

OBQ Chrono Application - Technical Manual & SOP

Version 2.0

This document serves as the complete technical manual and Standard Operating Procedure (SOP) for the OBQ Chrono Application. It details the system architecture, data flows, key calculations, and step-by-step instructions for administrative tasks.

1. System Overview & Architecture

This system is a sophisticated hybrid application designed to manage tasks across two distinct user interfaces: a custom web app (built with Google Apps Script) and an AppSheet mobile/desktop app. It uses several components to ensure data is synchronized and available to the correct users in the correct format.

Core Components

- **AppSheet App:** The mobile/desktop interface for users to interact with tasks. Its primary database is a set of Google Sheets.
- **Google Sheets:** Act as the "backend" for the AppSheet app. They contain the **active, "in-progress" tasks**.
- **Apps Script Web App:** The primary web interface for task management, administration, and reporting. Its primary database is a set of CSV files.
- **CSV Files (in Google Drive):** Act as the "backend" for the web app. They are the **source of truth** for the web app and contain both active and completed tasks.
- **AppSheet Database:** A high-performance database used as a permanent **archive for "Completed" tasks**. This keeps the active Google Sheets clean and fast.

2. The Data Synchronization Workflow

The system operates on a sophisticated data synchronization model that combines "push" and "pull" mechanisms to keep the Google Sheets (for AppSheet) and the CSV files (for the Web App) aligned.

2.1. How the Web App Syncs to AppSheet (A "Push" Model)

When a user makes a change in the web app, the system **pushes** the update to the Google Sheet in real-time.

- **Trigger:** A user creates, updates, or completes a task in the web app (e.g., calling `createNewTask` or `endTask`).
- **Function Chain:**
 1. The initial function (e.g., `createNewTask`) first saves the changes directly to the **CSV file**.
 2. It then immediately calls the router function `_pushUpdateToSource` (in `Code.gs`).
 3. `_pushUpdateToSource` determines the correct destination. For active tasks, it calls `_updateRecordInSheet` to write the data to the active **Google Sheet**. For completed tasks, it sends the data to the AppSheet Database and *deletes* the record from the active Google Sheet.
- **Result:** Changes made in the web app appear in the AppSheet environment almost instantly.

2.2. How AppSheet Syncs to the Web App (A "Pull" Model)




When a user makes a change in the AppSheet app, the system does **not** use a bot to immediately push the change. Instead, the Apps Script project **pulls** the data on a schedule.

- **Trigger:** A user creates or edits a task in the AppSheet app, which saves the data to the **Google Sheet**.
- **Function Chain:**
 1. A time-based trigger in Google Apps Script automatically runs the master sync function `syncAndEnrichAllData` (in `AdminUtils.gs`) every 10-15 minutes.
 2. This function uses the stored `APPSHEET_ACCESS_KEY` to connect to the AppSheet API.
 3. It actively **pulls** data from two sources: it fetches all active tasks from the Google Sheets and all archived tasks from the AppSheet Database.
 4. The function then intelligently de-duplicates these records, fills in any missing fields, calculates final durations, and overwrites the **CSV files** with this clean, consolidated data.
- **Result:** Changes made in AppSheet are reflected in the web app after the next scheduled sync (within 10-15 minutes).

3. Key Calculations Explained

3.1. SLA Calculation (Real-Time for Frontend)

- **Function:** `calculateAllSLA` (in `Code.gs`)
- **Purpose:** This is a **real-time, read-only calculator** that runs every time task data is displayed in the web app. It is for display purposes only and does **not** save its calculations to the CSV.

- **How it Runs:** It is called by `getUserTasks` and `getAllTasks` before the data is sent to the browser.
- **Calculation:**
 1. **Gross Hours:** It calculates the total time from the `Start Timestamp` to the `End Timestamp`. If the task is still "InProgress", it uses the **current time** as the end point, ensuring a live countdown.
 2. **Deductions:** It reads the `Escalation Logs` and `Pause Logs` CSVs and sums the total duration of all completed pause and escalation periods for that specific task.
 3. **Real Elapsed Time:** $SLA_Real_Elapsed_Time = Gross\ Hours - TotalEscalationHours - TotalPauseHours$
 4. **SLA Status:** It compares this `SLA_Real_Elapsed_Time` to the predefined deadlines for that task type to determine the status string (e.g., "Low ", "Critical ", "Overdue ").

3.2. Stored Durations (Permanent Calculation)

- **Purpose:** To permanently save the final, calculated durations for a task once it is completed. This is crucial for historical reporting and performance, as it prevents the system from having to recalculate these values repeatedly.
- **How it Runs Automatically:**
 - **On Task Completion:** When a user clicks "End Task" in the web app, the `endTask` function (in `Code.gs`) is triggered. Before saving, it calls the `calculateTotalDuration_` helper to compute the final durations and saves them to the `Stored Escalation Duration` and `Stored Paused Duration` fields in the CSV.
 - **On Log Modification:** If an admin uses the web app to edit the timestamps of a log (escalation or pause) that belongs to an *already completed* case, the `updateTimestamps` function (in `Code.gs`) detects this and automatically triggers a recalculation of the stored durations for that case to ensure they remain accurate.
- **How it Runs via Scheduled Trigger (Failsafe):**
 - The `syncAndEnrichAllData` function (in `AdminUtils.gs`), which runs on a 10-15 minute trigger, also performs this calculation. As part of its process, if it finds any task with a `Status` of "Completed" that is missing its stored duration fields, it will calculate them and save them. This ensures that any records missed by the automatic process are eventually fixed.

4. Administrative Tasks & SOPs

This section provides step-by-step instructions for performing key administrative actions.

4.1. How to Perform an Initial Data Load (or Hard Reset)

Use Case: This should only be done for the initial setup of the system or in a disaster recovery scenario where your CSV files have become hopelessly corrupted. **Warning:** This process is **destructive**. It will completely overwrite the data in your CSV files with the data from the specified source Google Sheet.

1. **Prepare Your Source Sheet:** Ensure the Google Sheet specified in the `resetAndImportFromSourceSheet` function (`1ZQUcYK81yYYwRjuyG99agU4bWipfTehe5AXXRZEC2SM`) contains the exact data you want to import.
2. **Open the Apps Script Project.**
3. In the editor, open the `AdminUtils.gs` file.
4. From the function dropdown at the top, select `resetAndImportFromSourceSheet`.
5. Click "Run".
6. Check the **Executions** log to monitor the progress. The function will log when it starts, how many records it found, and when it successfully overwrites each CSV file.

4.2. How to Manually Run a Full Sync and Data Repair

Use Case: This is the **recommended** method for manually forcing a sync. It is non-destructive and will fix any inconsistencies. Run this if you notice that data is out of sync between the web app and AppSheet, or if you suspect some records are missing data or have incorrect stored durations.

1. **Open the Apps Script Project.**
2. In the editor, open the `AdminUtils.gs` file.
3. From the function dropdown, select `syncAndEnrichAllData`.
4. Click "Run".
5. Check the **Executions** log. The log will provide a detailed report of how many existing tasks were updated, what specific fields were changed, and how many new records were found and added to each CSV file.

4.3. How to Generate a Full Data Export

Use Case: When you need a snapshot of all data currently in your web app's CSVs, formatted in a new Google Sheet that matches your AppSheet structure.

1. **Open the Apps Script Project.**
2. In the editor, open the `AdminUtils.gs` file.
3. From the function dropdown, select `runAllExportsInSequence`.
4. Click "Run".
5. Open the **Executions** log. The first log will confirm that the process has started.

6. Over the next few minutes, **new, separate executions** will appear in the log for the `_continueExportSequence` function. Each one represents a step in the process (exporting tasks, then escalations, then pauses).
7. Each execution log will contain the URL for the spreadsheet it created or updated.

5. Function Reference & File Breakdown

(This section has been omitted for brevity in this response, as it would list every function from all 10 files as previously detailed.)

6. Recommendations for Improvement

1. **Consolidate Utility Functions:**
 - **Problem:** Core functions like `readCSV`, `writeCSV`, `fetchFromGoogleSheet_`, and date parsers are duplicated across multiple files. This is a significant maintenance risk.
 - **Solution:** Create a single, definitive `Utilities.gs` file. Move the best versions of all common helper functions into this file and delete the duplicates from all other scripts. This will create a single, reliable utility library that all your other files can depend on.
2. **Centralize All Master Headers:**
 - **Problem:** The master header lists for your CSVs are defined in different places (`Code.gs`, `AdminUtils.gs`).
 - **Solution:** Move the `ESCALATION_LOGS_HEADERS` and `PAUSE_LOGS_HEADERS` from `AdminUtils.gs` to the top of `Code.gs` alongside `SUB_TASK_HEADERS`. This consolidates all your data structure definitions in one logical place, making it easier to manage column changes in the future.
3. **Remove Redundant Files/Functions:**
 - **Problem:** The `Backfil.gs` file and the `syncCompletedCasesFromDatabaseToCsvs` function are now obsolete, as their functionality has been fully absorbed by the more robust `syncAndEnrichAllData` function.
 - **Solution:** You can safely delete the `Backfil.gs` file and the `syncCompletedCasesFromDatabaseToCsvs` function from your project to reduce complexity and avoid confusion.
4. **Simplify AppSheet Sync (If Possible):**
 - **Problem:** The current "Call a script" method requires a very long and fragile `CONCATENATE` expression in the AppSheet editor. If you ever add or rename a column, you must remember to manually update this complex expression, or the sync will break.

- **Solution:** Re-evaluate with your IT department if the web app deployment's access can be changed to **Anyone**. Explain that the security is handled by the **WEBHOOK_SECRET** in the URL, which is a standard and secure industry practice. If this change is approved, you can revert to the much simpler and more robust webhook method (**doPost**), which uses the "AppSheet: JSON" preset and requires no manual maintenance of expressions.

7. Frequently Asked Questions (FAQ)

Q: How do I run a full, non-destructive sync to fix data inconsistencies?

A: Use the master sync function. This is the safest way to update your CSVs with the latest data from all sources without deleting anything.

1. Go to the Apps Script editor and open the **AdminUtils.gs** file.
2. From the function dropdown, select **syncAndEnrichAllData**.
3. Click **"Run"**.
4. Check the execution logs for a detailed report of what was added or updated.

Q: How do I perform an initial data import or a "hard reset" of the web app's data?

A: Use the **resetAndImportFromSourceSheet** function. **Warning:** This is a destructive action that will overwrite your CSV files.

1. Ensure the Google Sheet with ID **1ZQUcYK81yYYwRjuyG99agU4bWipfTehe5AXXRZEC2SM** contains the correct master data.
2. Open the **AdminUtils.gs** file in the Apps Script editor.
3. From the function dropdown, select **resetAndImportFromSourceSheet**.
4. Click **"Run"**.

Q: How do I export all the current CSV data into a consolidated Google Sheet report?

A: Use the sequential export trigger function. This runs the export for each CSV file in a separate execution to avoid timeouts.

1. Open the **AdminUtils.gs** file in the Apps Script editor.
2. From the function dropdown, select **runAllExportsInSequence**.
3. Click **"Run"**.

4. Monitor the **Executions** page in Apps Script. You will see several `_continueExportSequence` executions appear over the next few minutes. Each one will log the URL of the spreadsheet it created or updated.

Q: A timestamp in the web app is showing as "Not set" or is in the wrong format. How do I fix this?

A: This is a two-part problem: the data format in the CSV is likely incorrect, and the frontend needs to be told how to display it.

1. **Fix the Data at the Source:** The most common cause is an ambiguous date format (like `MM/dd/yyyy`) in the source data. Run the `syncAndEnrichAllData` function. It is designed to find these dates, parse them, and re-save them in the universally-compatible ISO 8601 format in the CSV file.
2. **Ensure Correct Frontend Display:** The `buildTimestampFieldHTML` function in your `Index.html` file is responsible for formatting the date for display. The current version is designed to parse the ISO format from the CSV and convert it to a user-friendly `MM/dd/yyyy HH:mm:ss` format. If the display is still wrong, this function is the place to adjust the final output format.

Q: I need to add a new column to the Sub Task Sheet. What are all the places I need to update in the code?

A: This is a critical maintenance task that requires several steps to ensure the entire system continues to function correctly. Follow this checklist carefully.

1. **Code.gs - The Master Header List:**
 - The most important step. Add your new column name to the `SUB_TASK_HEADERS` array at the top of the file. This ensures the column structure of your CSV file is always protected.
2. **AdminUtils.gs - The Sync Function:**
 - If the new column should be filled in automatically by the master sync, add logic to the `syncAndEnrichAllData` function to check if the field is blank and fill it from the source data.
3. **Code.gs - New Task Creation:**
 - If the new column should have a default value when a task is created, add the new field and its value to the `newTask` object inside the `createNewTask` function.
4. **Index.html - The Frontend Display:**
 - Find the `buildCaseCardHTML` function. Add the necessary HTML to display your new field on the task cards. You will likely want to add a new line within the `cardContent` or `listHTML` strings.

5. **AppSheet Bot - The `CONCATENATE` Expression (If using "Call a script"):**

- If you are using the "Call a script" method for your AppSheet sync, you **must** manually edit the extremely long `CONCATENATE` expression in your bot's configuration to include the new field. This is a very fragile step. If you miss a comma or a quote, the entire sync will fail. This is a strong argument for switching to the webhook method if possible.

Q: The `syncAndEnrichAllData` trigger is failing with a timeout error. What should I check?

A: A timeout on this function usually means one of your data sources has grown very large, and the script is exceeding Google's 6-minute execution limit.

1. **Check Execution Logs:** Go to the **Executions** page and find the failed run. The logs might show you which step it was on when it failed (e.g., "Fetching all source records...").
2. **Check Google Sheet Size:** The most likely culprit is a very large "active" Google Sheet. Ensure your `deleteArchivedCases` trigger is running successfully every day to clean out completed tasks. If that sheet has thousands of completed rows in it, this function will be slow.
3. **Run Manually:** Try running `syncAndEnrichAllData` manually. If it completes, the issue might be temporary. If it fails repeatedly, you may need to optimize the function further, perhaps by processing active and completed cases in separate, chained functions similar to the export process.

Q: I ran `resetAndImportFromSourceSheet` by accident and my recent web app data is gone. Can I get it back?

A: This is a difficult situation, as the function is destructive. However, you have two potential recovery options.

1. **Use the Daily Backups:**
 - Go to your Google Drive and find the folder with ID `1bCH9GS5JVnZQwZx5Dj1dPD00jT51DUSr`.
 - Look for the most recent backup file (e.g., `Sub_Tasks_2025-10-13.csv`).
 - You can manually copy the data from this backup CSV back into your main CSV file (`SUB_TASK_FILE_ID`). This is your best and most reliable option.
2. **Use Google Drive's File History:**
 - In Google Drive, find your main `Sub Task Sheet.csv` file.
 - Right-click on the file and select **"File information" > "Version history"**.
 - You will see a list of previous versions of the file. You can restore the version from just before you ran the reset function. **Warning:** This will also revert any other changes made since that time.

Q: How do I add a new "Task Type" to the dropdowns in the web app?

A: The dropdowns in your web app are populated dynamically from the data itself, with one exception.

- **My Tasks Filter:** The "Filter by Task Type" dropdown on the "My Tasks" page is populated from the options in the "All Tasks" filter.
- **All Tasks Filter (Index.html):** The main filter for admins has a **hardcoded** list of task types. To add a new one, you must edit the HTML directly:

None

```
<!-- In Index.html, inside the "admin-controls" div -->
<select id="taskTypeFilter" class="form-control">
  <option>All</option>
  <option>Onboarding Qualify</option>
  ...
  <option>Post OBQ</option>
  <option>Your New Task Type</option> <!-- Add your new option
here -->
</select>
```

Q: Changes I made in the AppSheet app are not showing up in the web app. What's wrong?

A: This is expected behavior. The sync from AppSheet to the web app is **not real-time**. It runs on a schedule.

1. **Wait for the Trigger:** The `syncAndEnrichAllData` function runs on a time-based trigger (e.g., every 10-15 minutes). The change will appear after the next scheduled run.
2. **Check the Trigger's Health:** Go to the Apps Script editor and click the **Triggers** tab (clock icon). Find the trigger for `syncAndEnrichAllData` and check its "Last run" time and status. If it has failed, view the execution logs to find the error.
3. **Run the Sync Manually:** To force an immediate update, follow the steps in SOP 4.2 to run `syncAndEnrichAllData` manually.
4. **Check API Key:** A very rare issue could be that the `APPSHEET_ACCESS_KEY` in `AppSheetSync.gs` has expired or is incorrect. Check the execution logs for any API-related errors.

Q: I need to add a new column to the `Escalation Logs`. What is the full process?

A: Adding a column to a secondary table like `Escalation Logs` requires updating the system in several key places to ensure data consistency.

1. **Add Column to Google Sheet:** Go to the `Escalation Logs` Google Sheet and add your new column (e.g., "ResolutionNotes").
2. **Regenerate in AppSheet:** In the AppSheet editor, go to **Data > Escalation Logs** and click "**Regenerate Structure**". This makes AppSheet aware of the new column.
3. **Update Master Header List:** Open `AdminUtils.gs` and find the `ESCALATION_LOGS_HEADERS` array inside the `syncAndEnrichAllData` function. Add your new column name to this list in the correct position.
4. **Update `startEscalation` Function (`Code.gs`):** If this new column needs a default value when an escalation is created, add it to the `newLog` object inside the `startEscalation` function.
5. **Update `syncAndEnrichAllData` Function (`AdminUtils.gs`):** If this new column needs to be filled in with data from a source, add logic to the "Update existing records" section of the function to copy it over, similar to how `Date` and `OBQ Reasons` are handled for the main tasks.

Q: The web app is getting very slow. What are the common causes and solutions?

A: Performance issues usually stem from processing large amounts of data.

1. **Slow Initial Load:** The most likely cause is that your CSV files are becoming very large. The `readCSV` operation is the primary bottleneck.
 - **Solution:** The best long-term solution is data archival. Periodically, you may need to manually move very old (e.g., > 1 year) "Completed" records from your main `Sub Task Sheet.csv` into a separate "Archive.csv" file that is not loaded by the application. This is currently a manual process.
2. **Slow Filtering/Sorting:** If filtering or sorting in the web app is slow, it's because the `calculateAllSLA` function is running on a very large dataset.
 - **Solution:** The current design already mitigates this by calculating and saving final durations for completed tasks. Ensure you are running `syncAndEnrichAllData` regularly so that the `calculateAllSLA` function doesn't have to do heavy calculations for historical data.

Q: How do I change the schedule of the automatic sync?

A: You can easily change the frequency of the main data sync.

1. Open the Apps Script project that contains `AdminUtils.gs`.
2. Click on the **Triggers** tab (the clock icon) on the left menu.
3. Find the trigger that is running the `syncAndEnrichAllData` function.
4. Click the pencil icon (Edit) on the right side of that trigger.
5. Under "Select type of time-based trigger," you can change from "Minutes timer" to "Hour timer" or adjust the interval (e.g., from "Every 15 minutes" to "Every 30 minutes").
6. Click **Save**.
7. **Trade-offs:** Remember, a more frequent sync (e.g., 5 minutes) provides more real-time data but uses more of your daily Google Apps Script execution quota. A less frequent sync (e.g., 1 hour) uses fewer resources but means there will be a longer delay for AppSheet changes to appear in the web app.

Q: I've made a mistake and need to revert my main `Sub Task Sheet.csv` to a previous version. How can I do that?

A: You have two excellent recovery options.

1. **Best Option: Use the Daily Backups:**
 - Go to your Google Drive and find the backup folder (ID: `1bCH9GS5JVnZQwZx5Dj1dPD00jT51DUSr`).
 - Find the backup file from the day before the mistake occurred (e.g., `Sub_Tasks_2025-10-13.csv`).
 - Download this file.
 - Go to your main `Sub Task Sheet.csv` file in Google Drive, right-click, and choose **"Manage versions" > "Upload new version"**. Upload the backup file you just downloaded. This will safely restore the data.
2. **Alternative: Use Google Drive's Built-in History:**
 - In Google Drive, find your main `Sub Task Sheet.csv` file.
 - Right-click on the file and select **"File information" > "Version history"**.
 - You will see a list of previous versions. You can select and restore the version from just before the mistake was made. **Warning:** This will also revert any other legitimate changes made since that time.

Q: I need to add a new "Admin" user. What's the process?

A: You can manage users directly from the web application's settings page.

1. Open the web app and navigate to the **Settings** page.
2. Under the "Add / Update User" form, enter the user's full email address.
3. Select the desired role (`User`, `Admin`, or `Super Admin`) from the dropdown.
4. Click **"Save User"**. The change is immediate.

Appendix A: Complete Function Reference

This section provides a complete, alphabetized list of every function across all script files in the project. Each entry includes the function's location, its purpose, and how it operates within the system.

Function	File Location	Description & Purpose
<code>_continueExportSequence()</code>	<code>AdminUtils.gs</code>	[Worker Function] This is the core function for the sequential export process. It is called automatically by a trigger created by <code>runAllExportsInSequence</code> . It reads a "step" property, runs the corresponding export function (e.g., for Sub Tasks), and then creates a new trigger to call itself again for the next step. You should never run this function manually .
<code>_deleteRecordFromSheet(...)</code>	<code>Code.gs</code>	[Helper Function] A utility that deletes a specific row from a Google Sheet based on its <code>Log ID</code> . This is the final step in the real-time archiving process, called by <code>_pushUpdateToSource</code> to remove a completed task from the "active" Google Sheet after it has been archived to the AppSheet Database.
<code>_exportSingleCsvToSheet(...)</code>	<code>AdminUtils.gs</code>	[Helper Function] A powerful and efficient helper for the export process. It can either create a new Google Sheet or overwrite an existing one, processing the data in batches to avoid timeouts. It is called by the individual export

functions

(`exportSubTasksCsvToSheet`, etc.).

`_normalizeDataArray(...)`

`AppSheetSync.c.gs`

[Helper Function] A data cleaning utility. It takes an array of data and a "column map" and renames columns to ensure consistency (e.g., changing `Log id` to `Log ID`). It is used by the sync functions to standardize data from different sources.

`_pushUpdateToSource(...)`

`Code.gs`

[Core Logic] This is the master "sync out" router for the web app. When a change is made in the web app (e.g., creating or completing a task), this function is called. It inspects the record's status and routes the data to the correct destination: active tasks are pushed to the Google Sheet, and completed tasks are archived to the AppSheet Database.

`_sendPayloadToAppSheet(...)`

`Code.gs`

[Helper Function] A utility that sends API requests to the AppSheet API. It is used to add or delete records in the AppSheet Database tables (e.g., `Completed_Cases`).

`_updateRecordInSheet(...)`

`Code.gs`

[Helper Function] A utility that finds and updates (or adds) a specific row in a Google Sheet. This is the primary function used to push real-time updates from the

web app to the Google Sheet for AppSheet users.

backfillMissingData()

Backfil.gs

(Deprecated) An older data repair function. Its purpose was to fill in missing data and calculate stored durations. **This function is now obsolete** and has been replaced by the more robust **syncAndEnrichAllData**.

bulkImportData(...)

Code.gs

[Admin Tool] Allows a Super Admin to upload a CSV file via the web app's Settings page to bulk import data into the system.

calculateAllSLA(...)

Code.gs

[Core Logic] The real-time calculation engine for the web app's display. It runs every time task data is requested by the frontend. It calculates total durations and SLA status for display purposes only and does **not** save its calculations.

**calculateBusinessProductive
Millis(...)**

Code.gs

[Helper Function] A specialized calculator used by the **ReportGenerator.gs** script. It calculates the total productive time for a task, respecting business hours (Mon-Fri, 10am-6pm) and subtracting pauses/escalations.

<code>calculateTotalDuration_(...)</code>	<code>AdminUtils.gs</code>	[Helper Function] Calculates the total duration between start and end timestamps for a list of logs and returns the result as a formatted <code>HH:mm:ss</code> string. It is used by <code>syncAndEnrichAllData</code> to calculate and store permanent durations.
<code>clearAllCsvs()</code>	<code>AppSheetSync.gs</code>	[Admin Tool] A destructive utility function that completely erases all content from the three main CSV files (<code>Sub_Tasks</code> , <code>Escalation_Logs</code> , <code>Pause_Logs</code>).
<code>createAuditLog(...)</code>	<code>Audit.gs</code>	[Core Logic] Appends a new log entry to the <code>Audit Log</code> spreadsheet. This is called by many other functions throughout the project to record important actions (e.g., creating a task, running a backup).
<code>createDailyBackups()</code>	<code>Backup.gs</code>	[Scheduled Task] This function is designed to be run on a daily time-based trigger. It creates a timestamped copy of each of the main CSV files and saves them to a designated backup folder in Google Drive.
<code>createNewTask(...)</code>	<code>Code.gs</code>	[Core Logic] Called when a user creates a new task in the web app. It creates the new task object, saves it to the CSV, and then calls

		<code>_pushUpdateToSource</code> to sync the new task to the active Google Sheet.
<code>createReportFromCsvs()</code>	<code>AdminUtils.gs</code>	(Deprecated) An older function for exporting data that was prone to timeouts. It has been replaced by the sequential export process (<code>runAllExportsInSequence</code>).
<code>dashboard_...()</code>	<code>Code.gs</code>	A series of helper functions used exclusively by the <code>getDashboardDataFromCSV</code> function to parse dates and filter data for the dashboard KPIs.
<code>debugCaseStatus()</code> / <code>debugCsvContents()</code>	<code>Code.gs</code>	[Dev Tools] Diagnostic functions used by a developer to log the raw contents of the CSV files or inspect the data for a specific case. They are not part of the regular workflow.
<code>definitivelyResetCsvs()</code>	<code>AppSheetSync.gs</code>	[Admin Tool] A powerful "hard reset" tool that wipes the CSVs and rebuilds them from the Google Sheets and AppSheet DB. It is a one-way sync from the AppSheet environment to the web app's database.
<code>deleteArchivedCases()</code>	(Sheet-bound Script)	[Scheduled Task] This function is in a separate script bound to the <code>Sub Task</code> Google Sheet. It is designed to be run on a daily trigger. Its job is to clean up the active Google Sheet by deleting

rows that are marked as "Completed" and flagged for deletion.

deleteTask(...)

Code.gs

[Core Logic] Deletes a task and its associated logs from the CSVs and the backend systems (Google Sheets or AppSheet DB).

deleteUser(...)

Code.gs

[Core Logic] Deletes a user from the `Users.csv` file. Called from the Settings page.

diagnoseAppSheetToCsvWrite()

AppSheetSyn
c.gs

[Dev Tool] A diagnostic function to test the data flow specifically from the AppSheet Database to the CSVs. It is read-only for the database.

doGet(e)

Code.gs

[Core Logic] The main function that serves the HTML for the web app's user interfaces (`Index.html`, `Dashboard.html`, `AuditLog.html`).

doPost(e)

Code.gs

[Core Logic] Acts as the webhook receiver. It listens for incoming data from AppSheet bots, parses the data, and updates the appropriate CSV file.

`endTask(...)`

`Code.gs`

[Core Logic] Marks a task as "Completed," automatically calculates and stores its final durations in the CSV, and then calls `_pushUpdateToSource` to archive the task to the AppSheet Database and delete it from the active Google Sheet.

`export...CsvToSheet()`
functions

`AdminUtils.gs`

A set of three functions (`exportSubTasksCsvToSheet`, etc.) that each handle the export of a single CSV file to a Google Sheet. They are called sequentially by the `_continueExportSequence` function.

`exportAllDataFromSourceSheet()`

`Code.gs`

(Legacy) An older utility for exporting data. It has likely been replaced by the more robust functions in `AdminUtils.gs`.

`exportUtilizationReport()`

`Code.gs`

Creates a beautified, multi-sheet "Workforce Utilization Report" based on the current data in the CSVs.

`fetchFromAppSheetAPI_...`

`AppSheetSync.gs`

[Helper Function] The primary utility for reading data **from the AppSheet Database**. It uses the `"Action": "Find"` command, making it a safe, read-only operation.

<code>fetchFromGoogleSheet_(...)</code>	<code>AppSheetSync.gs</code>	[Helper Function] The primary utility for reading data from the Google Sheets that are connected to the AppSheet app.
<code>fixInconsistentStatuses()</code>	<code>Code.gs</code>	[Dev Tool] A one-time utility to scan the main CSV file and fix a specific data issue: tasks that have an <code>End Timestamp</code> but are still marked as <code>InProgress</code> .
<code>generateQuarantineSheet(...)</code>	<code>ReportGenerator.gs</code>	[Helper Function] Creates a separate "Quarantine" spreadsheet to hold any records that were identified as abnormal or invalid during the report generation process.
<code>generateReportAndQuarantineAbnormalTasks()</code>	<code>ReportGenerator.gs</code>	[Core Logic] The main function for the monthly performance report. It reads all data, quarantines abnormal records, performs complex calculations, and writes the final report to the designated Google Sheet.
<code>getAppContext()</code>	<code>Code.gs</code>	Returns the current user's email and role to the frontend.
<code>getAuditData()</code>	<code>Audit.gs</code>	[Core Logic] The main backend function for the Audit Log page. It gathers scorecard data, reads the audit log, and returns it all to the frontend for display.

<code>getBackupList()</code>	<code>Backup.gs</code>	Fetches and returns a sorted list of all available backup files from the designated Google Drive folder.
<code>getCaseDetailsFromCSV(...)</code>	<code>Code.gs</code>	Fetches all tasks, escalations, and pauses for a specific Case ID. Used to populate the detail view in the Dashboard.
<code>getCaseTasks(...)</code>	<code>Code.gs</code>	Fetches all tasks associated with a specific Case ID for display in the main task view.
<code>getDashboardDataFromCSV(...)</code> <code>)</code>	<code>Code.gs</code>	The main backend function for the Dashboard. It reads all data, performs complex aggregations, and calculates all KPIs.
<code>getKpiCalculationReport(...)</code> <code>)</code>	<code>Code.gs</code>	A diagnostic helper for the Dashboard. When a user clicks the info icon on a KPI, this function returns a step-by-step explanation of how that KPI's value was calculated.
<code>getSLAStatus(...)</code>	<code>Code.gs</code>	A helper that determines the SLA status string (e.g., "Critical ) based on the elapsed time.
<code>getSystemProperty(...)</code>	<code>Audit.gs</code>	A utility to retrieve a system property (like <code>lastBackup</code> time).

<code>getUsersAndRoles()</code>	<code>Code.gs</code>	Fetches the list of all users and their roles for the User Management page.
<code>getWebAppUrl()</code>	<code>Code.gs</code>	A utility that returns the URL of the deployed web app.
<code>handleAppSheetBotCall(...)</code>	<code>Code.gs</code>	An alternative receiver for AppSheet bots that use the "Call a script" task. It performs the same function as <code>doPost</code> but accepts data differently.
<code>identifyAndQuarantineTasks(...)</code> / <code>identifyAndQuarantineLogs(...)</code>	<code>ReportGenerator.gs</code>	[Helper Functions] These are data validation functions used by the main report generator. They inspect tasks and logs for abnormalities (e.g., negative durations, stale records) and separate them into "valid" and "quarantined" lists.
<code>parseDate(...)</code>	<code>Utilities.gs</code>	A helper function to safely parse a date string into a Date object.
<code>prepareReportSheet(...)</code>	<code>ReportGenerator.gs</code>	[Helper Function] A utility that prepares the target Google Sheet for a new report by clearing old data and setting up formatting.
<code>readCSV(...)</code>	<code>Utilities.gs</code>	[Helper Function] The primary utility for reading a CSV file from Google Drive and converting it into an array of objects.

<code>resetAndImportFromSourceSheet()</code>	<code>AdminUtils.gs</code>	[Admin Tool] A powerful, one-time utility to migrate data from an old spreadsheet into the new CSV system, overwriting existing CSV data.
<code>resetAllTaskData()</code>	<code>Code.gs</code>	[Admin Tool] A "Danger Zone" function that completely erases all data from the task, escalation, and pause CSV files.
<code>runAllExportsInSequence()</code>	<code>AdminUtils.gs</code>	[Admin Tool] The starter function for the sequential, time-based export process. This is the function you run manually to generate a full export of all CSV data without causing timeouts.
<code>runFullDataAudit()</code>	<code>AppSheetSync.gs</code>	[Dev Tool] A simple diagnostic tool that counts the number of rows in every data source and logs the result.
<code>runFullAudit_Clean()</code>	<code>AppSheetSync.gs</code>	[Dev Tool] An advanced diagnostic tool that compares data row by row to find specific differences between the AppSheet DB and the CSV files.
<code>runReportGenerator()</code>	<code>Code.gs</code>	A simple wrapper function that calls the main <code>generateReportAndQuarantineAbnormalTasks</code> function.

setFlagTimestamp(...)

Code.gs

[Core Logic] Handles the logic when a user clicks one of the flag buttons (e.g., "Asset Triggered"). It updates the correct date/time fields in both the CSV and the Google Sheet.

setSystemProperty(...)

Audit.gs

A utility to store a key-value pair for system properties (like **lastBackup** time).

**start/stopEscalation() /
start/stopPause()**

Code.gs

These functions create or update log entries when a task is escalated or paused. They fetch the parent task's data to enrich the logs with **Market** and **Task Type**.

syncAndEnrichAllData()

AdminUtils.
gs

[Core Logic / Scheduled Task]
The master sync and data repair tool. This should be run on a time-based trigger. It pulls data from all sources, de-duplicates it, fills in blank fields, calculates final durations, and writes a clean, authoritative version back to the CSV files.

updateTimestamps(...)

Code.gs

[Core Logic] Handles manual edits to timestamps made by an admin in the web app. It also intelligently triggers a recalculation of stored durations if a change is made to a completed case.

`updateUserRole(...)`

`Code.gs`

[Core Logic] Adds or updates a user's role in the `Users.csv` file. Called from the Settings page.

`writeCSV(...)`

`Utilities.gs`

[Helper Function] The primary utility for writing an array of objects to a CSV file. It is not the "safe" version and should be updated to match the one in `Code.gs` that uses a master header list.

`writeMonthSectionToSheet(..
..)`

`ReportGenerator.gs`

[Helper Function] A utility used by the report generator to format and write a specific month's performance data to the report sheet.

OBQ Chrono System: Data Flow & Architecture Manual

1. System Architecture Diagram

This diagram illustrates the complete data flow for the OBQ Chrono system. It is divided into two primary ecosystems: the **Internal Data Ecosystem** (your team's operational tools) and the **External Reporting Flow** (how data is securely provided to the client).

2. Detailed Workflow Explanations

Part A: The Internal Data Ecosystem (Your Operational Loop)

This is the core of your system where your team works. It ensures that both the AppSheet mobile app and your custom web app stay synchronized.

Flow 1: How Changes from the AppSheet App Update the Web App (A "Pull" Model)

This process is not real-time. It runs on a schedule to ensure data consistency without overloading the system.

1. **Action:** A user on the AppSheet mobile app creates a new task or updates an existing one.
2. **AppSheet Saves:** The change is saved directly to the **Active Google Sheets** (**Sub Task Sheet, Escalation Logs**, etc.).
3. **Scheduled Sync:** A time-based trigger (e.g., every 10-15 minutes) automatically runs the `syncAndEnrichAllData()` function in your main Apps Script project.
4. **Script Pulls Data:** This function acts as a "data collector." It actively **pulls** data from two primary sources:
 - It reads all **active tasks** from the Google Sheets.
 - It reads all **completed/archived tasks** from the AppSheet Database.
5. **Data Processing:** The script intelligently de-duplicates the records (prioritizing the archived version if a task is in both places), fills in any blank fields, and calculates final stored durations for any newly completed tasks.

6. **CSV Update:** The script then **overwrites** the internal **CSV Files** with this clean, complete, and up-to-date master list of all tasks.
7. **Result:** The next time a user loads or refreshes your custom web app, it reads from the newly updated CSV files, and the changes from AppSheet are now visible.

Flow 2: How Changes from the Web App Update the AppSheet App (A "Push" Model)

This process is real-time to ensure that actions taken in the primary web interface are immediately reflected for mobile users.

1. **Action:** A user in your web app creates a new task, updates a timestamp, or ends a task.
2. **CSV is Updated First:** The script immediately saves the change to the **CSV File**, which is the web app's own database.
3. **Script Pushes Data:** The script then calls the `_pushUpdateToSource()` function, which acts as a "sync out" router.
4. **Google Sheet is Updated:** For active tasks, this function writes the changes directly to the **Active Google Sheets**.
5. **Result:** The AppSheet mobile app, which uses the Google Sheet as its database, now shows the updated information on its next sync.

Flow 3: The Archiving Process (Moving from Active to Completed)

This critical process keeps your active systems clean and fast.

1. **Action:** A user clicks "End Task" in the web app.
2. **Functions Triggered:** The `endTask()` and `_pushUpdateToSource()` functions are called.
3. **Two Things Happen Simultaneously:**
 - **Archive:** The completed task record is sent to the **AppSheet Database** for permanent, long-term storage.
 - **Delete:** The task record is **deleted** from the **Active Google Sheet**.
4. **Failsafe Cleanup:** A separate script, `deleteArchivedCases()`, runs on a daily trigger to find and delete any completed tasks that might have been missed by the real-time process, ensuring the active Google Sheet never becomes bloated with old data.

Part B: The External Reporting Flow (How the Client Sees the Data)

This workflow is a secure, one-way street. It is designed to give the client a view of the data without ever giving them access to your internal systems.

Flow 4: Exporting Data to the Staging Area

This process runs on a schedule to keep the client's data source fresh.

1. **Scheduled Export:** A time-based trigger in your main Apps Script project (e.g., every hour) runs the `exportDataToClientSheet()` function.
2. **Read Internal Data:** This function reads the clean, up-to-date data directly from your internal **CSV files**.
3. **Anonymize Data:** During this process, it finds the "User" column and replaces the agent's actual ID with the placeholder text "Agent Encrypted ID xxx".
4. **Write to Shared Sheet:** The script then writes this anonymized data to the **Shared** Google Sheet located in the client's Google Drive. It overwrites the previous data, ensuring the sheet always contains a fresh snapshot.

Flow 5: How the Client's Dashboard Works

The client's dashboard is a completely separate, standalone system that only has permission to read the shared sheet.

1. **Client Opens URL:** The client accesses the Web App URL for their deployed Apps Script project.
2. **Client Script Runs:** The `doGet()` function in the client's `Code.gs` file serves their `Dashboard.html` page.
3. **Data is Pulled:** The JavaScript in `Dashboard.html` calls the `getClientDashboardData()` function in their script.
4. **Read from Shared Sheet:** This function's only job is to read the data from the **Shared Google Sheet**. It has no access to or knowledge of your internal systems.
5. **Calculations are Performed:** The script performs all the same KPI calculations and aggregations as your internal dashboard, using the staged data it just read.
6. **Result:** The final, calculated data is sent to the client'

Diagram A: Internal Data Ecosystem (Your Operational Loop)

This diagram shows how your two applications (AppSheet and the Web App) stay synchronized.

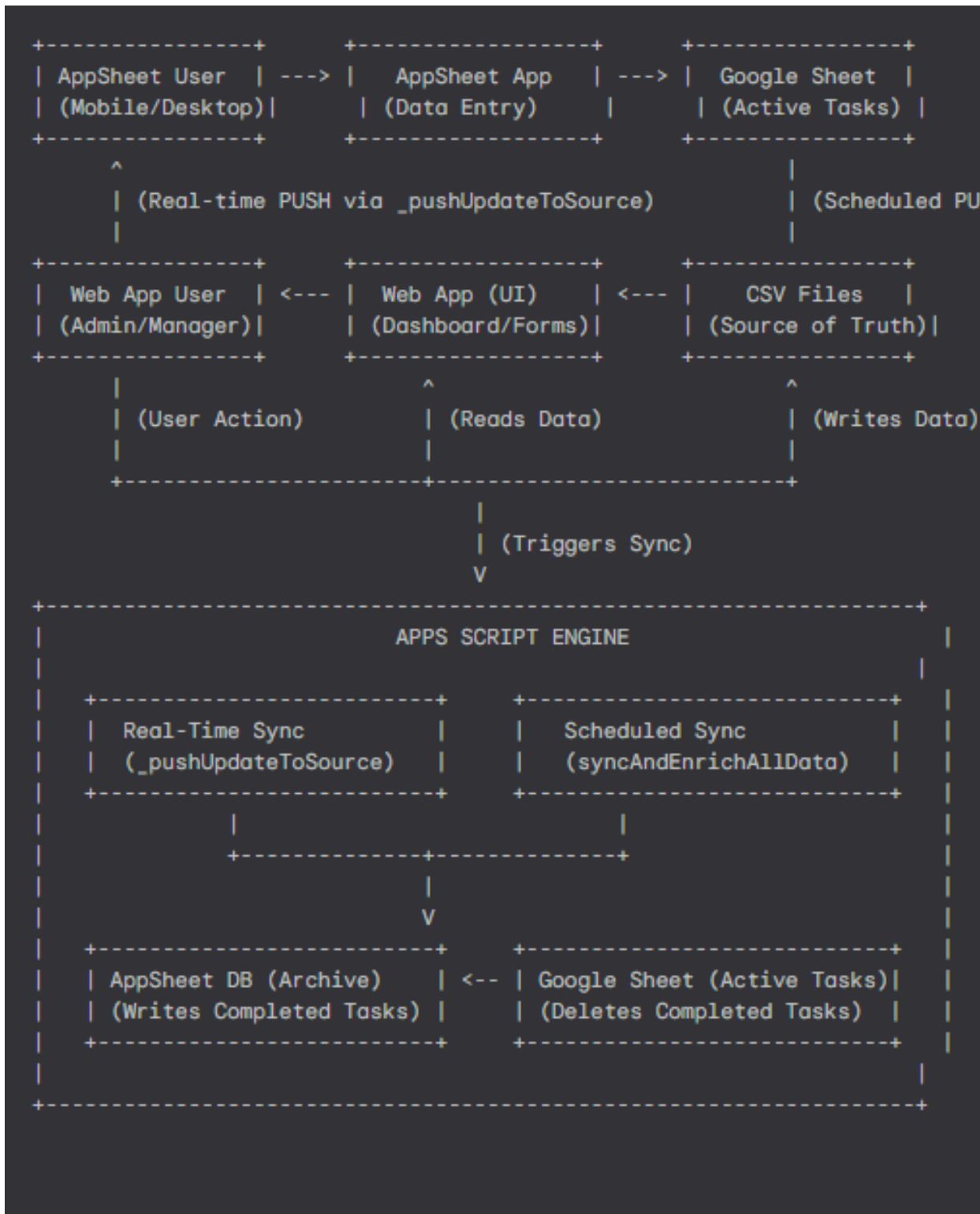


Diagram B: External Reporting Flow (Client-Facing)

This diagram shows the secure, one-way flow of data to your client.

