PROSPR Google Sheets Automation Project: Implementation and Deployment Strategy

Gustavo Marin

July 16, 2025

Abstract

This report details the implementation of key automation features for PROSPR's financial planning Google Sheet template, focusing on an Admin menu with access control and a Monthly Comparative Report tool. It provides a comprehensive explanation of the code structure, design choices, and a robust strategy for bulk deployment across multiple client spreadsheets using Google Apps Script Libraries and the Google Apps Script API. Key assumptions made during development are also outlined.

Contents

1 Introduction

This document serves as a technical report for the PROSPR Google Sheets automation project. The primary objective was to enhance a financial planning template with two core features: a secure "Admin" menu and an automated "Monthly Comparative Report" generator. Additionally, a crucial aspect of this project is to propose a scalable solution for deploying these functionalities to numerous client spreadsheets.

The implementation leverages Google Apps Script, Google's JavaScript-based platform for extending Google Workspace applications. Emphasis has been placed on clean, modular, and maintainable code, as well as robust error handling and user experience.

2 Task 1: Admin Menu with Access Control

The first task involved creating a custom "Admin" menu in the Google Sheet's top menu bar, protected by an access control mechanism.

2.1 Implementation Details

The Admin menu's functionality is managed through several interconnected functions:

- `onOpen()`: This special Google Apps Script trigger automatically runs when the spreadsheet is opened. Its primary role is to initialize the custom "Admin" menu by calling `addAdminMenuWithAccessControl()`.
- `addAdminMenuWithAccessControl()`: This function creates the initial, locked version of the "Admin" menu. It contains only one item: "Unlock Admin Menu," which triggers the authentication process.
- `showAdminPrompt()`: When "Unlock Admin Menu" is selected, this function displays a UI prompt asking the user for the admin code. The entered code is then passed to `verifyAdminCode()` for validation.
- `verifyAdminCode()`: This is the core of the access control. It uses `PropertiesService.getUserProperties()` to store and retrieve the admin code.
  - **User-Specific Codes**: By utilizing `getUserProperties()`, each individual user (Google account) interacting with the script will have their own unique admin code. This is a critical design choice for multi-client deployments, ensuring security and client autonomy.
  - **Initial Code Setup**: If a user attempts to unlock the menu for the first time and no admin code is found in their user properties, the script automatically sets a default initial code (`'PROSPR2025'`) and prompts the user to re-enter it. This streamlines the onboarding process.
- `addUnlockedAdminMenu()`: Upon successful verification of the admin code, this function replaces the locked menu with the full "Admin" menu. This unlocked menu includes options for "Monthly Comparative Report," "Set Admin Code," and "Lock Admin Menu."

- `resetAdminMenu()`: This function reverts the "Admin" menu back to its locked state, providing a way for users to secure the menu after use.
- `setNewAdminCode()`: This function allows an authenticated user to change their personal admin code. For security, it first prompts for the current admin code before allowing the user to set a new one.

2.2 Design Choices and Rationale

The decision to use `PropertiesService.getUserProperties()` for admin code storage is fundamental. It ensures:

- **Security**: Each client's admin access is isolated, preventing a compromised code from affecting other clients.
- **Scalability**: No centralized management of client-specific codes is required in the master script. Clients manage their own access.
- **User Autonomy**: Clients have full control over their admin password.

The dynamic menu updating enhances the user experience by clearly indicating whether admin options are available.

3 Task 2: Monthly Comparative Report Tool

The second task involved building a tool to generate a compact report comparing actual vs. planned financial data from the "Monthly Budget" tab.

3.1 Implementation Details

The report generation is handled primarily by the `runMonthlyComparativeReport()` function, supported by `prettyDate()`, `generateTabularReportSheet()`, and `generateReportAsEmailDraft()`.

- `runMonthlyComparativeReport()`:
  - **Data Extraction**: Reads the "Monthly Budget" tab, extracting category headers, item descriptions, budgeted amounts, and actual amounts. It dynamically determines the report period (month, year, BOM, EOM) from designated cells.
  - **Data Parsing Logic**: The script iterates through the budget data row by row. It intelligently identifies main category headers and their associated line items. Crucially, it distinguishes between line items and "Total" rows within categories to prevent duplication in the report.
  - **Deviation Calculation**: For each category and significant line item, it calculates the absolute and percentage deviation between actual and planned values. Special handling is implemented for scenarios where the planned (budget) value is zero to avoid division-by-zero errors and provide meaningful percentage deviations (e.g., 100% over budget if actual is positive and planned is zero).

- **Thresholding**: Only categories and individual items exhibiting a deviation greater than a configurable `DEVIATION_THRESHOLD` (defaulting to 20%) are included in the detailed report, focusing on significant variances.
- **Output Preparation**: Data is structured into two main formats: a tabular array for the Google Sheet and an array of strings for the Gmail draft.

- `generateTabularReportSheet()`: This function takes the prepared tabular data and creates/updates a new sheet named '[Month] Budget Comparison' (e.g., "May Budget Comparison").
  - **Dynamic Sheet Management**: It checks if the report sheet already exists; if so, it clears it; otherwise, it creates a new one.
  - **Header Information**: Populates the top of the sheet with report period details (Year, Month, BOM, EOM).
  - **Data Insertion & Formatting**: Inserts the report data and applies extensive formatting:
    - Bold, grey background for table headers.
    - Light blue background, bold, and underlined text for category summary rows.
    - Light grey background and smaller font for indented line item rows.
    - Conditional background coloring for "Over" (light red) and "Under" (light green) budget statuses in the status column.
    - Right-alignment for all numeric columns.
    - Auto-resizing of columns for readability.

- `generateReportAsEmailDraft()`: This function creates a draft email in the user's Gmail account with the summarized report content. The content is wrapped in `<pre>` tags to ensure monospace formatting for readability. Robust error handling is included to catch potential Gmail permission issues.

- `prettyDate()`: A utility function to format Date objects into a consistent "MM/dd/yyyy" string format.

3.2 Design Choices and Rationale

- **Dual Output Formats**: Providing both a detailed sheet and a concise email draft caters to different user needs (in-depth analysis vs. quick summary for sharing).

- **Clear Visual Cues**: Extensive formatting in the report sheet (colors, bolding, indentation) makes it easy to quickly identify important information and deviations.

- **Robust Deviation Calculation**: The explicit handling of zero-budget scenarios ensures that percentage deviations are always meaningful, preventing misleading results or errors.

- **Preventing Duplicates**: The refined data parsing logic (specifically, `categoryHeader.indexOf('Total') === -1` when adding items) successfully eliminates the duplication of "Total" rows as individual line items in the report.

4 Task 3: Bulk Deployment Strategy

The bonus task of bulk deploying this functionality to multiple client spreadsheets requires a scalable and maintainable approach. The recommended strategy centers around Google Apps Script Libraries.

4.1 Concept of Google Apps Script Libraries

A Google Apps Script Library is a standalone Apps Script project that can be shared and reused across multiple other Apps Script projects. This approach offers significant advantages for deployment:

- **Centralized Codebase**: All core functionalities (Admin menu, report generation) reside in a single master library project.
- **Simplified Updates**: When changes or bug fixes are made to the core features, only the master library needs to be updated and re-deployed. All linked client spreadsheets automatically receive the updates (upon next open/execution, depending on library versioning).
- **Reduced Client-Side Code**: Each client spreadsheet only needs a minimal script to link to and call functions from the library, reducing the complexity of individual client projects.

4.2 Deployment Workflow

The ideal bulk deployment workflow would involve the following steps:

1. **Create and Deploy the Master Library**:
   - The `Code.gs` (provided in Annex A) would be developed as a separate Google Apps Script project.
   - This project would then be deployed as a "Library" from the Apps Script editor (`Deploy > New deployment > Select type: Library`). This deployment provides a unique Project ID for the library.
2. **Client-Side Integration**:
   - Each client's spreadsheet would have a very small `Code.gs` file (similar to `ClientSideScript.gs` described in the README).
   - This client-side script's `onOpen()` function would simply call the main initialization function from the deployed library (e.g., `ProsprAdminLib.addAdminMenuWithAccessControl()`).
3. **Programmatic Linking and Injection (Automation)**:
   - This is the core of the "bulk deployment" automation. To programmatically link the master library to each client's spreadsheet and inject the minimal `onOpen()` function into their script project, interaction with the **Google Apps Script API** is required.

- **Google Apps Script API**: This API allows for managing Apps Script projects programmatically. It can be used to:
  - Update the `appsscript.json` manifest file of a client's script project to add the library as a dependency.
  - Modify the content of script files (e.g., `Code.gs`) within a client's project to insert the `onOpen()` call.
- `clasp` (**C**ommand **L**ine **A**pps **S**cript **P**roject): For developers, `clasp` is a command-line tool that simplifies local development and deployment of Apps Script projects. While it doesn't directly automate linking libraries to *other* projects, it's part of a robust deployment pipeline.
- **Role of "Master Script Library"**: The assignment mentions access to a "Master Script Library." It is highly probable that this library contains pre-built helper functions that abstract the complexities of direct Apps Script API calls. For instance, a function like `MasterScriptLibrary.linkLibraryToClient(spreadsheetId, libraryId, identifier)` could handle the API interactions to link the library, and `MasterScriptLibrary.injectOnOpen(spreadsheetId)` could inject the necessary `onOpen()` code. This would be the most practical and efficient way to achieve bulk deployment in a real PROSPR environment.

4.3 Deployment Script Placeholder

A conceptual `DeploymentScript.gs` (similar to the one in the GitHub repository's README) would orchestrate this process. It would iterate through a list of client spreadsheet URLs, extract their IDs, and then call the appropriate helper functions (likely from the "Master Script Library") to perform the linking and code injection. Robust error handling and logging would be paramount in such a script.

4.4 Security and Permissions Considerations

- **Library Permissions**: The master library will require permissions for all services it uses (e.g., `SpreadsheetApp`, `GmailApp`, `PropertiesService`). Users of client spreadsheets will be prompted to authorize these permissions upon first use.
- **Deployment Script Permissions**: The bulk deployment script itself will need elevated permissions, particularly `DriveApp` (to access client spreadsheets) and potentially scopes related to the Apps Script API (to modify other script projects).
- **Admin Code Security**: The use of `getUserProperties()` ensures that admin codes are isolated per user/client, enhancing overall security.

5 Assumptions Made

During the development of the Monthly Comparative Report and the conceptualization of the deployment strategy, the following key assumptions were made, aligning with the project's README for consistency:

- **Data Mapping**: The "Budget" column in the "Monthly Budget" tab was explicitly mapped to "Planned" values in the comparative report.
- **Blank/Empty Handling**: Blank, empty, or non-numeric cells are interpreted as zero to prevent calculation errors and accurately flag missing/uncategorized data. This was a specific area requiring careful handling during development to ensure robust reporting.
- **Consistent Structure**: It is assumed that the "Monthly Budget" tab maintains a consistent structure regarding category headers, item descriptions, and the placement of Budget and Actual values, as observed in the provided screenshots and data.
- **Per-Account Security**: Admin codes are managed per Google account using `PropertiesService.getUserProperties()` for decentralized security. This design choice ensures that each client's admin credentials are isolated.
- **Gmail Permissions**: The script will prompt for Gmail permissions as needed for draft generation. Users will be prompted for this authorization upon first use.
- **Master Script Library Availability**: For Task 3, it is assumed that the "Master Script Library" mentioned in the assignment provides helper functions to facilitate programmatic interaction with client script projects (e.g., for linking libraries or injecting code). Without such helpers, direct Apps Script API interaction would be significantly more complex and typically managed via 'clasp' or similar tools for a production environment.

Annex A: Code.gs

```gs
/**
 * @file Code.gs
 * @description This script provides the core functionality for the PROSPR financial planning template,
 * including a custom Admin menu with access control and a Monthly Comparative Report tool.
 * This file is designed to be a self-contained Google Apps Script project for submission.
 *
 * It adheres to principles of clean code and modularity, with extensive comments.
 */

/**
 * The onOpen function is a special Google Apps Script trigger that runs automatically
 * when a user opens the spreadsheet. It's used here to initialize the custom 'Admin' menu.
 */
function onOpen() {
  // Add the Admin menu with password protection when the spreadsheet is opened.
  addAdminMenuWithAccessControl();
  // Note: Original script included a ProsprScript.onOpen() call.
  // For this self-contained submission, it's assumed that any base ProsprScript
  // functionality is either integrated or not required for this specific task.
}

/**
 * Adds a custom 'Admin' menu to the Google Sheet UI.
 * Initially, this menu only contains an 'Unlock Admin Menu' option.
 */
function addAdminMenuWithAccessControl() {
  var ui = SpreadsheetApp.getUi();
  ui.createMenu('Admin')
    .addItem('Unlock Admin Menu', 'showAdminPrompt')
    .addToUi();
}
```

```
32
33 /**
34  * Displays a prompt for the admin code. If the code is correct, it
    ↪ unlocks the full Admin menu.
35  */
36 function showAdminPrompt() {
37   var ui = SpreadsheetApp.getUi();
38   var response = ui.prompt('Admin Access', 'Please enter the admin
    ↪ code to unlock admin options:', ui.ButtonSet.OK_CANCEL);
39   if (response.getSelectedButton() === ui.Button.OK) {
40     var code = response.getResponseText();
41     if (verifyAdminCode(code)) {
42       addUnlockedAdminMenu();
43       ui.alert('Admin options unlocked!');
44     } else {
45       ui.alert('Incorrect code. Access denied.');
46     }
47   }
48 }
49
50 /**
51  * Verifies the admin code against the user's stored property.
52  * If no code is set, it will prompt to set the initial code
    ↪ 'PROSPR2025'.
53  * This uses UserProperties, ensuring each user (Google account)
    ↪ interacting with the script
54  * has their own distinct admin code, which is ideal for
    ↪ multi-client scenarios.
55  * @param {string} code The code entered by the user.
56  * @returns {boolean} True if the code is correct, false otherwise.
57  */
58 function verifyAdminCode(code) {
59   var userProperties = PropertiesService.getUserProperties();
60   var storedCode = userProperties.getProperty('ADMIN_CODE');
61
62   if (!storedCode) {
63     // If no admin code is set for the current user, set the initial
      ↪ one.
64     // This happens on the first attempt to unlock the menu by a new
      ↪ user.
65     userProperties.setProperty('ADMIN_CODE', 'PROSPR2025');
66     storedCode = 'PROSPR2025';
```

```
67      SpreadsheetApp.getUi().alert('Initial admin code "PROSPR2025"
        ↪ has been set for your user account. Please try unlocking the
        ↪ menu again.');
68      return false; // Initial setup, user needs to re-enter the code.
69    }
70
71    return code === storedCode;
72  }
73
74  /**
75   * Adds the full 'Admin' menu with all options (report, set code,
      ↪ lock menu).
76   * This is called after successful admin code verification.
77   */
78  function addUnlockedAdminMenu() {
79    var ui = SpreadsheetApp.getUi();
80    ui.createMenu('Admin')
81      .addItem('Monthly Comparative Report',
        ↪ 'runMonthlyComparativeReport')
82      .addItem('Set Admin Code', 'setNewAdminCode') // New menu item
        ↪ to change admin code
83      .addItem('Lock Admin Menu', 'resetAdminMenu')
84      .addToUi();
85  }
86
87  /**
88   * Resets the Admin menu back to its locked state (only 'Unlock
      ↪ Admin Menu' visible).
89   */
90  function resetAdminMenu() {
91    var ui = SpreadsheetApp.getUi();
92    ui.createMenu('Admin')
93      .addItem('Unlock Admin Menu', 'showAdminPrompt')
94      .addToUi();
95    ui.alert('Admin menu locked again.');
96  }
97
98  /**
99   * Prompts the user to set a new admin code. Requires current code
      ↪ verification for security.
100  */
101 function setNewAdminCode() {
```

```
  var ui = SpreadsheetApp.getUi();
  var userProperties = PropertiesService.getUserProperties();

  var currentCodeResponse = ui.prompt('Verify Current Admin Code',
   ↪  'Please enter your current admin code to change it:',
   ↪  ui.ButtonSet.OK_CANCEL);
  if (currentCodeResponse.getSelectedButton() === ui.Button.OK) {
    var currentCode = currentCodeResponse.getResponseText();
    if (verifyAdminCode(currentCode)) { // Use verifyAdminCode to
     ↪  check current code
      var newCodeResponse = ui.prompt('Set New Admin Code', 'Enter
       ↪  the new admin code:', ui.ButtonSet.OK_CANCEL);
      if (newCodeResponse.getSelectedButton() === ui.Button.OK) {
        var newCode = newCodeResponse.getResponseText();
        if (newCode) {
          userProperties.setProperty('ADMIN_CODE', newCode);
          ui.alert('Admin code successfully updated!');
        } else {
          ui.alert('New admin code cannot be empty.');
        }
      }
    } else {
      ui.alert('Incorrect current admin code. Cannot change.');
    }
  }
}

/**
 * Generates the Monthly Comparative Report based on the 'Monthly
  ↪  Budget' tab.
 * It processes budget data, identifies deviations, creates a new
  ↪  report sheet,
 * and sends a draft email summary.
 */
function runMonthlyComparativeReport() {
  var ss = SpreadsheetApp.getActiveSpreadsheet();
  var budgetSheetName = 'Monthly Budget';
  var sheet = ss.getSheetByName(budgetSheetName);

  // Retrieve report period details from the sheet's designated
   ↪  cells.
  var year = sheet.getRange('F2').getValue();
```

```
137   var month = sheet.getRange('F3').getValue();
138   var bom = sheet.getRange('H2').getValue(); // Beginning of Month
      ↪  date
139   var eom = sheet.getRange('H3').getValue(); // End of Month date
140
141   if (!sheet) {
142     SpreadsheetApp.getUi().alert('Error: Sheet "' + budgetSheetName
      ↪  + '" not found! Please ensure the "Monthly Budget" tab
      ↪  exists.');
143     return;
144   }
145
146   // Define column indices for data extraction (0-indexed for
      ↪  arrays).
147   var CATEGORY_COL = 1;     // Column B for category headers (e.g.,
      ↪  "Shelter")
148   var ITEM_DESC_COL = 2;    // Column C for item descriptions (e.g.,
      ↪  "Mortgage")
149   var BUDGET_COL = 3;       // Column D for Budgeted amounts
150   var ACTUAL_COL = 5;       // Column F for Actual amounts
151
152   // --- Configuration ---
153   var DEVIATION_THRESHOLD = 0.20; // 20% threshold for reporting
      ↪  significant deviation (e.g., 0.20 for 20%)
154   var START_ROW = 5;              // Data parsing starts from row 5
      ↪  in the 'Monthly Budget' sheet.
155
156   var data = sheet.getDataRange().getValues(); // Get all data from
      ↪  the active sheet.
157
158   // Objects to store parsed data:
159   // `allCategoriesData` will hold aggregated data for each main
      ↪  category.
160   var allCategoriesData = {};
161   var currentCategoryName = null;
162   var currentCategoryItems = [];
163   var currentCategoryTotalBudget = 0;
164   var currentCategoryTotalActual = 0;
165
166   // Loop through each row of the budget data to parse categories,
      ↪  items, and their totals.
167   for (var i = START_ROW - 1; i < data.length; i++) {
```

```
168     var row = data[i];
169     // Safely convert cell values to string and trim whitespace.
170     var categoryHeader = (row[CATEGORY_COL] !== null &&
        ↪  row[CATEGORY_COL] !== undefined) ?
        ↪  String(row[CATEGORY_COL]).trim() : "";
171     var itemDescription = (row[ITEM_DESC_COL] !== null &&
        ↪  row[ITEM_DESC_COL] !== undefined) ?
        ↪  String(row[ITEM_DESC_COL]).trim() : "";
172
173     // Parse budget and actual values, treating empty or non-numeric
        ↪  values as 0.
174     var budgetValue = parseFloat(row[BUDGET_COL]) || 0;
175     var actualValue = parseFloat(row[ACTUAL_COL]) || 0;
176
177     // Detect a new main category header (e.g., "Shelter", "Food &
        ↪  Supplies").
178     // Ensure it's not empty and not a "Total" row (which marks the
        ↪  end of a category block).
179     if (categoryHeader && categoryHeader.indexOf('Total') === -1) {
180       // If we were processing a previous category, save its
          ↪  aggregated data before starting a new one.
181       if (currentCategoryName) {
182         allCategoriesData[currentCategoryName] = {
183           items: currentCategoryItems,
184           totalBudget: currentCategoryTotalBudget,
185           totalActual: currentCategoryTotalActual
186         };
187       }
188       // Initialize variables for the new category.
189       currentCategoryName = categoryHeader;
190       currentCategoryItems = [];
191       currentCategoryTotalBudget = 0;
192       currentCategoryTotalActual = 0;
193     }
194
195     // Add item details to the current category.
196     // This condition ensures that:
197     // 1. A category is currently being processed
        ↪  (`currentCategoryName`).
198     // 2. The row has an item description OR non-zero budget/actual
        ↪  values (to capture items with only one value).
199     // 3. The row is NOT a "Total" row (this prevents the "Total"
        ↪  line from being duplicated as an item).
```

```
    if (currentCategoryName && (itemDescription || budgetValue !== 0 || actualValue !== 0) && categoryHeader.indexOf('Total') === -1) {
        currentCategoryItems.push({
          description: itemDescription,
          budget: budgetValue,
          actual: actualValue
        });
    }

    // Detect a "Total" row, which signifies the end of a category's data block.
    if (categoryHeader.indexOf('Total') === 0 && currentCategoryName) {
      // Assign the total budget and actual values for the current category.
      currentCategoryTotalBudget = budgetValue;
      currentCategoryTotalActual = actualValue;

      // Save the complete category data (items and totals) to `allCategoriesData`.
      allCategoriesData[currentCategoryName] = {
        items: currentCategoryItems,
        totalBudget: currentCategoryTotalBudget,
        totalActual: currentCategoryTotalActual
      };

      // Reset variables to prepare for the next category.
      currentCategoryName = null;
      currentCategoryItems = [];
      currentCategoryTotalBudget = 0;
      currentCategoryTotalActual = 0;
    }
  }

  // Handle the last category in the sheet if the loop ends without encountering its "Total" row.
  if (currentCategoryName) {
    allCategoriesData[currentCategoryName] = {
      items: currentCategoryItems,
      totalBudget: currentCategoryTotalBudget,
      totalActual: currentCategoryTotalActual
```

```
    };
  }

  // Prepare data structures for the tabular report sheet and the email draft.
  var reportRowsForSheet = [
    [
      "Category",          // Column A header
      "Item Description", // Column B header
      "Actual",            // Column C header
      "Planned",           // Column D header (mapped from Budget)
      "Deviation ($)",     // Column E header
      "Deviation (%)",     // Column F header
      "Status"             // Column G header
    ]
  ];
  var reportLinesForEmail = []; // Array to build the email body content.

  // Iterate through all parsed categories to build the detailed report.
  for (var catName in allCategoriesData) {
    var d = allCategoriesData[catName];
    var actual = d.totalActual || 0;
    var budget = d.totalBudget || 0;

    // Calculate deviation in absolute terms.
    var deviation = actual - budget;

    // Calculate percentage deviation, handling division by zero for zero budgets.
    var deviationPct = 0;
    if (budget === 0) {
      deviationPct = (actual === 0) ? 0 : (actual > 0 ? 1 : -1); // If budget is 0, actual > 0 means 100% over, actual < 0 means 100% under.
    } else {
      deviationPct = deviation / budget;
    }

    var deviationPctStr = (deviationPct * 100).toFixed(1) + "%";
    // Determine status based on deviation threshold.
```

```
    var status = Math.abs(deviationPct) > DEVIATION_THRESHOLD
      ? (deviationPct > 0 ? "Over" : "Under")
      : "OK";
    var deviationSign = deviation > 0 ? "+" : ""; // Add '+' sign
     ↪  for positive deviations.
    var deviationStr = deviationSign + deviation.toFixed(2);

    // Include categories in the report only if they have
     ↪  significant deviation
    // or if there's a value in one column but zero in the other
     ↪  (indicating a notable difference).
    if (status !== "OK" || (budget === 0 && actual !== 0) || (budget
     ↪  !== 0 && actual === 0)) {
      // Add the category summary row for the Google Sheet report.
      reportRowsForSheet.push([
        catName, // Category name in Column A
        "",      // Empty for Column B (Item Description)
        "$" + actual.toFixed(2), // Formatted Actual value
        "$" + budget.toFixed(2), // Formatted Planned value
        deviationStr,
        deviationPctStr,
        status
      ]);

      // Add the category summary line for the email report.
      reportLinesForEmail.push(
        catName + ": " + status + " budget by " + deviationPctStr +
        " ($" + actual.toFixed(2) + " vs. $" + budget.toFixed(2) +
         ↪  ")"
      );

      // Collect and report on significant individual items within
       ↪  this category.
      var significantItems = [];
      for (var j = 0; j < d.items.length; j++) {
        var item = d.items[j];
        var itemDeviation = item.actual - item.budget;
        var itemDeviationPct = 0;

        if (item.budget === 0) {
          itemDeviationPct = (item.actual === 0) ? 0 : (item.actual
           ↪  > 0 ? 1 : -1);
```

```
306         } else {
307           itemDeviationPct = itemDeviation / item.budget;
308         }
309 
310         // Highlight items if their percentage deviation exceeds the
            ↪ threshold,
311         // or if one value is zero and the other is not.
312         if (Math.abs(itemDeviationPct) > DEVIATION_THRESHOLD ||
            ↪ (item.budget === 0 && item.actual !== 0) || (item.budget
            ↪ !== 0 && item.actual === 0)) {
313           var itemDiffSign = itemDeviation > 0 ? "+" : "";
314           significantItems.push({
315             description: item.description,
316             actual: item.actual,
317             budget: item.budget,
318             diff: itemDiffSign + itemDeviation.toFixed(2),
319             diffPct: (itemDeviationPct * 100).toFixed(1) + "%"
320           });
321 
322           // Add detailed item row for the sheet (indented in Column
              ↪ B).
323           reportRowsForSheet.push([
324             "", // Empty for Column A (Category)
325             item.description, // Item description in Column B
326             "$" + item.actual.toFixed(2),
327             "$" + item.budget.toFixed(2),
328             itemDiffSign + itemDeviation.toFixed(2),
329             (itemDeviationPct * 100).toFixed(1) + "%", // Show %
                ↪ deviation for individual items
330             ""  // No status for individual items
331           ]);
332         }
333       }
334 
335       // Add key items to the email summary if any significant items
          ↪ were found.
336       if (significantItems.length > 0) {
337         reportLinesForEmail.push("   Key Items:");
338         significantItems.forEach(function(item) {
339           reportLinesForEmail.push(
340             "     " + item.description + ": $" +
                ↪ item.actual.toFixed(2) + " (Actual) vs $" +
                ↪ item.budget.toFixed(2) + " (Planned) (Diff: " +
                ↪ item.diff + ", " + item.diffPct + ")"
```

```
            );
          });
        }
        reportLinesForEmail.push(""); // Add a blank line between
        ↪  categories for email clarity.

        // Add an empty row after each category block for visual
        ↪  separation in the sheet.
        reportRowsForSheet.push(["", "", "", "", "", "", ""]);
      }
    }

    // Generate the tabular report in a new sheet.
    generateTabularReportSheet(reportRowsForSheet, month, year, bom,
    ↪  eom);

    // Prepare and generate the email draft summary.
    var bomStr = prettyDate(bom);
    var eomStr = prettyDate(eom);
    var finalReportText =
      'Monthly Budget Deviation Report\n' +
      'Period: ' + month + ' ' + year + ' (' + bomStr + ' - ' + eomStr
      ↪  + ')\n' +
      'Generated on: ' + new Date().toLocaleDateString() + '\n\n' +
      reportLinesForEmail.join("\n");

    var emailSubject = month + ' ' + year + ' Budget Comparison';
    generateReportAsEmailDraft(finalReportText, emailSubject);
}

/**
 * Formats a Date object into a "MM/dd/yyyy" string.
 * @param {Date} date The date object to format.
 * @returns {string} The formatted date string.
 */
function prettyDate(date) {
    // Ensure the input is a Date object before formatting.
    if (!(date instanceof Date)) return date;
    return Utilities.formatDate(date, Session.getScriptTimeZone(),
    ↪  "MM/dd/yyyy");
}

```

```
/**
 * Generates a tabular report in a new Google Sheet.
 * This function handles sheet creation/clearing, header population,
 * data insertion, and formatting.
 * @param {Array<Array<any>>} tableData The data for the main report
 ↪  table.
 * @param {string} month The month for the report.
 * @param {number} year The year for the report.
 * @param {Date} bom The beginning of the month date.
 * @param {Date} eom The end of the month date.
 */
function generateTabularReportSheet(tableData, month, year, bom,
↪  eom) {
  var ss = SpreadsheetApp.getActiveSpreadsheet();
  var reportSheetName = month + ' Budget Comparison';
  var reportSheet = ss.getSheetByName(reportSheetName);

  // Clear or create the report sheet.
  if (reportSheet) {
    reportSheet.clear(); // Clear existing content if sheet exists.
  } else {
    reportSheet = ss.insertSheet(reportSheetName); // Create new
    ↪  sheet if it doesn't exist.
  }

  // Populate the top-right header section with report period
  ↪  details.
  reportSheet.getRange('C2').setValue("Year");
  reportSheet.getRange('D2').setValue(year);
  reportSheet.getRange('E2').setValue("BOM");
  reportSheet.getRange('F2').setValue(Utilities.formatDate(bom,
  ↪  Session.getScriptTimeZone(), "M/d/yyyy"));

  reportSheet.getRange('C3').setValue("Month");
  reportSheet.getRange('D3').setValue(month);
  reportSheet.getRange('E3').setValue("EOM");
  reportSheet.getRange('F3').setValue(Utilities.formatDate(eom,
  ↪  Session.getScriptTimeZone(), "M/d/yyyy"));

  // Output the main report table data starting at row 5, column 1.
  var nRows = tableData.length;
  var nCols = 7; // Number of columns in the report table.
```

```
414   var dataStartRow = 5; // The row where the main table data begins.
415   reportSheet.getRange(dataStartRow, 1, nRows,
      ↪  nCols).setValues(tableData);
416
417   // Apply formatting to the header row of the report table.
418   var headerRange = reportSheet.getRange(dataStartRow, 1, 1, nCols);
419   headerRange.setFontWeight('bold');
420   headerRange.setBackground('#e3e3e3'); // Light grey background.
421   headerRange.setFontFamily('Arial, Helvetica, sans-serif');
422
423   // Apply a consistent font family to all data rows below the
      ↪  header.
424   if (nRows > 1) { // Ensure there are actual data rows.
425     reportSheet.getRange(dataStartRow + 1, 1, nRows - 1,
       ↪  nCols).setFontFamily('Arial, Helvetica, sans-serif');
426   }
427
428   // Apply conditional formatting and styling to data rows based on
      ↪  content.
429   for (var i = 0; i < nRows - 1; i++) { // Loop through data rows
      ↪  (skip the table header row).
430     var currentRowIndex = dataStartRow + 1 + i; // Actual row index
       ↪  in the sheet.
431     var rowData = tableData[i + 1]; // Get the corresponding data
       ↪  from the `tableData` array.
432
433     var categoryCellContent = String(rowData[0]); // Content of
       ↪  Column A for the current row.
434
435     // If the row is an empty separator row (added for visual
       ↪  spacing).
436     if (categoryCellContent === "") {
437         reportSheet.getRange(currentRowIndex, 1, 1,
          ↪  nCols).setBackground('#ffffff'); // White background.
438         continue; // Skip further formatting for empty rows.
439     }
440
441     // If Column A is empty but Column B has content, it's an
       ↪  indented item row.
442     if (rowData[0] === "" && rowData[1] !== "") {
443       reportSheet.getRange(currentRowIndex, 1, 1,
        ↪  nCols).setBackground('#f7f7f7'); // Light grey background.
```

```
444       reportSheet.getRange(currentRowIndex, 2).setFontSize(9); //
          ↪ Smaller font for item description.
445       reportSheet.getRange(currentRowIndex,
          ↪ 2).setHorizontalAlignment('left'); // Left align item
          ↪ description.
446     } else {
447       // Otherwise, it's a main category summary row.
448       var status = rowData[6]; // Get the 'Status' from Column G.
449       var statusCell = reportSheet.getRange(currentRowIndex, 7); //
          ↪ Reference to the Status cell.
450
451       // Apply background color based on status (Over/Under budget).
452       if (status === "Over") {
453         statusCell.setBackground('#ffd6d6'); // Light red for over
            ↪ budget.
454       } else if (status === "Under") {
455         statusCell.setBackground('#d6ffd6'); // Light green for
            ↪ under budget.
456       }
457       reportSheet.getRange(currentRowIndex,
          ↪ 1).setFontWeight('bold'); // Bold category name.
458       reportSheet.getRange(currentRowIndex,
          ↪ 1).setFontLine('underline'); // Underline category name.
459       reportSheet.getRange(currentRowIndex, 1, 1,
          ↪ nCols).setBackground('#f0f8ff'); // Light blue background
          ↪ for categories.
460     }
461   }
462
463   // Auto-resize all columns for optimal readability.
464   for (var c = 1; c <= nCols; c++) {
465     reportSheet.autoResizeColumn(c);
466   }
467
468   // Right-align numeric columns for better presentation.
469   reportSheet.getRange(dataStartRow, 3, nRows,
      ↪ 1).setHorizontalAlignment('right'); // Actual (Column C)
470   reportSheet.getRange(dataStartRow, 4, nRows,
      ↪ 1).setHorizontalAlignment('right'); // Planned (Column D)
471   reportSheet.getRange(dataStartRow, 5, nRows,
      ↪ 1).setHorizontalAlignment('right'); // Deviation ($) (Column
      ↪ E)
```

```
472   reportSheet.getRange(dataStartRow, 6, nRows,
      ↪ 1).setHorizontalAlignment('right'); // Deviation (%) (Column
      ↪ F)
473
474   // Set the newly generated report sheet as the active sheet for
      ↪ user visibility.
475   ss.setActiveSheet(reportSheet);
476   SpreadsheetApp.getUi().alert('Report generated successfully in the
      ↪ "' + reportSheetName + '" sheet.');
477 }
478
479 /**
480  * Creates a draft email in Gmail with the report content.
481  * This provides an alternative output format for the report
      ↪ summary.
482  * @param {string} content The plain text content of the report
      ↪ summary for the email body.
483  * @param {string} subject The subject line for the email draft.
484  */
485 function generateReportAsEmailDraft(content, subject) {
486   try {
487     // Create a new Gmail draft. The content is wrapped in <pre>
        ↪ tags for monospace formatting.
488     GmailApp.createDraft('', subject, '', { htmlBody: '<pre>' +
        ↪ content + '</pre>' });
489     SpreadsheetApp.getUi().alert('Report has been saved as a draft
        ↪ in your Gmail.');
490   } catch (e) {
491     // Catch block to handle potential Gmail permissions issues.
492     SpreadsheetApp.getUi().alert('Could not create Gmail draft.
        ↪ Please ensure you have granted script permissions for Gmail.
        ↪ Error: ' + e.message);
493   }
494 }
```