APEC 8221 - Assignment 2: Automation and API Data Acquisition

Submission File

[YunshuYang]

2025-09-22

Table of contents

GitHub Repository URL: https://github.com/yunshuyang1129/apec8221-yang9291.git

1 Build Your API Function

```
# Load required packages
library(tidyverse)
library(httr2)
library(jsonlite)

# Secure API key storage - load from .Renviron file
# usethis::edit_r_environ()
fred_api_key <- Sys.getenv("FRED_API_KEY")
print(fred_api_key)
```

```
[1] "7a4920f09141fa8521fed8a425d635f7"
```

1.1 API Setup and Testing

```
# Test basic API connectivity
# Your code here to test the FRED API connection

# Set the base API URL
base_url <- "https://api.stlouisfed.org/fred/series/observations"

# Build request properly with httr2
test_url <- paste0(
  base_url, "?",
  "series_id=UNRATE",
  "&api_key=", fred_api_key,
  "&file_type=json"
```

```
)

# Send the request
response <- request(test_url) %>% req_perform()

# Check status (200 = success)
resp_status(response)
```

```
[1] 200
```

```
# Parse JSON response
raw_data <- response %>% resp_body_json()
str(raw_data, max.level = 2)
```

```
List of 13
 $ realtime_start   : chr "2025-09-05"
 $ realtime_end     : chr "2025-09-05"
 $ observation_start: chr "1600-01-01"
 $ observation_end  : chr "9999-12-31"
 $ units            : chr "lin"
 $ output_type      : int 1
 $ file_type        : chr "json"
 $ order_by         : chr "observation_date"
 $ sort_order       : chr "asc"
 $ count            : int 932
 $ offset           : int 0
 $ limit            : int 100000
 $ observations     :List of 932
  ..$ :List of 4
  ..$ :List of 4
  ..$ :List of 4
  ..$ :List of 4
  ..$ :List of 4
  ..$ :List of 4
  ..$ :List of 4
  ..$ :List of 4
  ..$ :List of 4
  ..$ :List of 4
  ..$ :List of 4
  ..$ :List of 4
  ..$ :List of 4
```

```
..$ :List of 4
..$ :List of 4
..$ :List of 4
..$ :List of 4
..$ :List of 4
..$ :List of 4
..$ :List of 4
..$ :List of 4
..$ :List of 4
..$ :List of 4
..$ :List of 4
..$ :List of 4
..$ :List of 4
..$ :List of 4
..$ :List of 4
..$ :List of 4
..$ :List of 4
..$ :List of 4
..$ :List of 4
..$ :List of 4
..$ :List of 4
..$ :List of 4
..$ :List of 4
..$ :List of 4
..$ :List of 4
..$ :List of 4
..$ :List of 4
..$ :List of 4
..$ :List of 4
..$ :List of 4
..$ :List of 4
..$ :List of 4
..$ :List of 4
..$ :List of 4
..$ :List of 4
..$ :List of 4
..$ :List of 4
..$ :List of 4
..$ :List of 4
..$ :List of 4
..$ :List of 4
..$ :List of 4
..$ :List of 4
```

```
..$ :List of 4
..$ :List of 4
..$ :List of 4
..$ :List of 4
..$ :List of 4
..$ :List of 4
..$ :List of 4
..$ :List of 4
..$ :List of 4
..$ :List of 4
..$ :List of 4
..$ :List of 4
..$ :List of 4
..$ :List of 4
..$ :List of 4
..$ :List of 4
..$ :List of 4
..$ :List of 4
..$ :List of 4
..$ :List of 4
..$ :List of 4
..$ :List of 4
..$ :List of 4
..$ :List of 4
..$ :List of 4
..$ :List of 4
..$ :List of 4
..$ :List of 4
..$ :List of 4
..$ :List of 4
..$ :List of 4
..$ :List of 4
..$ :List of 4
..$ :List of 4
..$ :List of 4
..$ :List of 4
..$ :List of 4
..$ :List of 4
..$ :List of 4
..$ :List of 4
..$ :List of 4
..$ :List of 4
..$ :List of 4
```

```
.. [list output truncated]
```

1.2 Write Your Data Acquisition Function

```r
# Function to fetch state unemployment data from FRED

get_fred_unemployment <- function(state_code, api_key) {

  # 1. Input Validation
  if (!grepl("^[A-Z]{2}$", state_code)) {
    stop("Error: state_code must be a 2-letter uppercase abbreviation, e.g., 'CA'.")
  }

  # Construct the FRED series ID for the state
  # create series_id
  series_id <- paste0(state_code, "UR")

  # 2. API Request
  base_url <- "https://api.stlouisfed.org/fred/series/observations"
  response <- request(base_url) %>%
    req_url_query(
      series_id = series_id,
      api_key   = api_key,
      file_type = "json"
    ) %>%
    req_perform()

  # 3. Error Handling
  if (resp_status(response) != 200) {
    stop(paste("API request failed for state:", state_code))
  }

  # 4. JSON Parsing
  raw_data <- response %>% resp_body_json()

  # 5. Data Cleaning
  df <- raw_data$observations %>%
    map_df(~ tibble(
      date = as.Date(.x$date),
      unemployment_rate = as.numeric(.x$value),
      state = state_code
```

```
    ))

  return(df)
}
# Test your function

test_result <- get_fred_unemployment("CA", fred_api_key)
head(test_result)
```

```
# A tibble: 6 x 3
  date       unemployment_rate state
  <date>                 <dbl> <chr>
1 1976-01-01               9.2 CA
2 1976-02-01               9.2 CA
3 1976-03-01               9.1 CA
4 1976-04-01               9   CA
5 1976-05-01               8.9 CA
6 1976-06-01               8.9 CA
```

2 Automate Data Collection

2.1 Define Your State List

```
# Select 8-10 states from different regions
target_states <- c("CA", "WA",   # West Coast
                   "TX", "FL", "GA",  # South
                   "OH", "IL",        # Midwest
                   "NY", "MA",        # Northeast
                   "CO")              # Mountain

# Print your selected states
target_states
```

```
 [1] "CA" "WA" "TX" "FL" "GA" "OH" "IL" "NY" "MA" "CO"
```

2.2 Implement Rate Limiting

```r
# Enhanced version of get_fred_unemployment with rate limiting

get_fred_unemployment <- function(state_code, api_key, pause = 1) {

  # Input validation
  if (!grepl("^[A-Z]{2}$", state_code)) {
    stop("Error: state_code must be a 2-letter uppercase abbreviation, e.g., 'CA'.")
  }

  # Informative message
  message("Fetching unemployment data for state: ", state_code)

  # Construct the series ID
  series_id <- paste0(state_code, "UR")

  # Build request
  base_url <- "https://api.stlouisfed.org/fred/series/observations"
  response <- request(base_url) %>%
    req_url_query(
      series_id = series_id,
      api_key   = api_key,
      file_type = "json"
    ) %>%
    req_perform()

  # Error handling
  if (resp_status(response) != 200) {
    stop(paste("API request failed for state:", state_code))
  }

  # Parse JSON
  raw_data <- response %>% resp_body_json()

  # Clean data
  df <- raw_data$observations %>%
    map_df(~ tibble(
      date = as.Date(.x$date),
      unemployment_rate = as.numeric(.x$value),
      state = state_code
    ))

  # Rate limiting: pause
```

```
  Sys.sleep(pause)

  return(df)
}

# Collect data for all states in target_states
all_states_data <- map_dfr(target_states, ~ get_fred_unemployment(.x, fred_api_key))

head(all_states_data)
```

```
# A tibble: 6 x 3
  date       unemployment_rate state
  <date>                 <dbl> <chr>
1 1976-01-01               9.2 CA
2 1976-02-01               9.2 CA
3 1976-03-01               9.1 CA
4 1976-04-01               9   CA
5 1976-05-01               8.9 CA
6 1976-06-01               8.9 CA
```

2.3 Compare Iteration Methods

2.3.1 Method 1: `for` Loop Approach

```
# Initialize empty data frame
unemployment_data_loop <- data.frame()

# Loop through states
for (state in target_states) {

  # Print message
  cat("Downloading data for", state, "...\n")

  # Get data for this state
  state_data <- get_fred_unemployment(state, fred_api_key)

  # Add to main data frame
  unemployment_data_loop <- bind_rows(unemployment_data_loop, state_data)
}
```

```
Downloading data for CA ...

Downloading data for WA ...

Downloading data for TX ...

Downloading data for FL ...

Downloading data for GA ...

Downloading data for OH ...

Downloading data for IL ...

Downloading data for NY ...

Downloading data for MA ...

Downloading data for CO ...
```

```r
# Check results
head(unemployment_data_loop)
```

```
        date unemployment_rate state
1 1976-01-01               9.2    CA
2 1976-02-01               9.2    CA
3 1976-03-01               9.1    CA
4 1976-04-01               9.0    CA
5 1976-05-01               8.9    CA
6 1976-06-01               8.9    CA
```

2.3.2 Method 2: `lapply` Approach

```r
# Use lapply to get data for all states
unemployment_list <- lapply(target_states, function(state) {
  message("Downloading data for: ", state)
  get_fred_unemployment(state, fred_api_key)
})

# Combine into single data frame
unemployment_data_apply <- bind_rows(unemployment_list)

# Check results
head(unemployment_data_apply)
```

```
# A tibble: 6 x 3
  date       unemployment_rate state
  <date>                 <dbl> <chr>
1 1976-01-01               9.2 CA
2 1976-02-01               9.2 CA
3 1976-03-01               9.1 CA
4 1976-04-01               9   CA
5 1976-05-01               8.9 CA
6 1976-06-01               8.9 CA
```

2.3.3 Method 3: purrr::map Approach

```r
# Use purrr::map_dfr to get data for all states
unemployment_data_purrr <- target_states %>%
  map_dfr(~ {
    message("Downloading data for: ", .x)
    get_fred_unemployment(.x, fred_api_key)
  })

# Check results
head(unemployment_data_purrr)
```

```
# A tibble: 6 x 3
  date       unemployment_rate state
  <date>                 <dbl> <chr>
1 1976-01-01               9.2 CA
2 1976-02-01               9.2 CA
```

```
3 1976-03-01               9.1 CA
4 1976-04-01               9   CA
5 1976-05-01               8.9 CA
6 1976-06-01               8.9 CA
```

2.3.4 Method 4: Manual Approach (Demonstration)

```
# This is what we would have to do without automation:
ca_data <- get_fred_unemployment_safe("CA", fred_api_key)
tx_data <- get_fred_unemployment_safe("TX", fred_api_key)
ny_data <- get_fred_unemployment_safe("NY", fred_api_key)
# ... and so on for each state

# Then manually combine them:
# unemployment_data_manual <- rbind(ca_data, tx_data, ny_data, ...)
```

Iteration Method Comparison: [*Write 2-3 sentences comparing the three approaches. Which is most readable? Most efficient? Why is automation valuable?*]The for loop is intuitive and easy to follow, but the code is somewhat verbose. The lapply approach is more concise and reduces the need for manual binding, but it can be less readable for beginners. The purrr::map_dfr method is the most modern and elegant: it combines iteration and binding in a single step, making the code both compact and highly readable. In contrast, the manual calls approach is clearly inefficient, error-prone, and not scalable. Automation is valuable here because it reduces redundancy, minimizes the risk of errors, and allows the same code to scale easily from a few states to many.

Scaling Analysis: [*How would you modify your approach for analyzing unemploymentdata for all 50 states instead of just 8? What challenges would arise and how would eachiteration method handle increased scale?*]The for loop is intuitive and easy to follow, but the code is somewhat verbose. The lapply approach is more concise and reduces the need for manual binding, but it can be less readable for beginners. The purrr::map_dfr method is the most modern and elegant: it combines iteration and binding in a single step, making the code both compact and highly readable. In contrast, the manual calls approach is clearly inefficient, error-prone, and not scalable. Automation is valuable here because it reduces redundancy, minimizes the risk of errors, and allows the same code to scale easily from a few states to many.

---

3 Data Analysis and Visualization

```
# Use the data from your preferred method above
unemployment_data <- unemployment_data_loop  # or unemployment_data_apply or ...

# Check data structure
glimpse(unemployment_data)
```

```
Rows: 5,960
Columns: 3
$ date              <date> 1976-01-01, 1976-02-01, 1976-03-01, 1976-04-01, 197~
$ unemployment_rate <dbl> 9.2, 9.2, 9.1, 9.0, 8.9, 8.9, 9.0, 9.1, 9.3, 9.4, 9.~
$ state             <chr> "CA", "CA", "CA", "CA", "CA", "CA", "CA", "CA", "CA"~
```

3.1 Regional Unemployment Patterns

3.1.1 Basic Summary Statistics

```
# Calculate average unemployment rate by state (2019-2024)
unemployment_recent <- unemployment_data %>%
  filter(date >= as.Date("2019-01-01") & date <= as.Date("2024-12-31"))

state_averages <- unemployment_recent %>%
  group_by(state) %>%
  summarise(avg_rate = mean(unemployment_rate, na.rm = TRUE)) %>%
  arrange(desc(avg_rate))

# Identify state with highest peak unemployment
highest_peak <- unemployment_recent %>%
  group_by(state) %>%
  summarise(max_rate = max(unemployment_rate, na.rm = TRUE)) %>%
  arrange(desc(max_rate)) %>%
  slice(1)

# Find state with most volatility (highest std dev)
volatility_analysis <- unemployment_recent %>%
  group_by(state) %>%
  summarise(sd_rate = sd(unemployment_rate, na.rm = TRUE)) %>%
  arrange(desc(sd_rate)) %>%
```

```
  slice(1)


# Display results
print(state_averages)
```

```
# A tibble: 10 x 2
   state avg_rate
   <chr>    <dbl>
 1 CA        5.99
 2 NY        5.59
 3 IL        5.58
 4 WA        5.14
 5 OH        4.92
 6 TX        4.82
 7 MA        4.80
 8 CO        4.27
 9 FL        4.22
10 GA        4.01
```

```
print(highest_peak)
```

```
# A tibble: 1 x 2
  state max_rate
  <chr>    <dbl>
1 IL        18.3
```

```
print(volatility_analysis)
```

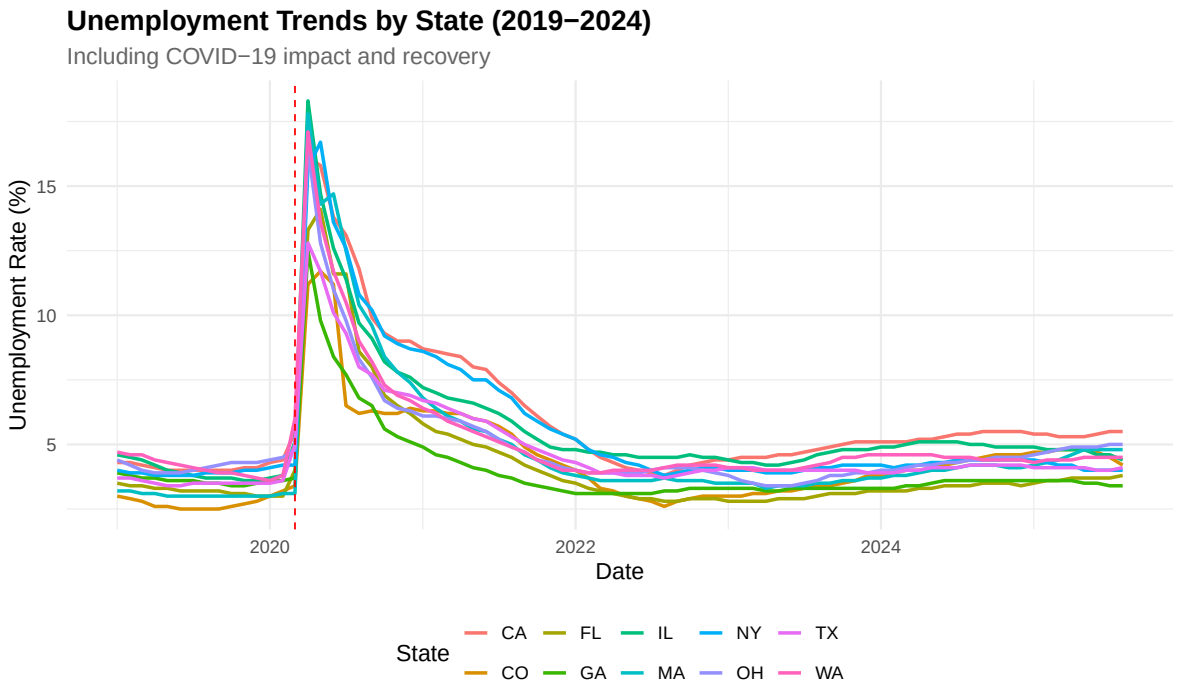
```
# A tibble: 1 x 2
  state sd_rate
  <chr>   <dbl>
1 MA       2.91
```

Key Findings: [*Summarize the main patterns you observe in the data*]Based on the summary statistics from 2019 to 2024, clear differences emerge across states in both average unemployment levels and patterns of volatility. California shows the highest average unemployment rate at about 5.99%, followed closely by New York and Illinois, while Georgia records the lowest average at just over 4%, suggesting a relatively stronger labor market. In terms of peak unemployment, Illinois experienced the sharpest spike with a maximum of 18.3%, indicating that

it was particularly exposed to severe labor market disruptions during this period. When examining volatility, measured by the standard deviation of unemployment rates, Massachusetts stands out with the highest level at around 2.91, meaning its unemployment rates fluctuated more dramatically than those of other states. Together, these findings highlight important regional contrasts: some states consistently face higher unemployment, some experience severe one-time shocks, and others undergo more unstable labor market conditions over time.

3.2 Time Series Visualization

```
# Create time series plot
unemployment_plot <- unemployment_data %>%
  filter(date >= as.Date("2019-01-01")) %>%
  ggplot(aes(x = date, y = unemployment_rate, color = state)) +
  geom_line(linewidth = 1) +
  geom_vline(xintercept = as.Date("2020-03-01"),
             linetype = "dashed", color = "red") +
  labs(
    title = "Unemployment Trends by State (2019-2024)",
    subtitle = "Including COVID-19 impact and recovery",
    x = "Date",
    y = "Unemployment Rate (%)",
    color = "State"
  ) +
  theme_minimal(base_size = 14) +
  theme(
    legend.position = "bottom",
    plot.title = element_text(face = "bold"),
    plot.subtitle = element_text(color = "gray40")
  )

# Print plot
print(unemployment_plot)
```

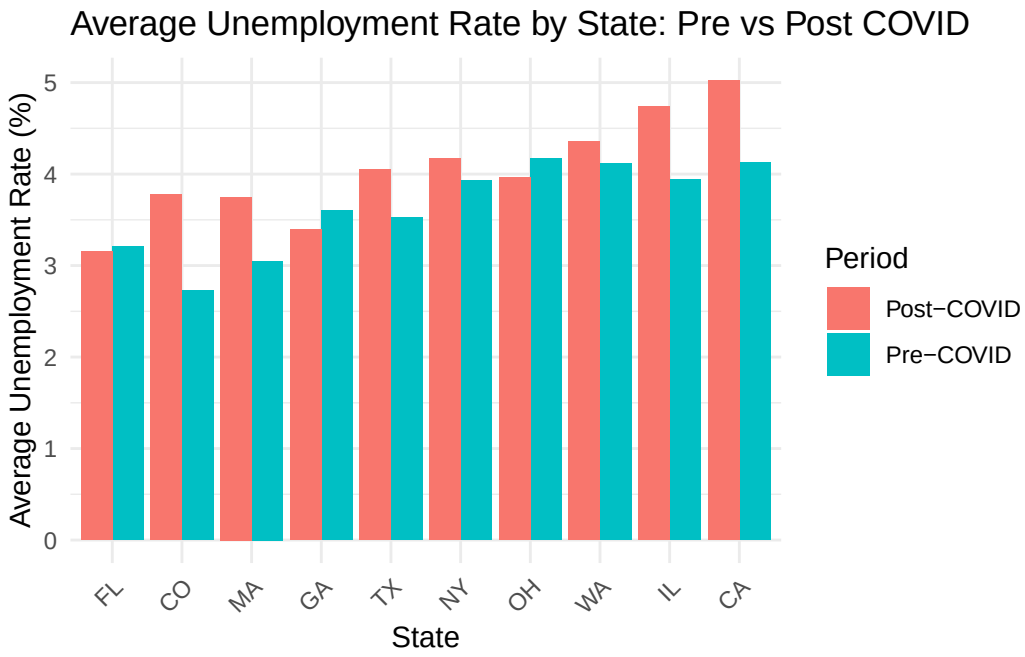


Visualization Insights: [*Describe what the plot reveals about regional unemployment patterns and COVID impact*]The time series plot shows that unemployment rates across all states were relatively stable between 2019 and early 2020, generally hovering around 3–5%. However, there was a sharp spike in unemployment in March 2020, corresponding to the onset of the COVID-19 pandemic, with some states reaching peaks above 15%. Following this surge, unemployment rates gradually declined, though the pace of recovery varied across states. By 2022–2024, most states had stabilized back to pre-pandemic levels, but some states such as California and New York exhibited slightly higher persistent unemployment compared to others. This pattern highlights both the common national shock of COVID-19 and the regional differences in labor market resilience and recovery.

3.3 State Comparison Analysis

```
# Define time periods
pre_covid <- unemployment_data %>%
  filter(date >= as.Date("2019-01-01"), date <= as.Date("2020-02-29")) %>%
  group_by(state) %>%
  summarise(avg_unemployment = mean(unemployment_rate, na.rm = TRUE)) %>%
  mutate(period = "Pre-COVID")
```

```r
post_covid <- unemployment_data %>%
  filter(date >= as.Date("2023-01-01"), date <= as.Date("2024-12-31")) %>%
  group_by(state) %>%
  summarise(avg_unemployment = mean(unemployment_rate, na.rm = TRUE)) %>%
  mutate(period = "Post-COVID")

# Combine periods for comparison
period_comparison <- bind_rows(pre_covid, post_covid)

# Create comparison bar chart
comparison_plot <- period_comparison %>%
  ggplot(aes(x = reorder(state, avg_unemployment),
             y = avg_unemployment,
             fill = period)) +
  geom_col(position = "dodge") +
  labs(title = "Average Unemployment Rate by State: Pre vs Post COVID",
       x = "State",
       y = "Average Unemployment Rate (%)",
       fill = "Period") +
  theme_minimal() +
  theme(axis.text.x = element_text(angle = 45, hjust = 1))

print(comparison_plot)
```



Pre/Post COVID Analysis: [*Compare how unemployment changed between periods across your selected states*]Across the selected states, unemployment rates were consistently higher in the post-COVID period (2023–2024) compared to the pre-COVID period (2019–early 2020). States such as California (CA) and Illinois (IL) show the largest increases, with average unemployment rates rising by more than one percentage point. In contrast, states like Florida (FL) and Georgia (GA) exhibit relatively smaller changes, suggesting stronger labor market resilience or faster recovery. Overall, the comparison highlights persistent elevation in unemployment rates across most states after the pandemic, though the magnitude of the impact varies regionally.

---

4 Extension and Critical Thinking

4.1 API Pagination Challenge

4.1.1 Step 1: Explore API Metadata

```
# Explore pagination with a limited request
response <- request("https://api.stlouisfed.org/fred/series/observations") %>%
```

```
  req_url_query(
    series_id = "CAUR",  # California unemployment
    api_key = fred_api_key,
    file_type = "json",
    limit = 100  # Small limit to see pagination
  ) %>%
  req_perform()

full_data <- response %>% resp_body_json()

# Examine metadata structure
str(full_data, max.level = 1)
```

```
List of 13
 $ realtime_start   : chr "2025-09-22"
 $ realtime_end     : chr "2025-09-22"
 $ observation_start: chr "1600-01-01"
 $ observation_end  : chr "9999-12-31"
 $ units            : chr "lin"
 $ output_type      : int 1
 $ file_type        : chr "json"
 $ order_by         : chr "observation_date"
 $ sort_order       : chr "asc"
 $ count            : int 596
 $ offset           : int 0
 $ limit            : int 100
 $ observations     :List of 100
```

Metadata Analysis: [What pagination information does the FRED API provide? What parameters control pagination? How many total observations are available? How many observations were retrieved in the first call?]The FRED API provides pagination information through three key parameters: count, limit, and offset. The count field indicates the total number of available observations (596 for California unemployment). The limit field specifies how many observations were requested per page (100 in this case), and the offset shows the starting index (0 here). In the first call, the API returned 100 observations out of the total 596, demonstrating how pagination works.

4.1.2 Step 2: Build Complete Data Function

```r
get_fred_unemployment_complete <- function(state_code, api_key, page_size = 1000) {

  # Step 1: Get metadata (to know total number of observations)
  base_url <- "https://api.stlouisfed.org/fred/series/observations"
  meta_response <- request(base_url) %>%
    req_url_query(
      series_id = paste0(state_code, "UR"),   # build series_id, e.g. CAUR
      api_key = api_key,
      file_type = "json",
      limit = 1                               # only need one obs to get metadata
    ) %>%
    req_perform()

  meta_data <- meta_response %>% resp_body_json()
  total_obs <- meta_data$count

  message("Total observations: ", total_obs)

  # Step 2: Calculate number of pages
  num_pages <- ceiling(total_obs / page_size)
  message("Need to fetch ", num_pages, " pages of data.")

  # Step 3: Loop through pages and collect data
  all_data <- list()
  for (i in 0:(num_pages - 1)) {
    offset_val <- i * page_size
    message("Fetching page ", i + 1, " of ", num_pages, " (offset = ", offset_val, ")")

    page_response <- request(base_url) %>%
      req_url_query(
        series_id = paste0(state_code, "UR"),
        api_key = api_key,
        file_type = "json",
        limit = page_size,
        offset = offset_val
      ) %>%
      req_perform()

    page_data <- page_response %>% resp_body_json()
    obs <- page_data$observations %>%
      map_df(~ tibble(
```

```
        date = as.Date(.x$date),
        unemployment_rate = as.numeric(.x$value),
        state = state_code
      ))

    all_data[[i + 1]] <- obs
  }

  # Step 4: Combine and return complete dataset
  result <- bind_rows(all_data)
  return(result)
}

# ---- Test the function ----
complete_ca_data <- get_fred_unemployment_complete("CA", fred_api_key, page_size = 500)
head(complete_ca_data)
```

```
# A tibble: 6 x 3
  date       unemployment_rate state
  <date>                 <dbl> <chr>
1 1976-01-01               9.2 CA
2 1976-02-01               9.2 CA
3 1976-03-01               9.1 CA
4 1976-04-01               9   CA
5 1976-05-01               8.9 CA
6 1976-06-01               8.9 CA
```

4.1.3 Step 3: Pagination Analysis

```
# Compare complete vs. recent data
# Average unemployment using complete data
avg_full <- complete_ca_data %>%
  summarise(avg_rate_full = mean(unemployment_rate, na.rm = TRUE))

# Average unemployment using recent data (2019-2024)
avg_recent <- complete_ca_data %>%
  filter(date >= as.Date("2019-01-01")) %>%
  summarise(avg_rate_recent = mean(unemployment_rate, na.rm = TRUE))

# Compare
```

```r
comparison <- bind_cols(avg_full, avg_recent)
print(comparison)
```

```
# A tibble: 1 x 2
  avg_rate_full avg_rate_recent
          <dbl>           <dbl>
1          7.11            5.93
```

```r
# Answer the required questions:
# 1. How many total observations for California?
# 2. How many API calls needed with page_size = 500?
# 3. Trade-offs between page sizes?
# 4. When to use complete vs. recent data?
```

Pagination Insights: [*Answer the four analysis questions about pagination trade-offs and use cases*]For the California unemployment series, the API metadata indicates a total of 596 observations. With a page size of 500, the function requires 2 API calls to retrieve the complete dataset. Larger page sizes reduce the number of requests and generally improve efficiency, but they may risk timeouts or memory issues if the server or client struggles with large responses. Smaller page sizes are safer and more flexible, though they increase the number of calls and can slow performance. Complete historical data is most useful for long-term trend analysis, model training, and research on structural changes, while recent data is better suited for real-time monitoring, dashboards, and short-term policy evaluation where speed and recency are more important than historical context.

4.2 Alternative API Practice

Selected API: [*Choose Option A (World Bank), Option B (BLS), Option C (Alpha Vantage), Option D (Census), or Option E (OpenWeatherMap) and identify which you selected*] Option A – World Bank API (International Development Data)

4.2.1 API Documentation Research

[*Write 2-3 sentences about what you learned from reading the API documentation. How is the structure different from FRED?*] The World Bank API uses a RESTful structure where you specify the country code, indicator code, and date range in the URL. Data is returned in JSON format, where the first element contains metadata and the second element contains the observations. Compared to FRED, the World Bank API typically requires you to navigate nested lists, and you must extract the second element explicitly to get usable data.

4.2.2 Custom API Function

```
# Function to get World Bank data for a country and indicator
get_worldbank_data <- function(country, indicator, start = 2010, end = 2022) {
  url <- paste0("https://api.worldbank.org/v2/country/", country,
                "/indicator/", indicator,
                "?date=", start, ":", end,
                "&format=json")

  response <- request(url) %>%
    req_perform()

  data <- response %>%
    resp_body_json(simplifyVector = TRUE)

  # Extract observations (second list element)
  df <- data[[2]] %>%
    as_tibble() %>%
    select(date, value, country = countryiso3code)

  return(df)
}

# Test with one country
test_result <- get_worldbank_data("US", "NY.GDP.PCAP.CD")
head(test_result)
```

```
# A tibble: 6 x 3
  date   value country
  <chr>  <dbl> <chr>
1 2022  77861. USA
2 2021  71307. USA
3 2020  64402. USA
4 2019  65228. USA
5 2018  62876. USA
6 2017  60048. USA
```

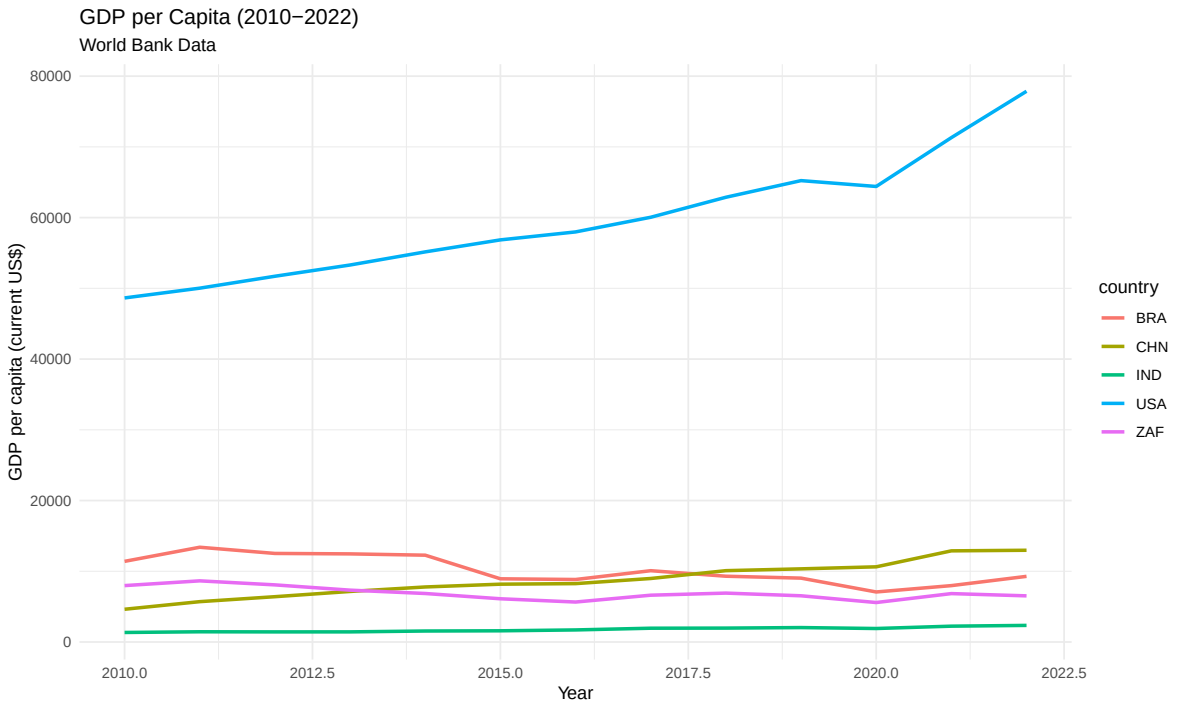
4.2.3 Data Collection and Analysis

```r
# Select countries and indicator (GDP per capita, current US$)
countries <- c("US", "CHN", "IND", "BRA", "ZAF")
indicator <- "NY.GDP.PCAP.CD"

# Collect data for multiple countries
wb_data <- map_dfr(countries, ~ get_worldbank_data(.x, indicator))

# Quick summary: latest available GDP per capita by country
summary_stats <- wb_data %>%
  group_by(country) %>%
  summarise(latest_gdp = value[which.max(date)],
            .groups = "drop")

summary_stats
```

```
# A tibble: 5 x 2
  country latest_gdp
  <chr>        <dbl>
1 BRA          9281.
2 CHN         12971.
3 IND          2347.
4 USA         77861.
5 ZAF          6523.
```

4.2.4 Visualization

```r
api_plot <- wb_data %>%
  mutate(date = as.numeric(date)) %>%
  ggplot(aes(x = date, y = value, color = country)) +
  geom_line(size = 1) +
  labs(title = "GDP per Capita (2010-2022)",
       subtitle = "World Bank Data",
       x = "Year", y = "GDP per capita (current US$)") +
  theme_minimal()

print(api_plot)
```



4.2.5 API Comparison and Reflection

[*Write 2-3 sentences about what was different/challenging compared to the FRED API. What did you learn about working with diverse APIs?*] Compared to the FRED API, the World Bank API had a more straightforward structure but required different query parameters (country code and indicator). One challenge was handling different naming conventions in the metadata, which required adapting my function logic. From this exercise, I learned that working with diverse APIs requires carefully reading documentation and adjusting data processing steps to match each API's unique response format.

4.3 API Landscape Survey

4.3.1 Quick API Exploration

```
# Explore three different API structures without writing full functions

# API 1: NASA Astronomy Picture of the Day (nested JSON, government API)
nasa_response <- request("https://api.nasa.gov/planetary/apod?api_key=DEMO_KEY") %>%
  req_perform()
```

```r
nasa_data <- nasa_response %>% resp_body_json()
str(nasa_data)
```

```
List of 8
 $ copyright      : chr "\nImran Sultan\n"
 $ date           : chr "2025-09-22"
 $ explanation    : chr "On Saturn, the rings tell you the season.  On Earth, today marks an
 $ hdurl          : chr "https://apod.nasa.gov/apod/image/2509/Saturn6Years_Sultan_960.jpg"
 $ media_type     : chr "image"
 $ service_version: chr "v1"
 $ title          : chr "Equinox at Saturn"
 $ url            : chr "https://apod.nasa.gov/apod/image/2509/Saturn6Years_Sultan_960.jpg"
```

```r
# API 2: JSONPlaceholder (flat JSON, RESTful design)
posts_response <- request("https://jsonplaceholder.typicode.com/posts/1") %>%
  req_perform()
posts_data <- posts_response %>% resp_body_json()
str(posts_data)
```

```
List of 4
 $ userId: int 1
 $ id    : int 1
 $ title : chr "sunt aut facere repellat provident occaecati excepturi optio reprehenderit"
 $ body  : chr "quia et suscipit\nsuscipit recusandae consequuntur expedita et cum\nreprehend
```

```r
# API 3: Cat Facts (simple response, no authentication)
facts_response <- request("https://catfact.ninja/fact") %>%
  req_perform()
facts_data <- facts_response %>% resp_body_json()
str(facts_data)
```

```
List of 2
 $ fact  : chr "Cats are the world's most popular pets, outnumbering dogs by as many as three
 $ length: int 84
```

4.3.2 API Structure Analysis

Response Structure Differences: [*How do these three APIs differ in their response structures compared to FRED?*] The NASA API returns a nested JSON with multiple descriptive

fields (e.g., copyright, date, explanation, media type, and image URLs), while JSONPlaceholder provides a flat JSON structure with only a few keys such as userId, id, title, and body. In contrast, Cat Facts gives a minimal structure with just fact and length. Compared to the FRED API, which delivers structured time series data with observations and metadata, these APIs highlight a broader diversity: descriptive, mock data, and simple entertainment responses.

Authentication Patterns: [*Which APIs require authentication and which don't? What does this tell you about API design?*]NASA requires an API key, even for demo access, reflecting government data management practices. JSONPlaceholder and Cat Facts do not require authentication, making them accessible but less controlled. This shows that APIs designed for official or sensitive data almost always require authentication, while test or entertainment APIs prioritize ease of use.

Government vs. Commercial vs. Fun APIs: [*What differences do you notice between NASA (government), your chosen API (likely commercial/research), and Cat Facts (entertainment)?*]NASA, as a government API, emphasizes completeness and metadata, ensuring traceability and scientific use. JSONPlaceholder resembles a commercial mock API designed to mimic RESTful structures for developers to test. Cat Facts is an entertainment API, simple and lightweight, with no authentication. This contrast illustrates how purpose—scientific reliability, developer testing, or casual fun—shapes API design.

Overall API Patterns: [*After working with 5+ different APIs total, what patterns emerge in API design? What makes an API easy or difficult to work with?*]Looking across all the APIs (including FRED, World Bank, Weather, etc.), JSON remains the dominant response format, but depth and complexity vary greatly. Authentication is a key differentiator: official and financial APIs nearly always require keys, while entertainment/test APIs don't. A well-designed API is easier to use if it has clear documentation, consistent fields, and predictable responses; challenges arise when responses are deeply nested or poorly documented.