

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	2
2	Task 1: Admin Menu with Access Control	2
2.1	Implementation Details	2
2.2	Design Choices and Rationale	3
3	Task 2: Monthly Comparative Report Tool	3
3.1	Implementation Details	3
3.2	Design Choices and Rationale	4
4	Task 3: Bulk Deployment Strategy	5
4.1	Concept of Google Apps Script Libraries	5
4.2	Deployment Workflow	5
4.3	Deployment Script Placeholder	6
4.4	Security and Permissions Considerations	6
5	Assumptions Made	6

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 B) 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, containing the minimal `onOpen()` setup code. This code 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 (Command Line Apps Script Project):** 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` (provided in Annex A) 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: DeploymentScript.gs

```
1  /**
2   * @file DeploymentScript.gs
3   * @description Conceptual script for bulk deploying PROSPR Admin
4   *   ↳ features to client spreadsheets.
5   * This script demonstrates how a master deployment tool would
6   *   ↳ interact with
7   * a hypothetical 'MasterScriptLibrary' to link a shared library
8   *   ↳ and
9   * inject necessary client-side setup code.
10  */
11
12 /**
13  * Initiates the bulk deployment process for PROSPR Admin features.
14  * Iterates through a list of client spreadsheet URLs and attempts
15  *   ↳ to
16  * link the shared library and inject the onOpen() setup code.
17  */
18 function deployProsprFeaturesToClients() {
19   // List of client Google Sheet URLs. In a production environment,
20   // this list would typically be managed dynamically (e.g., from a
21   *   ↳ database,
22   // a configuration sheet, or an administrative UI).
23   var clientSheetUrls = [
24     'https://docs.google.com/spreadsheets/d/CLIENT_SHEET_ID_1/edit',
25     'https://docs.google.com/spreadsheets/d/CLIENT_SHEET_ID_2/edit',
26     // Add more client sheet URLs as needed
27   ];
28
29   // The Project ID of your deployed ProsprAdminLib library.
30   // This ID is obtained when you deploy your ProsprAdminLib project
31   *   ↳ as a library.
32   var prosprAdminLibId = 'YOUR_PROSPR_ADMIN_LIB_PROJECT_ID'; // <<<
33   *   ↳ IMPORTANT: Replace with your actual Library Project ID
34
35   // The identifier (alias) that the client script will use to refer
36   *   ↳ to your library.
```



```

32  var libraryIdentifier = 'ProsprAdminLib';
33
34  // The minimal onOpen() code to inject into each client's Code.gs.
35  // This function simply calls the library's main setup function.
36  var clientOnOpenCode = [
37      '/*',
38      ' * @file Code.gs',
39      ' * @description Client-side script for PROSPR financial
    ↪ planning template.',
40      ' * This script links to the shared ProsprAdminLib library.',
41      ' */',
42      'function onOpen() {',
43      '    // Ensure ProsprScript.onOpen() is called if it exists and is
    ↪ part of client\'s base setup.',
44      '    // This assumes ProsprScript is either a globally available
    ↪ function or another linked library.',
45      '    if (typeof ProsprScript !== \'undefined\' &&
    ↪ ProsprScript.onOpen) {',
46      '        ProsprScript.onOpen();',
47      '    }',
48      '    // Initialize the custom Admin menu from the shared
    ↪ library.',
49      '    ' + libraryIdentifier + '.addAdminMenuWithAccessControl();',
50      '}'
51  ].join('\n');
52
53  Logger.log('Starting bulk deployment...');
54
55  clientSheetUrls.forEach(function(url) {
56      var spreadsheetId = getIdFromUrl(url);
57      if (!spreadsheetId) {
58          Logger.log('Skipping invalid URL: ' + url);
59          return;
60      }
61
62      try {
63          Logger.log('Processing client sheet: ' + url + ' (ID: ' +
    ↪ spreadsheetId + ')');
64
65          // Step 1: Link the ProsprAdminLib library to the client's
    ↪ script project.
66          // This hypothetical function would use the Apps Script API to
    ↪ update the

```

```

67 // client project's appsscript.json manifest to include the
   ↪ library dependency.
68 // Example: MasterScriptLibrary.addLibrary(projectId,
   ↪ libraryId, identifier)
69 // For a real implementation, you'd need the client script's
   ↪ Project ID,
70 // which can be obtained via DriveApp or Sheets API if
   ↪ permissions allow.
71 // Assuming MasterScriptLibrary handles getting the script ID
   ↪ from spreadsheet ID.
72 Logger.log(' - Attempting to link library ' +
   ↪ libraryIdentifier + ' (ID: ' + prosprAdminLibId + ')');
73
74 // In a real Master Script Library, this might look like:
75 // var clientScriptProjectId =
   ↪ MasterScriptLibrary.getScriptProjectIdForSpreadsheet (spreadsheetId);
76 //
   ↪ MasterScriptLibrary.addLibraryDependency(clientScriptProjectId,
   ↪ prosprAdminLibId, libraryIdentifier);
77
78 // Placeholder for actual library linking logic:
79 // This line represents the action of linking the library.
80 // In a real system, this would be a call to a function in the
   ↪ Master Script Library
81 // that uses the Apps Script API to modify the client's script
   ↪ project.
82 Logger.log(' [Conceptual Action:
   ↪ MasterScriptLibrary.linkLibrary(spreadsheetId,
   ↪ prosprAdminLibId, libraryIdentifier)]');
83
84
85 // Step 2: Programmatically inject or update the client's
   ↪ onOpen() function.
86 // This hypothetical function would use the Apps Script API to
   ↪ update the
87 // content of the Code.gs file in the client's script project.
88 // Example: MasterScriptLibrary.updateScriptFile(projectId,
   ↪ 'Code.gs', clientOnOpenCode)
89 Logger.log(' - Attempting to inject/update onOpen() function
   ↪ in Code.gs');
90
91 // In a real Master Script Library, this might look like:

```

```

92      //
      ↪ MasterScriptLibrary.updateScriptFileContent(clientScriptProjectId,
      ↪ 'Code.gs', clientOnOpenCode);
93
94      // Placeholder for actual code injection logic:
95      // This line represents the action of injecting the onOpen
      ↪ code.
96      // Similar to library linking, this would be a call to a
      ↪ Master Script Library function.
97      Logger.log('      [Conceptual Action:
      ↪ MasterScriptLibrary.injectCode(spreadsheetId, "Code.gs",
      ↪ clientOnOpenCode)]');
98
99      Logger.log('Successfully processed client sheet: ' + url);
100
101      } catch (e) {
102          Logger.log('Failed to deploy to ' + url + ': ' + e.message);
103      }
104  });
105
106  Logger.log('Bulk deployment script finished. Check logs for
      ↪ detailed status.');
```

SpreadsheetApp.getUi().alert('Bulk deployment script finished.
 ↪ Check Google Apps Script logs for details.');

```

108 }
109
110 /**
111  * Extracts the Google Sheet ID from a given URL.
112  * @param {string} url The URL of the Google Sheet.
113  * @returns {string|null} The spreadsheet ID or null if not found.
114  */
115 function getIdFromUrl(url) {
116     var match = url.match(/[-\w]{25,}/);
117     return match ? match[0] : null;
118 }
119
120 // To run this deployment script, you would typically execute it
    ↪ from
121 // its own Apps Script project (e.g., via the 'Run' menu in the
    ↪ editor).
122 // Ensure this script project has the necessary OAuth scopes
    ↪ enabled
```

```
123 // (e.g., Google Drive API, Google Apps Script API) and is
    ↪ authorized.
124 // function runDeployment() {
125 //   deployProsprFeaturesToClients();
126 // }
```

Annex B: Code.gs

```
1  /**
2   * @file Code.gs
3   * @description This script provides the core functionality for the
4   *   ↳ PROSPR financial planning template,
5   * including a custom Admin menu with access control and a Monthly
6   *   ↳ Comparative Report tool.
7   * This file is designed to be a self-contained Google Apps Script
8   *   ↳ project for submission.
9   *
10  * It adheres to principles of clean code and modularity, with
11  *   ↳ extensive comments.
12  */
13
14 /**
15  * The onOpen function is a special Google Apps Script trigger that
16  *   ↳ runs automatically
17  * when a user opens the spreadsheet. It's used here to initialize
18  *   ↳ the custom 'Admin' menu.
19  */
20
21 function onOpen() {
22   // Add the Admin menu with password protection when the
23   *   ↳ spreadsheet is opened.
24   addAdminMenuWithAccessControl();
25   // Note: Original script included a ProsprScript.onOpen() call.
26   // For this self-contained submission, it's assumed that any base
27   *   ↳ ProsprScript
28   // functionality is either integrated or not required for this
29   *   ↳ specific task.
30 }
31
32 /**
33  * Adds a custom 'Admin' menu to the Google Sheet UI.
34  * Initially, this menu only contains an 'Unlock Admin Menu' option.
35  */
36
37 function addAdminMenuWithAccessControl() {
38   var ui = SpreadsheetApp.getUi();
39   ui.createMenu('Admin')
40     .addItem('Unlock Admin Menu', 'showAdminPrompt')
41     .addToUi();
42 }
```

```

32
33 /**
34  * Displays a prompt for the admin code. If the code is correct, it
35  * ↳ unlocks the full Admin menu.
36  */
37 function showAdminPrompt() {
38     var ui = SpreadsheetApp.getUi();
39     var response = ui.prompt('Admin Access', 'Please enter the admin
40     * ↳ code to unlock admin options:', ui.ButtonSet.OK_CANCEL);
41     if (response.getSelectedButton() === ui.Button.OK) {
42         var code = response.getResponseText();
43         if (verifyAdminCode(code)) {
44             addUnlockedAdminMenu();
45             ui.alert('Admin options unlocked!');
46         } else {
47             ui.alert('Incorrect code. Access denied.');
```

```

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() {
```

```

102  var ui = SpreadsheetApp.getUi();
103  var userProperties = PropertiesService.getUserProperties();
104
105  var currentCodeResponse = ui.prompt('Verify Current Admin Code',
    ↪ 'Please enter your current admin code to change it:',
    ↪ ui.ButtonSet.OK_CANCEL);
106  if (currentCodeResponse.getSelectedButton() === ui.Button.OK) {
107      var currentCode = currentCodeResponse.getResponseText();
108      if (verifyAdminCode(currentCode)) { // Use verifyAdminCode to
    ↪ check current code
109          var newCodeResponse = ui.prompt('Set New Admin Code', 'Enter
    ↪ the new admin code:', ui.ButtonSet.OK_CANCEL);
110          if (newCodeResponse.getSelectedButton() === ui.Button.OK) {
111              var newCode = newCodeResponse.getResponseText();
112              if (newCode) {
113                  userProperties.setProperty('ADMIN_CODE', newCode);
114                  ui.alert('Admin code successfully updated!');
115              } else {
116                  ui.alert('New admin code cannot be empty.');
```



```

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).

```

```

200     if (currentCategoryName && (itemDescription || budgetValue !== 0
    ↪ || actualValue !== 0) && categoryHeader.indexOf('Total') ===
    ↪ -1) {
201         currentCategoryItems.push({
202             description: itemDescription,
203             budget: budgetValue,
204             actual: actualValue
205         });
206     }
207
208     // Detect a "Total" row, which signifies the end of a category's
    ↪ data block.
209     if (categoryHeader.indexOf('Total') === 0 &&
    ↪ currentCategoryName) {
210         // Assign the total budget and actual values for the current
    ↪ category.
211         currentCategoryTotalBudget = budgetValue;
212         currentCategoryTotalActual = actualValue;
213
214         // Save the complete category data (items and totals) to
    ↪ `allCategoriesData`.
215         allCategoriesData[currentCategoryName] = {
216             items: currentCategoryItems,
217             totalBudget: currentCategoryTotalBudget,
218             totalActual: currentCategoryTotalActual
219         };
220
221         // Reset variables to prepare for the next category.
222         currentCategoryName = null;
223         currentCategoryItems = [];
224         currentCategoryTotalBudget = 0;
225         currentCategoryTotalActual = 0;
226     }
227 }
228
229 // Handle the last category in the sheet if the loop ends without
    ↪ encountering its "Total" row.
230 if (currentCategoryName) {
231     allCategoriesData[currentCategoryName] = {
232         items: currentCategoryItems,
233         totalBudget: currentCategoryTotalBudget,
234         totalActual: currentCategoryTotalActual

```

```

235     };
236 }
237
238 // Prepare data structures for the tabular report sheet and the
    ↪ email draft.
239 var reportRowsForSheet = [
240     [
241         "Category",           // Column A header
242         "Item Description",    // Column B header
243         "Actual",             // Column C header
244         "Planned",            // Column D header (mapped from Budget)
245         "Deviation ($)",       // Column E header
246         "Deviation (%)",       // Column F header
247         "Status"              // Column G header
248     ]
249 ];
250 var reportLinesForEmail = []; // Array to build the email body
    ↪ content.
251
252 // Iterate through all parsed categories to build the detailed
    ↪ report.
253 for (var catName in allCategoriesData) {
254     var d = allCategoriesData[catName];
255     var actual = d.totalActual || 0;
256     var budget = d.totalBudget || 0;
257
258     // Calculate deviation in absolute terms.
259     var deviation = actual - budget;
260
261     // Calculate percentage deviation, handling division by zero for
    ↪ zero budgets.
262     var deviationPct = 0;
263     if (budget === 0) {
264         deviationPct = (actual === 0) ? 0 : (actual > 0 ? 1 : -1); //
    ↪ If budget is 0, actual > 0 means 100% over, actual < 0
    ↪ means 100% under.
265     } else {
266         deviationPct = deviation / budget;
267     }
268
269     var deviationPctStr = (deviationPct * 100).toFixed(1) + "%";
270     // Determine status based on deviation threshold.

```

```

271     var status = Math.abs(deviationPct) > DEVIATION_THRESHOLD
272         ? (deviationPct > 0 ? "Over" : "Under")
273         : "OK";
274     var deviationSign = deviation > 0 ? "+" : ""; // Add '+' sign
275     ↪ for positive deviations.
276     var deviationStr = deviationSign + deviation.toFixed(2);
277
278     // Include categories in the report only if they have
279     ↪ significant deviation
280     // or if there's a value in one column but zero in the other
281     ↪ (indicating a notable difference).
282     if (status !== "OK" || (budget === 0 && actual !== 0) || (budget
283     ↪ !== 0 && actual === 0)) {
284         // Add the category summary row for the Google Sheet report.
285         reportRowsForSheet.push([
286             catName, // Category name in Column A
287             "", // Empty for Column B (Item Description)
288             "$" + actual.toFixed(2), // Formatted Actual value
289             "$" + budget.toFixed(2), // Formatted Planned value
290             deviationStr,
291             deviationPctStr,
292             status
293         ]);
294
295         // Add the category summary line for the email report.
296         reportLinesForEmail.push(
297             catName + ": " + status + " budget by " + deviationPctStr +
298             " ($" + actual.toFixed(2) + " vs. $" + budget.toFixed(2) +
299             ↪ " )"
300         );
301
302         // Collect and report on significant individual items within
303         ↪ this category.
304         var significantItems = [];
305         for (var j = 0; j < d.items.length; j++) {
306             var item = d.items[j];
307             var itemDeviation = item.actual - item.budget;
308             var itemDeviationPct = 0;
309
310             if (item.budget === 0) {
311                 itemDeviationPct = (item.actual === 0) ? 0 : (item.actual
312                 ↪ > 0 ? 1 : -1);

```

```

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

```

```

341         );
342     });
343 }
344 reportLinesForEmail.push(""); // Add a blank line between
    ↪ categories for email clarity.
345
346 // Add an empty row after each category block for visual
    ↪ separation in the sheet.
347 reportRowsForSheet.push(["", "", "", "", "", "", ""]);
348 }
349 }
350
351 // Generate the tabular report in a new sheet.
352 generateTabularReportSheet(reportRowsForSheet, month, year, bom,
    ↪ eom);
353
354 // Prepare and generate the email draft summary.
355 var bomStr = prettyDate(bom);
356 var eomStr = prettyDate(eom);
357 var finalReportText =
358     'Monthly Budget Deviation Report\n' +
359     'Period: ' + month + ' ' + year + ' (' + bomStr + ' - ' + eomStr
    ↪ + ')\n' +
360     'Generated on: ' + new Date().toLocaleDateString() + '\n\n' +
361     reportLinesForEmail.join("\n");
362
363 var emailSubject = month + ' ' + year + ' Budget Comparison';
364 generateReportAsEmailDraft(finalReportText, emailSubject);
365 }
366
367 /**
368  * Formats a Date object into a "MM/dd/yyyy" string.
369  * @param {Date} date The date object to format.
370  * @returns {string} The formatted date string.
371  */
372 function prettyDate(date) {
373     // Ensure the input is a Date object before formatting.
374     if (!(date instanceof Date)) return date;
375     return Utilities.formatDate(date, Session.getScriptTimeZone(),
    ↪ "MM/dd/yyyy");
376 }
377

```

```

378  /**
379   * Generates a tabular report in a new Google Sheet.
380   * This function handles sheet creation/clearing, header population,
381   * data insertion, and formatting.
382   * @param {Array<Array<any>>} tableData The data for the main report
383   *   ↳ table.
384   * @param {string} month The month for the report.
385   * @param {number} year The year for the report.
386   * @param {Date} bom The beginning of the month date.
387   * @param {Date} eom The end of the month date.
388   */
388  function generateTabularReportSheet(tableData, month, year, bom,
389   ↳ eom) {
389      var ss = SpreadsheetApp.getActiveSpreadsheet();
390      var reportSheetName = month + ' Budget Comparison';
391      var reportSheet = ss.getSheetByName(reportSheetName);
392
393      // Clear or create the report sheet.
394      if (reportSheet) {
395          reportSheet.clear(); // Clear existing content if sheet exists.
396      } else {
397          reportSheet = ss.insertSheet(reportSheetName); // Create new
398   ↳ sheet if it doesn't exist.
398      }
399
400      // Populate the top-right header section with report period
401   ↳ details.
401      reportSheet.getRange('C2').setValue("Year");
402      reportSheet.getRange('D2').setValue(year);
403      reportSheet.getRange('E2').setValue("BOM");
404      reportSheet.getRange('F2').setValue(Utilities.formatDate(bom,
405   ↳ Session.getScriptTimeZone(), "M/d/yyyy"));
406
406      reportSheet.getRange('C3').setValue("Month");
407      reportSheet.getRange('D3').setValue(month);
408      reportSheet.getRange('E3').setValue("EOM");
409      reportSheet.getRange('F3').setValue(Utilities.formatDate(eom,
410   ↳ Session.getScriptTimeZone(), "M/d/yyyy"));
411
411      // Output the main report table data starting at row 5, column 1.
412      var nRows = tableData.length;
413      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 }