to include actual out-of-sample forecasting after evaluation, providing users with more actionable insights. ## 3. Application Performance
Allowing the user more freedom to choose multiple features to add into the ADL model involves tradeoff with machine performance. Despite the fact that prediction results could be improved by adding more features since we have more information available, the app runs slower and slower as more features are added. This is due to our ineffective data pre-processing for model training. In addition, we believe there are many redundant and repetitive steps in our application construction. Therefore, it is necessary for us to figure out a universal method to prepare data for training for multiple models to avoid this issue.

## V. Conclusion

We present a simple tool to compare performance of different forecasting model to quantify the impact of data revision on our forecasting model. I We want to highlight the importance of evaluating models using both real-time and latest vintage data to ensure a more realistic assessment of how models perform under conditions faced by actual forecasters at the time of prediction

## VI. Appendix

### 1. Preparing ADL data

For ADL model, in addition to its lag value, we also need to preprocess the feature data that will be used as additional independent variables. First, we need a function to extract the feature data of our interest. Unlike GDP, economic indicators go through very little revision, thus we will retrieve the latest available quarterly data.

```
get_feature_data <- function(series_id, v_year, v_quarter, units = 'lin') {
    result_df <- fredr_series_observations(series_id = series_id,
                                           units = units,
                                           frequency = 'q',
                                           aggregation_method = 'avg')
    return(result_df)
  }
```

Here we define to helper function to convert the retrieved feature data into our desire format: - date_to_year_quarter: this help convert a Date object into a dataframe with the year and quarter seperately - clean_feature: this takes in the feature data we retrieved and turn it into a dataframe that match our data format

```
date_to_year_quarter <- function(date) {
    date <- as.Date(date)  # ensure it's a Date object
    year <- as.numeric(format(date, "%Y"))
    month <- as.numeric(format(date, "%m"))
    quarter <- ceiling(month / 3)
    data.frame(year = year, quarter = quarter)
  }
  clean_feature <- function(feature_data) {
    result <- feature_data %>%
      mutate(year = date_to_year_quarter(date)$year,
             quarter = date_to_year_quarter(date)$quarter) %>%
      select(year,quarter, series_id, value) %>%
      pivot_wider(names_from = series_id, values_from = value)
    return(result)
  }
```

Then we merge the feature data, into our GDP data

```
get_adl_data <- function(features,  v_year2, v_quarter2, units) {
    final_data <- filter_function(v_year1 = v_year2, v_quarter1 = v_quarter2) %>%
      select(year, quarter, current_growth) %>% right_join(latest_vintage_data(), by = c('year
      select(year,quarter, current_growth, latest_growth)
    for (i in 1:length(features)) {
      latest_feature_data <- clean_feature(get_feature_data(series_id = features[i],
                                                            v_year = 2025,
                                                            v_quarter = 2,

                                                            units = units[i]))
      final_data <- final_data %>% full_join(latest_feature_data, by = c('year', 'quarter')) %
                    mutate(across(everything(), ~replace_na(.x, 0)))
    }
    final_data <- final_data %>% filter(year >= 1963, year <= v_year2)

    return(final_data)
  }
```

## 2. Vintage setting and Forecasting period (Dataset Tab)

As mentioned earlier, our initial plan was to support both quarterly and monthly data. Therefore, we need to detect the data frequency and adjust the vintage selection options accordingly.

```
output$vintage_period_ui <- renderUI({
    req(data_frequency())

    if (data_frequency() == "quarterly") {
      selectInput("vintage_period", "Select Starting Quarter:", choices = 1:4, selected = 1)
    } else if (data_frequency() == "monthly") {
      selectInput("vintage_period", "Select Vintage Month:", choices = 1:12, selected = 1)
    }
  })
```

Thus, you may find instead of using calling input$vintage_{q}uarter, wecalledinput$vintage_period when pulling the starting quarter as inputs. But since we our analysis only conduct on quarterly data, you can assume vintage_period here is vintage_quarter

The helper function to determine the minimum ending period given the starting date is as follow

```
# Server code
min_end_date_from_start <- function(start_year, start_quarter, min_quarters = 30) {
    start_year <- as.numeric(start_year)
    start_quarter <- as.numeric(start_quarter)
    start_index <- start_year * 4 + (start_quarter - 1)
    end_index <- start_index + (min_quarters - 1)
    max_index <- 2024 * 4 + 3
    end_index <- min(end_index, max_index)
    end_year <- end_index %/% 4
    end_quarter <- (end_index %% 4) + 1
    list(year = end_year, quarter = end_quarter)
  }
```

With this function, we can dynamically update the ending period to ensure our minimum requirement of forecasting points

```
# Server code
observeEvent({
    input$vintage_year
    input$vintage_period
  }, {
    req(input$vintage_year, input$vintage_period)

    min_date <- min_end_date_from_start(as.numeric(input$vintage_year), as.numeric(input$vintag

    # Render end year input
    output$end_year_ui <- renderUI({
      numericInput(
        "end_year",
        "Select End Year (for forecast)",
        value = min_date$year,
        min = min_date$year,
        max = 2024
      )
    })
    # Render end quarter input
    output$end_quarter_ui <- renderUI({
      selectInput(
        "end_quarter",
        "Select End Quarter (for forecast)",
        choices = 1:4,
        selected = min_date$quarter)
    })
  })
```

```
# UI code
numericInput(inputId = "vintage_year", label = "Select Starting Year",
                                      min = 1965, max = 2024, value = 2000),
                          uiOutput("vintage_period_ui"),
                          uiOutput("end_year_ui"),
                          uiOutput("end_quarter_ui")
```

## 3. Feature selection

For feature selection we also assign an id. This is because each feature also have different parameters to tune in. Then we store them into a list for later retrieval

```
# Server code
output$all_features_ui <- renderUI({
    lapply(feature_ids(), function(i) {
      wellPanel(
        h4(paste("Feature", i)),
        textInput(paste0("feature_id_", i), label = "Feature Series ID"),
        numericInput(paste0("feature_lag_id_", i), label = "Lags (optional)", value = 1, min =
        textInput(paste0("feature_transform_id_", i), label = "Transformation (optional)", valu
        actionButton(paste0("delete_feature_", i), "Delete", class = "btn-danger")
      )
    }) %>% tagList()
  })
```

### 4. Model Outputs

First is to output the forecast results

```
output$resulttable <- renderUI({
  req(results(), modeltype())
  req(length(results()) > 0)

  res <- results()
  type <- modeltype()

  output_list <- lapply(seq_along(res), function(i) {
    output_id <- paste0("result_", i)

    output[[output_id]] <- DT::renderDataTable({
      req(res[[i]])
      res[[i]]
    })

    tagList(
      h4(paste("Model", i, "-", type[[i]])),
      DT::DTOutput(output_id),
      tags$hr()
    )
  })
```

Second is to rank the model performance in the Comparison Tab. Here we also do simple bar chart to compare the RMSE and MAE between each model.

```
## Combine and rank performance
perf_df <- do.call(rbind, lapply(seq_along(per_lst), function(i) {
      df <- per_lst[[i]]
      model <- model_lst[[i]]
      data.frame(
        model = model,
        cur_rmse = df$cur_rmse[1],
```

```r
        cur_mae = df$cur_mae[1],
        latest_rmse = df$latest_rmse[1],
        latest_mae = df$latest_mae[1]
      )
    }))

# Create ranking dataframes
rank_cur <- perf_df %>%
      arrange(cur_rmse) %>%
      mutate(rank = row_number()) %>%
      select(rank, model, cur_rmse, cur_mae)

rank_latest <- perf_df %>%
      arrange(latest_rmse) %>%
      mutate(rank = row_number()) %>%
      select(rank, model, latest_rmse, latest_mae)

    performance(list(cur = rank_cur, latest = rank_latest))


# Comparison Tab
  output$comparison <- renderUI({
    req(performance())

    output$cur_perf_table <- DT::renderDataTable({
      performance()$cur
    })


    output$latest_perf_table <- DT::renderDataTable({
      performance()$latest
    })
    # Bar Chart: Current Forecast
    output$cur_perf_plot <- renderPlot({
      req(length(performance()) > 0)
      cur_df <- performance()$cur

      cur_df_long <- tidyr::pivot_longer(
        cur_df,
        cols = c("cur_rmse", "cur_mae"),
        names_to = "metric",
        values_to = "value"
      )

      ggplot(cur_df_long, aes(x = reorder(model, value), y = value, fill = metric)) +
        geom_bar(stat = "identity", position = position_dodge(width = 0.8), width = 0.5) +
        scale_fill_manual(values = c("cur_rmse" = "steelblue", "cur_mae" = "yellow2")) +
        labs(
          title = "Current Forecast: RMSE and MAE",
          x = "Model",
          y = "Error Value",
          fill = "Metric"
        ) +
        theme_minimal()

    })

    #Bar Chart: Latest Forecast
    output$latest_perf_plot <- renderPlot({
      req(length(performance()) > 0)
      latest_df <- performance()$latest

      latest_df_long <- tidyr::pivot_longer(
        latest_df,
        cols = c("latest_rmse", "latest_mae"),
        names_to = "metric",
        values_to = "value"
      )

      ggplot(latest_df_long, aes(x = reorder(model, value), y = value, fill = metric)) +
        geom_bar(stat = "identity", position = position_dodge(width = 0.8), width = 0.5) +
        scale_fill_manual(values = c("latest_rmse" = "steelblue", "latest_mae" = "yellow2")) +
        labs(
          title = "Latest Forecast: RMSE and MAE",
          x = "Model",
          y = "Error Value",
          fill = "Metric"
        ) +
        theme_minimal()

    })

    tagList(
      h6("Model Performance Comparison"),

      tabsetPanel(
        tabPanel("Current Forecast",
                 plotOutput("cur_perf_plot"),
                 DT::DTOutput("cur_perf_table")),
        tabPanel("Latest Forecast",
                 plotOutput("latest_perf_plot"),
                 DT::DTOutput("latest_perf_table"),)
      )
    )
```

```
  })
```

Last is the Visualization Tab

```
# Visualization Tab
  output$modelplot <- renderUI({
    req(results())
    res <- results()
    types <- modeltype()

    output_list <- lapply(seq_along(res), function(i) {
      plot_id <- paste0("model_plot_", i)

      output[[plot_id]] <- renderPlot({
        df <- res[[i]]

        ggplot(df, aes(x = as.Date(zoo::as.yearqtr(paste0(df$year, " Q", df$quarter))))) +
          geom_line(size = 1,aes(y = cur_pred, color = "Current Forecast")) +
          geom_line(size = 1,aes(y = latest_pred, color = "Latest Forecast")) +
          geom_line(size = 1,aes(y = latest_growth, color = "Actual Growth"), linetype = "dashe
          scale_color_manual(values = c(
            "Current Forecast" = "blue",
            "Latest Forecast" = "indianred",
            "Actual Growth" = "black")) +
          scale_x_date(labels = function(x) zoo::format.yearqtr(x, "%YQ%q")) +
          labs(
            x = "Quarter", y = "GDP Growth (%)", color = NULL
          ) +
          theme_classic() +
          theme(legend.position = "top")
      })

      tagList(
        h4(paste("Model", i, "-", types[[i]])),
        plotOutput(plot_id),
        tags$hr()
      )
    })

    do.call(tagList, output_list)
  })
```