ChatGPT

Operation Summer – Technical Specification and Project Documentation

Frontend (Flutter)

The Operation Summer app's frontend is built with **Flutter**, enabling a single codebase for iOS, Android, and Web. The UI is organized into four main sections: a **Home Dashboard**, a **Chore Board** (Kanban-style task board), a **Calendar View**, and a **Learning Hub**. Each section is implemented as a separate screen with a cohesive widget hierarchy, and the app uses Flutter's navigation (e.g. a BottomNavigationBar or drawer) to allow quick switching between these features. The design emphasizes an intuitive, family-friendly interface, with real-time data updates via Firebase and smooth integration of Google services for calendar and document handling.

UI Features Overview

- **Home Dashboard:** The landing screen provides an at-a-glance overview of the day. It might greet the user (parent or child) and display key summary info: today's date, total points earned so far by the child, and quick links to pending chores and lessons. If multiple children profiles exist, the dashboard allows selecting a profile (e.g. a dropdown or avatar switcher for each child). The dashboard can show a progress snapshot (e.g. X of Y chores done today, or a progress bar) and perhaps a **"Today's Schedule"** snippet that lists upcoming events or lessons. It serves as a hub with navigation cards/buttons leading to the Chore Board, Calendar, and Learning Hub.

- **Chore Board (Kanban):** The chore board screen presents chores in a Kanban-style layout with columns such as **To Do**, **In Progress**, and **Done**. Each chore is a card (a draggable widget) displaying its title, an icon or image (if applicable), and possibly the point value of that chore. Users (typically children) can drag and drop chores between columns to update their status. For example, a child might drag a task from "To Do" to "Done" once completed, which triggers an update in the backend. This provides a visual, interactive way to manage tasks [1]. The Chore Board supports reordering tasks within a column and moving tasks across columns via Flutter's drag-and-drop framework or a dedicated package (e.g. the `kanban_board` Flutter package) [1]. The UI updates in real-time as chore data changes (e.g. if a parent assigns a new chore from another device, it appears instantly). A floating action button ("+") on this screen allows adding a new chore, bringing up a dialog to input chore details (description, due date, point value, etc.).

- **Calendar View:** The calendar screen embeds a functional calendar that reads and writes events from **Google Calendar**. It provides a familiar month/week view of events, allowing users to see scheduled activities (e.g. outings, deadlines, or scheduled lessons). Key feature is drag-and-drop rescheduling – users can press and drag an event to a different date or time slot to quickly reschedule [2]. The calendar UI will likely utilize a Flutter calendar widget (for example, Flutter's `TableCalendar` or Syncfusion's `SfCalendar`) that supports displaying multiple events per day and drag-and-drop interaction [2]. When an event is dragged to a new slot, the app will prompt an update to the event's schedule. The Calendar View can also overlay chore deadlines or lesson times: e.g. if a chore has a due time or a lesson is scheduled at 3 PM, those appear on the calendar alongside Google Calendar events. Users can tap on a day to see a detailed list of events/chores for that day, and possibly add new events. Any changes (adding, editing, or moving events) sync back to Google Calendar through the appropriate API calls in the background.

- **Learning Hub:** The learning hub is a section for daily lessons or educational content. It might be organized by date or subject. For each day of "Operation Summer," the parent can assign **daily lessons** (reading materials, worksheets, activities) which are stored in the app. The Learning Hub screen could show a list of lessons (with titles or thumbnails) for the selected day or child. Each lesson item can include a description and a link to a resource (PDF or video). Tapping a lesson opens a detail page where the content can be viewed or downloaded – for example, opening a PDF worksheet within the app. The UI may indicate completion status of each lesson (e.g. a checkmark once the child marks it done or the parent verifies it). Like chores, lessons can have points or rewards for completion, which tie into the profile's point system. The learning hub provides quick actions to **Preview** or **Download** attached PDFs, and a **"Print All"** button to print all lesson materials for the day (more on this below). This screen ensures educational tasks are as engaging as chores, potentially including multimedia content or quizzes (if extended).

Component Structure and Widget Hierarchy

The Flutter app is organized into modular widgets for maintainability. At the top level, the app uses a `MaterialApp` (with theming) and a main `Scaffold` that holds common UI elements (like an AppBar and a bottom navigation bar to switch between the main sections). Each main feature is implemented in a separate screen widget, and within each screen, the UI is further broken down into reusable components. Below is a high-level widget hierarchy outline:

- **MainApp (StatefulWidget)** – Initializes Firebase, handles user authentication state, and defines the navigation structure.
- `MaterialApp` -> `HomePage` (if logged in) or `LoginPage` (if not authenticated).
- `LoginPage` – Uses Firebase Auth UI or custom form to sign in/up the parent user (with email/password or Google sign-in).
- `HomePage` (StatefulWidget) – Main screen with bottom navigation.
  - `Scaffold` (AppBar with app title and profile switcher, BottomNavBar with sections: Dashboard, Chores, Calendar, Learning).
  - `DashboardScreen` (widget for the Home Dashboard content).
  - `ChoreBoardScreen` (widget for chores, appears when Chores tab selected).
  - `CalendarScreen` (widget for calendar, for Calendar tab).
  - `LearningHubScreen` (widget for lessons, for Learning tab).

Each screen is composed of smaller widgets: - **DashboardScreen**: could be a `SingleChildScrollView` or Column containing widgets like `ProfileHeader` (shows selected child's name, points, and a profile picker), `TodaySummary` (a card summarizing today's chore/lesson progress), and shortcut buttons or cards linking to other sections (e.g. a "Go to Chore Board" card showing count of remaining chores). - **ChoreBoardScreen**: implemented as a row of **Column widgets** (or a custom BoardView). For example: - A `Row` with three `Expanded` columns: one for each status ("To Do", "In Progress", "Done"). Each column has a header (Text) and a list of chore cards beneath. - Each chore card is a custom widget (`ChoreCard`) displaying the task name, maybe an icon, and points. `ChoreCard` could have a visual indicator if it's draggable (e.g. a grip icon). - The chore cards are made draggable via Flutter's **Draggable** widget. The columns act as **DragTarget** to accept dropped cards and update state. - For improved structure, we can use a ready-made Kanban widget or BoardView which internally manages the drag targets for us. - The state of the ChoreBoardScreen holds the list of chores grouped by status (fetched from Firestore). It listens to Firestore updates and rebuilds when data changes. - **CalendarScreen**: contains a calendar widget view (month or week). For example, using Syncfusion's `SfCalendar`: - `SfCalendar(view: CalendarView.month, dataSource: EventDataSource(events), allowDragAndDrop: true, onDragEnd: handleDragEnd, onTap: handleEventTap)` [2]. - The `EventDataSource` provides events to display (populated from Google Calendar API and any Firestore events). - If using `TableCalendar` (a simpler widget), we'd manually display markers for days with events and show a list on date tap. - Possibly a toggle to switch view (month <-> week). In a week/day view, events could render as blocks that can be dragged along the timeline (supported by some calendar widgets natively). - A floating action button "+" might allow adding a new event (opening a dialog with fields like title, date/time). - **LearningHubScreen**: could use a vertical `ListView` of lesson items: - Each lesson item is a card or ListTile with title and status icon. If a PDF is attached, maybe a thumbnail or PDF icon is shown. - There may be a date selector at top (e.g. a calendar picker or a horizontal list of dates/weekdays) to choose which day's lessons to view (defaulting to today). - Tapping a lesson opens a `LessonDetailPage` where the lesson description is shown and an embedded PDF viewer if applicable. - On `LessonDetailPage`, there are actions like **Preview PDF**, **Download PDF** (saves to device), and **Print**. - Alternatively, for multiple PDFs, the Learning Hub main screen might allow selecting multiple or simply have a "Print All" to send all today's lesson PDFs to print.

Across these UI components, **State Management** is crucial. We will likely use a state management solution like Provider or Riverpod to manage shared state (e.g. the currently selected child profile, or loaded data lists). For instance, a `ProfileProvider` can hold the current child's ID and info, so that all screens know which child's data to show. Firestore's real-time streams can feed directly into `StreamBuilder` widgets for lists of chores and lessons, keeping the UI live-updated. For more complex logic (e.g. handling drag-and-drop reordering or syncing with external APIs), we might employ the BLoC pattern or controller classes to separate UI from logic.

Integration with Firebase and Google Services

Firebase Auth: User authentication is handled via Firebase. The parent (administrator) will create an account (likely using email/password signup, or alternatively using Google Sign-In through Firebase). We integrate Firebase Authentication using the `firebase_auth` Flutter plugin. Upon successful login, the app navigates to the HomePage and loads the user's data (child profiles, chores, etc. from Firestore). We enforce that certain screens or actions are only available to authenticated users. If using Google Sign-In (to facilitate Google Calendar/Sheets access), we link the Google OAuth provider to the Firebase account. This way, the user can sign in once with Google and gain both Firebase auth and Google API authorization tokens in one go. Role management is simple on the client side: by default the logged-in user is assumed to be the parent (with full privileges). If we allow children to log in separately (see Backend roles below), the UI will adapt to a limited mode (e.g. not showing admin-only features like adding new chores). The app could use Firebase's custom claims or a Firestore flag to determine the role and adjust the UI (for example, hide the "add chore" button if the role is "child").

Cloud Firestore Data: The app uses Cloud Firestore as the primary online database for app data (profiles, chores, lessons, etc.). The Flutter app integrates via the `cloud_firestore` plugin. Data flows are generally as follows: - On app launch/refresh, the app queries Firestore for the current user's children profiles and initial data. This can be done with a query like `FirebaseFirestore.instance.collection("users/{uid}/children").get()` (or an equivalent if using a global collection). After the initial load, the app sets up listeners. - Real-time updates: For lists of items like chores and lessons, the app uses `StreamBuilder` or `StreamProvider` to listen to Firestore query snapshots. For example, the ChoreBoardScreen might listen to `...children/{childId}/chores` collection changes. This way, any update (like chore status change or new chore added from another device) pushes instantly to the UI. - Writing data: When the user interacts (e.g. drags a chore to "Done", or checks off a lesson), the app will update the corresponding Firestore document (e.g. set `status = "Done"` on that chore's doc, or set a `completed = true` on a lesson). We use batched writes or transactions when necessary (for example, if completing a chore should also increment a points counter atomically). The Firestore client handles network syncing and offline caching by default, so the app remains responsive even with spotty connectivity.

Google APIs integration: Operation Summer heavily integrates with Google services (Calendar and Sheets). We leverage Google's APIs to provide seamless sync. On the Flutter side, we handle Google authorization and API calls as follows:

- **Google Sign-In & OAuth Scopes:** The app will use the **google_sign_in** Flutter plugin (or Firebase Auth's Google provider) to initiate OAuth for the Google account. We request the necessary scopes during sign-in – e.g. Calendar scope (`https://www.googleapis.com/auth/calendar`) and Sheets scope (`https://www.googleapis.com/auth/spreadsheets`) in addition to basic profile. Upon sign-in, we retrieve an `GoogleSignInAuthentication` object which includes an OAuth access token (and refresh token if configured for offline access). This token is used to authorize API calls to Google services.

- **Google API calls from Flutter:** Using the authenticated credentials, the app can call Google APIs directly. We have two primary ways:

- Use the HTTP REST endpoints of Google APIs with the `http` package and attach the `Authorization: Bearer <token>` header.
- Use the official Google APIs Dart package (`googleapis`), which provides client libraries for many Google services [3]. For example, we can use `calendar_v3` API from that package to insert or list events, and `sheets_v4` for Google Sheets. This package simplifies calling methods and handling responses, after we obtain an authenticated HTTP client with the user's credentials [4]. We will likely utilize this approach for structured access to Calendar and Sheets data.

Using the Google APIs from Flutter is appropriate since these involve user data and the user grants permission [5]. (We avoid embedding any Google service account credentials in the app for security [6] – all calls are on behalf of the signed-in user or done via Cloud Functions on secure server side as needed.)

- **Google Calendar Integration:** In the Calendar View, after user logs in with Google, the app will fetch events from the user's Google Calendar. Typically, we will decide on a specific calendar to use:
- **Option A:** Use the user's primary calendar. The app pulls events from primary and can create events there.

- **Option B:** Use a dedicated calendar (perhaps a separate Google Calendar named “Operation Summer” or one per child). This can be configured by the parent – e.g. the parent creates calendars for each child and the app stores the calendar ID for each profile in Firestore. Then events (like lessons schedule or chore reminders) go into that specific calendar. This helps separate kid-related events from others.

To display events, the app calls `calendarApi.events.list(calendarId, timeMin, timeMax, ...)` to get events within the visible date range. These events are converted to our internal `Event` model and fed to the calendar widget (e.g. as a list or via a Calendar DataSource). When the user adds a new event via the UI, we call `calendarApi.events.insert(...)` with the event details to Google. When an event is dragged to a new time or edited, we call `calendarApi.events.patch/update` to modify it. All these operations use the OAuth token of the user (so changes reflect on their Google Calendar immediately). We ensure to handle error cases (e.g. token expiry – refresh it via GoogleSignIn if needed).

- **Google Sheets Integration:** The app may also interact with Google Sheets if needed (for example, an “Import chores from Sheet” feature). However, direct Sheets editing is more likely done via backend automation (discussed later). On the client, we might implement a simple viewer or the ability to trigger sync:
- We could include a button in a settings or admin menu like “Sync with Google Sheets” which when tapped triggers either a Cloud Function or uses the Google Sheets API to pull data.
- If we do any Sheets API calls from Flutter, we will similarly use the OAuth token to access the Sheets API. For instance, reading a Google Sheet (spreadsheet) might be done to display it or to import tasks. The user would need to provide or select the spreadsheet ID (unless it’s fixed).

- Because the typical workflow is administrative (parents prefer using a spreadsheet to manage chores offline), we expect most Sheets sync to be handled automatically on the backend, rather than interactive in the app. Therefore, the Flutter front-end’s role is mainly to ensure the Google account is linked (to authorize backend or to use tokens) and possibly to notify the user when syncs happen.

- **PDF Preview, Download, and “Print All”:** The Learning Hub involves PDF lesson materials which the app must handle. For PDF **preview**, we use a Flutter PDF viewer component. There are multiple options:

- Flutter plugins like `flutter_pdfview` or Syncfusion’s PDF Viewer can display PDF files within the app.
- Another approach is to use the `printing` plugin in preview mode, which can render PDFs on a Canvas for viewing [7].

We will let the user tap on a lesson’s PDF link, which will either fetch the PDF from Firebase Storage or a public URL. Once downloaded to memory or file, the PDF is displayed in a viewer widget. The user can scroll, zoom, etc., to read the material.

For **downloading** a PDF, we will provide a button that saves the file to the device (e.g. to the Downloads folder). On mobile, this might use the `path_provider` to get a directory and then write the file. We might also leverage the share functionality (to allow the parent to share the PDF via email or messaging).

The **“Print All”** feature on the front-end is a bulk action that allows the user (likely the parent) to print all lesson documents for the day (or all currently visible PDFs) in one go. When the user taps “Print All,” the app will trigger a Cloud Function call (using Firebase Functions callable or an HTTPS request) to initiate the print job. The UI will likely show a loading indicator or confirmation ("Sending documents to printer…"). The actual printing workflow is handled by the backend (discussed in the Print Pipeline section), so the app doesn't directly connect to printers (unless it's running on a desktop with local printing capability).

After triggering print, the app could listen for a response or update (for example, the Cloud Function might immediately return "Print job queued" and later update a Firestore field indicating success or failure). We can reflect that in the UI with a snackbar or notification ("Print successful" or "Print failed: printer offline"). This asynchronous design ensures the UI isn't frozen during printing.

Note: Flutter does have the capability to print to local printers via plugins (on iOS/Android using AirPrint/ Google Print frameworks, and on desktop via CUPS through system dialogs) [8]. However, since our goal is an automated print pipeline for a specific printer, we choose to offload printing to the backend. This avoids platform-specific issues and allows scheduled/background prints even when the app is not open.

Backend (Firebase + Cloud Functions)

The backend for Operation Summer is powered by Firebase services, primarily Cloud Firestore for data storage, Firebase Authentication for user and profile management, and Cloud Functions for server-side logic (including PDF generation, printing, and third-party API sync). Additionally, Firebase Cloud Storage is used for storing PDFs and other media files. Integration with Google Sheets and Calendar is handled via Google's APIs, either directly in Cloud Functions or via authorized calls. Below we outline the **Firestore data model**, the authentication and role strategy, and the key **Cloud Functions** that implement the backend logic (PDF printing, Google Sheets sync, etc.).

Firestore Data Model

Operation Summer's data is structured in Firestore to support multiple children profiles under a parent, daily chore and lesson tracking, and cross-integration with external systems. We adopt a **hierarchical data model** where each parent user has a collection of children profiles, and each child has subcollections for their tasks and lessons. This structure simplifies security rules (a parent can only access their own children's data) and queries (fetch all chores for one child easily). Below is the proposed schema:

Collection / Subcollection	**Document ID**	**Key Fields** (Type)	**Description**
users	`{userId}` (parent's UID)	**name** (string), **email** (string), **role** (string)	Parent user accounts. Contains parent profile info and possibly a role ("parent"). Could also store an array of child profile IDs or basic settings (e.g. default printer or linked Google account info).

Collection / Subcollection	**Document ID**	**Key Fields** (Type)	**Description**
users/ {userId}/ children	`{childId}` (auto ID or name)	**name** (string), **age** (number), **points** (number), **calendarId** (string, optional), **profilePic** (url, optional)	Child profiles for a given parent. Each child doc represents one child. It stores the child's name, age, and the current accumulated points (for chores). We can also store a reference to an external calendar ID if a unique Google Calendar is used for this child (optional).
users/ {userId}/ children/ {childId}/ chores	`{choreId}` (auto ID)	**title** (string), **description** (string), **status** (string), **points** (number), **dueDate** (timestamp, optional), **completedAt** (timestamp, optional), **sheetRowId** (string, optional)	Chore tasks for the child. Fields: a title/ short name ("Clean Room"), a longer description if needed, a status field (e.g. "todo", "in_progress", "done"), point value for completion, and an optional due date/time (if this chore is scheduled or should appear on calendar). When a chore is marked done, `status` becomes "done" and `completedAt` is set. **sheetRowId** can store an identifier to sync with a Google Sheet row (if using two-way sync).
users/ {userId}/ children/ {childId}/ lessons	`{lessonId}` (auto ID or date-based ID)	**title** (string), **description** (string), **date** (date), **resourceURL** (string, optional), **completed** (bool), **attachmentPath** (string, optional)	Daily lessons or educational activities for the child. Each document might correspond to one lesson or one day's set of lessons. For flexibility, we treat each lesson as a document. Fields: a title (e.g. "Math Worksheet 1"), description/instructions, the date it's assigned (used to group by day), and an optional URL or storage path to the resource (PDF, video link, etc.). `completed` marks if the child finished the lesson. If the lesson has an attached PDF in Firebase Storage, **attachmentPath** stores the storage path or download URL.

Collection / Subcollection	**Document ID**	**Key Fields** (Type)	**Description**
events (optional, or could be under each child)	`{eventId}`	**title** (string), **startTime** (timestamp), **endTime** (timestamp), **childId** (string), **googleEventId** (string, optional), **type** (string)	(Optional) A collection for calendar events if we choose to store them in Firestore. This could be used to cache Google Calendar events or to store events that are internal (like chores deadlines). Each event could reference a child (or be family-general). If linked to Google, store the Google Calendar event ID for sync. `type` might distinguish "chore" vs "activity" vs "lesson". In many cases, we might not need this, relying on Google Calendar and the chores/lessons collections directly.
printJobs (optional)	`{jobId}` (auto ID)	**files** (array of strings), **timestamp** (timestamp), **status** (string), **printer** (string)	Used for the print pipeline. Each doc represents a request to print some files. **files** is a list of file URLs or storage paths (e.g. PDFs to print). **status** can be "queued", "printing", "completed", "error". **printer** might specify which printer (if multiple) or printer settings. Cloud Functions or a local service will monitor this collection to process jobs.

Data relationships and usage: The above model treats the parent user as the owner of children profiles. A parent can have many children; each child has their own sets of chores and lessons. Typically, queries will be scoped to one child at a time (e.g. display one child's chores). If needed, the parent user document can hold an array of childIds or the children subcollection can be listed to show all children. This model also makes it simple to enforce security: only the parent (and possibly the child, if they have an account) can read/write the documents in `users/{uid}/children/...`.

For quick lookup of all chores or lessons by date or status (across children), composite indexes or alternative structures might be needed (notably if a parent wants a combined view or if we implement global features like "All tasks due today for all kids"). However, given the scope, focusing on per-child subcollections keeps it straightforward.

If we consider children having their **own accounts** (Firebase Auth users), an alternative data model would have a more flattened structure: - A top-level **children** collection where each child doc has a field pointing to the parent's user ID. - Chores and lessons could then be subcollections under each child (now at top level), or in separate top-level collections with a childId field. - In that scenario, each child's Firebase Auth `uid` could serve as the document ID in the children collection. The parent's user doc could list authorized childIds, and security rules would allow the parent to access those child docs. The child accounts would have a `role: "child"` and a `parentId` field so that security rules allow them to access only their own data (and maybe read-only access to some parent-shared info). - For our project, we proceed with the simpler parent-centric model first (no separate login for child), but design it such that adding child logins later would be possible with minor adjustments (essentially by moving child docs to a global scope and setting up roles in rules).

Security and Permissions: Firestore Security Rules will enforce role-based access to the data. For example, rules can ensure that only the authenticated parent user can read/write their child subcollection documents. If separate child auth is used, rules will permit the child to read their own documents (matching `request.auth.uid == childId` for their doc, or checking a map of roles). This approach aligns with Firestore's role-based access control recommendations [9] (defining permissions per user or group). Each chore and lesson inherently belongs to a child which belongs to a parent, so the rules will reflect that hierarchy.

Firebase Authentication and Role Handling

Firebase Auth is configured to allow multiple profiles under one account. The **parent account** is the primary authenticated user who can manage all data. Children can either use the app under the parent's login (with the parent toggling "child mode" in the UI to let them mark chores complete, etc.), or have separate login credentials in a controlled way: - **Single Account, Multiple Profiles:** In this approach, only the parent logs in via Auth. The app then manages an internal list of children (from Firestore) and the UI can switch context to a particular child. If the parent hands the device to a child, they might activate a "Child View" (possibly protected by a simple PIN) that limits what the child can do (for instance, the child might only see their chores and lessons, and cannot navigate to admin settings). This is simple because we deal with one Auth user. - **Multiple Auth Accounts (Family Accounts):** Alternatively, each child could have a Firebase Auth account (perhaps created by the parent). In this case, we define roles: e.g. `role=parent` for the parent's user, and `role=child` for child accounts. We link child accounts to the parent (for example, store `parentId` in the child's Firestore profile or in a custom claim). A Cloud Function or secure admin action could be used to set up child accounts. For example, the parent enters the child's email and a temporary password or gets a link to invite the child. Once the child account is created and marked with `parentId`, that child can log in on their own device. The app, upon login, sees the role is "child" and will restrict access (the child app instance would query only their own chores and lessons, and not be allowed to modify others or add new ones). The parent account, conversely, can query all children profiles belonging to them and has full control (the app UI for the parent would list all their children and let them manage chores).

For simplicity and initial development, **single account with multiple profiles** is a fine approach, but we keep the door open for a multi-account system. The Firestore model described supports either: if child accounts exist, their user ID could match a child document ID so that data access is straightforward.

Firebase Auth will be set up with Email/Password (for a classic login flow) and with Google Sign-In (because we need Google API access). During authentication, we may use Firebase's multi-provider linking: the parent could sign in with Google (which verifies their Google account and gives us tokens for Calendar/Sheets access) and that also creates a Firebase user under the hood. If the parent prefers not to use Google sign-in for auth, we can allow email sign-up and later prompt them to "Connect Google Account" for calendar integration (using Firebase Auth linking or storing OAuth tokens).

Role enforcement in Security Rules: We will implement rules such that: - A parent user can read/write their own document in `users/{uid}` and anything under it (children, chores, lessons). - No user can read another parent's user doc or children. - If child accounts are used, a child user can read their own profile doc under someone's children subcollection or global children collection (matching their uid), and read/write their chores/lessons. They should *not* be able to create new child profiles or view sibling's data. The parent's `uid` on their profile or a roles map will be checked to ensure only the rightful parent can edit the child's data (for example, allow write if `request.auth.uid == parentId` or `request.auth.uid == childId` and the write is limited to certain fields like marking complete). - We may set custom claims in Auth for "parent" vs "child" to easily differentiate, but it's not strictly needed if Firestore rules derive permissions from data.

Cloud Functions: Backend Logic and Integrations

We use **Cloud Functions for Firebase** (written in TypeScript/Node.js) to implement server-side functionalities that keep the app data in sync with external services and handle automated tasks. Below are the key Cloud Functions and backend services in Operation Summer:

1. **PDF Generation & Aggregation (for print):** We create a Cloud Function (HTTPS callable or triggered) named e.g. `generateDailyPDF`. This function can compile information or gather multiple files into a printable format. Two use cases:
2. **Daily Summary PDF (Optional):** The function could generate a PDF that contains a summary of today's schedule, chores checklist, or a reward chart. Using a library like pdfkit or Puppeteer (to render HTML to PDF), the function could pull data from Firestore (e.g. list of chores and lessons for the day) and produce a nicely formatted PDF. This PDF can be saved to Cloud Storage and/or sent directly to print.
3. **Aggregate PDFs:** If multiple PDFs (lesson materials) need printing, the function might create a combined PDF (concatenate pages) or simply orchestrate printing them one after another. If combining, it could download each PDF from storage, merge them, and save a combined file.

This function is typically invoked when the user hits "Print All" in the app. The app could call a Callable Function `printAllMaterials(childId, date)` which internally calls `generateDailyPDF` or prepares the list of files for printing, then creates a `printJobs` entry (see Print Pipeline below).

1. **Print Dispatcher Function:** A Cloud Function (`enqueuePrintJob`) will handle the request from the app to print. Implementation steps:
2. It receives a list of files to print (either URLs or references to known files, such as all lesson PDFs for a given day/child).
3. It writes a new document in the `printJobs` collection with those file references, status "queued", timestamp now.
4. It can then either respond to the app ("Print job queued") immediately, or wait for confirmation of completion. However, since actual printing is done outside Firebase (on a local printer), this function's role is primarily to enqueue the job.

5. (In an alternative design, this function could directly attempt to contact the printer or a local server via HTTP, but firewall and network issues make that unreliable. Queuing in Firestore is a safer decoupled approach.)

6. **Google Sheets Sync – Firestore to Sheets:** We set up a Cloud Function that triggers on Firestore changes in the chores collection (and possibly lessons if needed). For example, an **onWrite trigger** on `.../chores/{choreId}` can respond whenever a chore is created, updated, or deleted. This function will use the Google Sheets API (via the Node.js `googleapis` library) to reflect those changes in a Google Spreadsheet. The spreadsheet ID can be stored in a configuration or in the parent's user document. Function logic:

7. If a new chore is created in Firestore, the function will append a new row to the Google Sheet containing the chore details (child name, chore title, due date, points, etc.). The function can also retrieve the row index and update the Firestore doc with `sheetRowId` (or the row number) so we know where it lives in the sheet.
8. If a chore is updated (e.g. marked done or edited), the function finds the corresponding row (via the stored row number or by searching an ID column) and updates the relevant cells (for status, completion date, etc.) on the sheet.
9. If a chore is deleted, the function could either remove the row or mark it as deleted in the sheet (depending on desired behavior; often better to mark as completed instead of outright deleting historical records).

Using Cloud Functions in this way keeps Firestore as the "single source of truth" while Sheets is a secondary interface [10]. The function ensures Firestore changes propagate to Sheets [11] [12]. For example, if the parent marks a chore as done in the app, shortly after, the Google Sheet will show that chore as done (maybe with a strikethrough or a "Done" status cell) automatically.

1. **Google Sheets Sync – Sheets to Firestore:** To handle changes made directly in Google Sheets (perhaps by a parent or another caregiver who prefers the spreadsheet interface), we implement a syncing mechanism in the other direction [13]. Cloud Functions cannot directly receive push events from Google Sheets, so we have a couple options:
2. Use a **Google Apps Script** on the spreadsheet: The spreadsheet can have an `onEdit` trigger that calls an external URL (a webhook). We can deploy an HTTPS Cloud Function (`receiveSheetUpdate`) that acts as a webhook endpoint. When the sheet's script detects a change in a specific range or row, it sends the updated data (perhaps the row contents and an identifier) to our cloud function via an HTTP POST [14] [15]. The cloud function verifies a token (to ensure the request is from an authorized source) and then updates the corresponding Firestore document (finding it by an ID or row number).
3. Alternatively, set up a scheduled Cloud Function to periodically pull data from the sheet and reconcile differences. For instance, every night or every hour, fetch all rows from the Google Sheet and for each row compare with Firestore (or simply overwrite Firestore with sheet data if sheet is considered the source for certain fields). This is simpler but near real-time sync is preferable if multiple interfaces are actively used.

By implementing these, any changes in Google Sheets (like adding a new chore row or editing a chore's status) would reflect in Firestore in near real-time [13]. This two-way sync ensures consistency across the app and the sheet, letting users "have their cake and eat it too" – use whichever interface is convenient [10].

1. **Google Calendar Sync:** Integration with Google Calendar can also be enhanced via Cloud Functions:
2. While the app itself will handle most Calendar operations through direct API calls when the user is active (for example, adding or updating events as the user drags them), we may want server-side logic for certain cases. One such case is if we want to automatically create calendar events for certain

entries. For example, when a new chore with a due date/time is added in Firestore, we might want to insert a corresponding event into the Google Calendar so that it shows up outside the app. We could write a Cloud Function trigger on `chore` creation that checks if `dueDate` is set and, if so, uses a stored Google OAuth credential or service account to create a calendar event for that chore.
3. However, using the user's OAuth token server-side is tricky (we'd have to store a refresh token in Firestore or use Firebase Auth's ability to mint a custom token). A cleaner approach is to have the client create the event via its own Google session. So this function is optional.
4. Another potential function is to subscribe to Google Calendar push notifications. Google Calendar API can send push updates to a webhook when events change (if we set up a channel). We could register a Cloud Function as that webhook endpoint. This would allow, for instance, catching if the user or someone else edits the calendar externally (maybe via Google's calendar app) and updating Firestore or the app's local cache accordingly. This is advanced and requires renewing subscriptions periodically. For now, we assume the app will fetch latest events when opened or refreshed by the user, so we might not implement this push listener initially.

In summary, most Calendar sync can remain on the client, but Cloud Functions can be used for **automation** (like ensuring chores/lessons with dates become calendar events, or sending reminders via FCM or email when an event is near – though the latter is beyond current scope).

1. **Point Tracking and Rewards:** While points accumulation could be handled purely client-side, we might include a Cloud Function to maintain consistency and security:
2. For example, an **onUpdate trigger** on a chore document could check if the `status` changed to "done". If so, it could increment the corresponding child's `points` field in Firestore (under the child profile) by the chore's point value. This server-side increment ensures that even if a malicious client tried to mark a chore done multiple times or manipulate points, the function can control the logic (e.g. only add points if it was not done before, etc.). Firestore supports atomic increments, but implementing as a function allows adding any additional logic (like capping points or triggering a reward).

3. This function might also handle things like issuing an achievement or logging the completion in a separate "history" collection if needed.

4. **Scheduled Functions:** We will utilize **Firebase Scheduled Functions (Cloud Scheduler)** for periodic tasks:

5. **Daily Print Job:** Each morning, a scheduled function (`autoPrintMorningMaterials`) can run (at e.g. 7:00 AM local time) to automatically prepare and print the day's materials. It would iterate through all active children (or all families) in Firestore, gather that day's lessons' PDFs and perhaps a chore checklist, and enqueue a print job for each. This ties into the same Print Pipeline (creating `printJobs` entries). The idea is that the parent wakes up to find the day's worksheets already printed, without even opening the app.
6. **Data Backup or Sync:** We could schedule a function to backup Firestore data to a Google Sheet or Drive CSV at intervals (if not already covered by the two-way sync). Or a function to sync any unsynced changes (retrying failed syncs).

7. **Notification Function (optional):** Not requested, but as an idea: a function could send a push notification or email summary to the parent each evening with what was completed or what's

planned for tomorrow. This again would query chores/lessons and use Firebase Cloud Messaging or SendGrid for email.

8. **Miscellaneous Functions:**

9. **Create Child Profile Callable:** If we allow the parent to create separate child accounts, a callable function could help by taking an email and creating a Firebase Auth account for the child (with a temporary password or link), then writing the Firestore profile and setting the appropriate `parentId` and custom claim. This ensures only the parent (authenticated) can create accounts tied to them.
10. **Cleanup Functions:** e.g. if a child profile is deleted, clean up all subcollections; or if a print job is stuck, a function to clear it after a timeout.
11. **Storage Trigger (if needed):** If parents upload images or PDFs (for custom lessons), a storage trigger could, for instance, create a thumbnail or sanitize the file. This might be overkill for our uses, but worth noting if images are used for chores (like chore photo proof).

Integration with External Systems (Google APIs) – Backend

For Google Sheets and certain Google Calendar operations, the backend functions will use service credentials or stored tokens: - We will set up a **Google Cloud service account** that has access to the Google Sheets that need syncing. The easiest way is to share the Google Sheet with the service account's email. The Cloud Function (running with that service account's key) can then call the Sheets API without user intervention. We will store the service account credentials securely (e.g. in Firebase Secrets or Cloud Secret Manager) and initialize the Google API client with those creds. This avoids requiring the user's OAuth for background sync. However, this means the specific spreadsheet must be known and shared — which is fine if we have a single sheet per family and we can automate that setup (the user might input the sheet ID in the app, and we then use an admin action to share it with our service account). - For Google Calendar, we generally rely on user OAuth tokens. We do not want to store the user's refresh token in Firestore unless absolutely needed (security risk). If we did need server-side calendar access when the user isn't actively using the app, we could store a long-lived refresh token (obtained by requesting offline access in GoogleSignIn) in Firestore (encrypted), and Cloud Functions could use it to get a fresh access token to make calendar changes. This requires careful handling. In most cases, it might be unnecessary if the calendar changes happen either interactively or not at all when user is away. One exception is the idea of scheduling events for chores automatically – that could be done at chore creation time when the user is presumably using the app, so the app itself can call the calendar API. - The Cloud Functions environment can use Node.js Google API libraries. For Sheets, `google.sheets('v4')` allows reading/writing cells easily [16] [17]. For example, the Firestore-to-Sheets function will use `sheets.spreadsheets.values.update(...)` as shown in the blueprint [17], specifying the range and values to update. - All secrets (API keys, OAuth client IDs, etc.) will be stored in environment config or secret manager, not in code.

By using Cloud Functions to mediate interactions with Google Sheets and possibly Calendar, we ensure that heavy operations or cross-system sync logic resides in a controlled environment, and the client app stays responsive.

Firestore Rules and Indexes

We will implement Firestore indexes to support queries we need. For example: - An index on `children/{childId}/chores` by `status` might be useful if we query tasks by status (though we may not, since we typically load all and then filter in UI). - If we list chores by `dueDate` or lessons by `date`, composite indexes on those fields might be needed if combined with parent/child filters or sorting. - If a global `events` collection is used to filter by childId and date range, that definitely needs composite indexes (childId + date). - We'll identify these during development and add via Firebase console or `firestore.indexes.json`.

The security rules will correspond to the data model described. For instance:

```
rules_version = '2';
service cloud.firestore {
  match /databases/{database}/documents {
    match /users/{userId} {
      allow read, write: if request.auth.uid != null && request.auth.uid ==
userId;
      match /children/{childId} {
        allow read, write: if request.auth.uid != null && request.auth.uid ==
userId;
        match /chores/{choreId} {
          allow read, write: if request.auth.uid != null && request.auth.uid ==
userId;
        }
        match /lessons/{lessonId} {
          allow read, write: if request.auth.uid != null && request.auth.uid ==
userId;
        }
      }
    }
  }
}
```

This simplistic rule grants the parent full access to their data. If child accounts are introduced, rules will be expanded: e.g. allow child to read/write their own profile and chores where `childId == request.auth.uid` or where the parentId matches. We'll also ensure points cannot be arbitrarily altered by a child (maybe only server can update points field, or parent).

Calendar View – Technical Details

The Calendar View is a core component that bridges the app's data with the user's Google Calendar. It provides a visual scheduler and interactive planner for Operation Summer activities. This section details how events are represented, how Google Calendar is embedded and synchronized, and how the drag-and-drop rescheduling is implemented under the hood.

Calendar UI Implementation

We will use a Flutter calendar widget to render the calendar UI. Two main options are considered: - **Syncfusion Flutter Calendar (`SfCalendar`)** – a powerful calendar component that supports multiple views (month, week, day) and built-in **drag-and-drop** for appointments [2]. Using SfCalendar would expedite development as it natively allows dragging events to reschedule and even resizing time slots. We simply set `allowDragAndDrop: true` and handle callbacks like `onDragEnd` to detect the new time slot of the event. - **TableCalendar or Custom Implementation** – a more lightweight approach that may involve drawing the calendar grid and managing events manually. Drag-and-drop in this case would require using Draggables for event widgets and calculating drop targets for date cells, which is more work.

Given the complexity of scheduling, we lean towards using **Syncfusion's SfCalendar** (or a similar library) to leverage built-in functionality. For example, in week view, a user can press and hold an event and move it to a new time slot; the widget will provide a callback with the new time details once dropped. *Note:* The drag-and-drop is supported in week/day view on mobile; month view might not support it on small devices due to space [18] – in month view we might instead allow dragging to another day if possible or fallback to an edit dialog.

The calendar will likely default to a **month view** for an overview of the summer. We will provide UI controls to switch to a week or day agenda view for fine-grained control. Each event will be rendered with a distinct color or icon: - We can use one color for general events (from Google Calendar) and another color for chores or lessons if they are shown on the calendar. - If multiple children's events are shown together, we could color-code by child (though we might also just filter one child at a time to avoid clutter).

Event Data Schema and Management

Event Representation: An `Event` in the app will have properties like: - `id` (internal ID, which might be the Google Calendar event ID for events synced with Google), - `title`, - `startTime`, `endTime`, - `location` (if any), - `description` or `notes`, - `source` (e.g. "Google" or "Chore" or "Lesson" to know where it came from), - `childId` (if applicable, to know which child this event is for, e.g. if calendars combined).

If using Syncfusion, we will wrap our events in their `CalendarDataSource` or convert to `Appointment` objects. Syncfusion allows custom appointment data, so we could subclass `CalendarDataSource` to work with our Event model.

Linking to Firestore: There are a few scenarios: - **Regular Calendar Events:** These are events that exist in Google Calendar (like "Doctor appointment" or "Vacation"). They will *not* be stored in Firestore (no need to duplicate data). They'll be fetched via API when needed and exist only in memory on the client. - **Chore/ Lesson Events:** If a chore has a due date/time, or a lesson is scheduled at a specific hour, we might want them to appear on the calendar. We have choices: - Create corresponding Google Calendar events for them. This way they appear uniformly in the Google Calendar and in the app. We'd need to ensure they're labeled clearly (e.g. event title could be "Chore: [task name]" or put them on a special calendar). - Or, simply render them on the calendar UI directly from Firestore data without involving Google (just overlay them in the calendar widget). This requires merging two data sources: Google events and local events.

Merging data sources can be done by fetching Google events and Firestore events for the given date range and combining into one list for the calendar widget. We will likely implement it this way, as it avoids creating clutter on the user's actual Google Calendar unless desired. If the user does want everything on Google Calendar, we can treat that as an optional sync feature (e.g. a setting "Show chores on my Google Calendar" that if enabled, creates events for chores via an API call).

Sync with Firebase: We will maintain minimal persistent event data in Firestore: - If we do not create Google events for chores/lessons, then Firestore remains the sole store for those (with fields like dueDate in chores as described). The app will read those and display them on the calendar. Rescheduling such an event (e.g. moving a lesson from morning to afternoon) would mean updating the `dueDate` field in Firestore for that chore or lesson. - If we do create Google events for them, then the Google Calendar becomes the primary data source. We might still store the mapping from chore to Google event (via an event ID field in Firestore) so we can update/delete it later if needed. - The Calendar View might cache some events in Firestore (for offline use). For example, we could store the next week's worth of events in an `events` subcollection for offline viewing. However, implementing offline cache for Google Calendar events might be overkill if internet is usually available. A simpler approach is to just refetch on demand or let the user manually refresh.

Drag-and-Drop Rescheduling Implementation: When a user drags an event in the Calendar UI: - If using SfCalendar, the control will handle the visual drag and call our `onDragEnd` callback with details. In that callback, we will determine what event was moved and to what new time. - We then perform the update: - If it's a Google-synced event (i.e. a normal calendar event or a chore that had a Google event), we call the Google Calendar API to update the event's `start` and `end` time to the new timeslot [2]. This is done using the credentials of the user. On success, we also update our local state so the UI reflects the change (if the calendar widget hasn't already internally moved it). - If it's a Firestore-only event (like a lesson that wasn't on Google), we update the Firestore document's time fields. Firestore listener will then propagate that change to the UI (possibly causing the calendar to rebuild its events list). We might also toast a confirmation "Rescheduled lesson to 2:00 PM". - We should handle the case where the API call fails (e.g. network issues). In that case, we might revert the event to its original position or show an error. Using optimistic UI (moving it immediately) with rollback on failure is user-friendly. - For drag-and-drop to different days (month view scenario): If unsupported directly on month cells on mobile, we might implement an alternative: e.g. a long-press on an event could pop up a small menu to "Move to another day" and then the user picks a date from a date picker. But ideally, we want the intuitive drag action as much as possible.

Creating and Deleting Events: Apart from rescheduling, the calendar allows adding new events and possibly deleting events: - Adding a new event via app: We open a form, then call Google API to create it (and add to Firestore if needed). After creation, we fetch the updated events or directly insert it into the calendar state. - Deleting an event: Could be triggered by a long-press or a delete button on event details. We'd call `calendarApi.events.delete`. If the event was associated with a chore/lesson, we might also want to update that (maybe mark the chore as done or remove the schedule). We'll likely not allow deleting chores via the calendar (that should be done in chore board), so deletion in calendar is mainly for non-chore events.

Multiple Calendar Management: If each child has their own Google Calendar (distinct calendarId): - The app will need to fetch from multiple calendars. For example, if viewing a combined family calendar, call events.list on each relevant calendar and merge results. Or if only one child view is selected, just fetch that child's calendar. - We will store each child's calendar ID in their profile (as mentioned). The parent might set this up in a settings screen by selecting or inputting a calendar ID (we could use Google's calendar list API to let the parent pick one of their calendars to associate with the child). - This separation keeps each child's schedule isolated if needed. In the UI, a parent could toggle which children's events to show (like checkboxes for each child). - However, initially we might assume a single calendar (the parent's primary or a main Operation Summer calendar) to keep it simple, which contains events for everything.

Syncing Calendar Data with Firebase

While Google Calendar is the primary store for event data, we ensure that relevant info is also reflected in Firestore where it intersects with our domain: - **Chore/Lesson scheduling:** If a chore is given a due time and we choose *not* to push it to Google Calendar, then Firestore is the only place it lives. The calendar view will query Firestore for chores where `dueDate` is on the current day, etc. We can create a composite index on `dueDate` to support queries by date. Alternatively, since chores are subcollection data, we might simply fetch all chores for the week and filter client-side. - **Google event caching:** Optionally, when the app fetches Google events, it could write them into Firestore (under `events` collection or into each child profile) for caching. For example, after fetching events for the upcoming week, we store them under `/users/{uid}/events/` or similar. This would allow the rest of the app (or a widget like a "Upcoming events" list on the dashboard) to use this data without additional API calls. We would need to periodically clean or update these cached events to avoid stale data. Implementing this cache is not strictly required, but it's an enhancement for offline support. Otherwise, if offline, the app might show the last fetched events kept in memory.

- **Two-way consistency:** If we have cached events or events that originated from Firestore (like chore deadlines), we want to avoid conflicts:
- If a chore's dueDate is changed in Firestore (say via the Chore Board or some other UI), we should update the Google Calendar event if one exists.
- If an event is moved in Google Calendar (outside the app), the app will only catch it when fetching again. We might consider a Cloud Function with push notifications from Google or more simply, whenever the user opens the calendar screen, do a fresh fetch to override any stale info.
- Essentially, Firestore is the authority for chores/lessons, Google Calendar is authority for events; where they overlap, one needs to update the other. Our plan: treat Google Calendar as authoritative for scheduling *once an event is created there*. If a chore is linked to a Google event, then moving it on Google should dictate the schedule (we'll update Firestore dueDate on next sync). Conversely, if user moves it in our app (which will call Google API), it's already in sync by design.

User Experience and Performance

To ensure a smooth user experience: - The calendar screen will lazy-load events. We won't fetch an entire year of events at once. Likely, we load a window (say the current month plus one month ahead/back). If the user scrolls or moves to next month, we trigger another fetch for that range. - Drag-and-drop operations will be animated by the UI component, making the interaction feel native. After dropping, we'll maybe show a small checkmark or toast indicating "Rescheduled!" once the backend update succeeds. - We will implement loading indicators for the calendar (e.g. a pull-to-refresh or a spinner while events are loading). - Any conflicts or errors (like trying to drag onto a slot that's in the past or overlapping events) will be handled gracefully – possibly by preventing the drop or asking for confirmation if overlapping.

By embedding Google Calendar functionality directly and synchronizing it with our domain data, the Calendar View becomes a powerful tool for parents and kids to visualize and manage their summer activities in one place.

Chore Board (Kanban) – Technical Details

The Chore Board is where daily tasks are listed and their statuses managed. It employs a Kanban paradigm to make task management interactive for children. This section describes the data schema for chores, how the Flutter UI implements drag-and-drop for tasks, syncing chores with Google Sheets, and how task status and points are tracked.

Chore Data Schema in Firestore

As outlined in the data model, each chore is stored as a Firestore document (in a subcollection under the child's profile). Key fields include: - **title**: Short name of the chore (e.g. "Make Your Bed"). - **description**: Longer instructions if needed ("Make sure to straighten the sheets and pillow."). - **status**: Can be one of `"todo"`, `"in_progress"`, or `"done"` (or similar). Initially, tasks are "todo". If a child actively starts a chore, the app could move it to "in_progress" (optionally), and finally to "done". - **points**: Numeric value of reward points the chore is worth upon completion. - **dueDate**: (Optional) deadline or suggested completion time for the chore (if present, we might show this on calendar or sort chores by time). - **completedAt**: Timestamp when marked done (if status = done, this gets set, used for history or to avoid double-counting points). - **sheetRowId**: If syncing to a Google Sheet, this stores an identifier to locate the chore's row in the spreadsheet (e.g. the row number or a unique ID). - **childId**: If chores were in a central collection, we'd have this to identify the owner child. In our chosen structure, the childId is implicitly the parent document, so not needed within the document itself.

Chores are typically ephemeral daily tasks, but we will not automatically delete them on completion (in case we want records). Instead, we could filter the view to hide old tasks. Alternatively, we might move old tasks to an archive subcollection or mark them with a date completed and query only today's tasks by a date field. This detail can be refined with usage; initially, we might just show all tasks and let the parent manually clear them if needed.

Flutter UI and Drag-and-Drop Mechanics

Kanban Board UI: The Flutter UI for the chore board consists of columns representing task status: - We'll have three columns (for To-Do, In Progress, Done). These can be arranged in a `Row` with each column as a `Expanded` containing a `ListView` of chores filtered by that status. - Each chore in the list is displayed using a custom `ChoreCard` widget. This widget might look like a small card with the chore title, maybe a checkbox or icon, and possibly the point value. It could also have a visual indicator if it's draggable (though in a touch interface, users learn by trying). - We enable drag-and-drop: - Each `ChoreCard` (in To-Do or In-Progress columns) is wrapped in a `LongPressDraggable` widget. The `data` carried by the draggable is the chore's ID or object. - The drop targets are the columns themselves. For example, the "Done" column's list container is a `DragTarget` that accepts Chore data. We implement `onWillAccept` to highlight the column when a card is dragged over it, and `onAccept` to handle the drop logic. - When a chore card is dropped onto a new column, we update its status field in Firestore (which triggers UI updates through our listener). - During drag, Flutter will create a floating drag widget that follows the touch (we can style this, maybe showing the card semi-transparent). - If a card is dropped outside of any valid target, it snaps back to its original place (default Draggable behavior).

Alternatively, using a specialized library like `boardview` or `kanban_board` could reduce manual coding, as these provide built-in drag-and-drop of list items across columns [1]. Such a package likely manages state internally and calls callbacks when reordering or moving happens. We can evaluate it: for example, the `kanban_board` package would allow us to define columns and items and handle drag logic for us [1]. We would integrate it by feeding it our list of chores grouped by status and providing a callback when an item's group changes, wherein we then perform the Firestore update.

Adding/Editing Tasks: The UI includes an "Add chore" button. When tapped, it opens a form dialog where the parent can enter chore details (title, description, points, optional due time) and save. This will create a new Firestore document in the "todo" status. The UI will automatically show it under To-Do (since our listeners pick up the new doc). Editing a chore (e.g. to change its title or due date) could be done via tapping the card (maybe open a similar form pre-filled, if we want that functionality for parents).

Marking Complete without Drag: For convenience, especially on mobile, we may also allow marking a chore done by a simple tap or swipe gesture. For example, swiping a chore card to the right could mark it done (like a check-off). Or tapping a checkbox on the card could instantly set status to done (and move it to Done column). Drag-and-drop is fun and visual, but quick actions improve usability. We'll ensure consistency whichever method: any status change triggers the same Firestore update and point awarding logic.

Syncing Chores with Google Sheets

One unique aspect is keeping chore data in sync with a Google Sheet (for easier editing or record-keeping by parents). As described in the backend section, the integration works via Cloud Functions and possibly Apps Script: - When the parent adds or edits a chore in the app, the Cloud Function will update the Google Sheet accordingly (e.g. add a new row with the chore or change the status cell) [16] [19]. - If the parent (or someone) edits the Google Sheet (like change a chore's due date or mark something done from the sheet), the Apps Script triggers and our Cloud Function updates Firestore [14] [20]. - To facilitate this, we'll ensure the data mapping between Firestore and the Sheet is consistent. We can designate columns in the sheet such as: Child Name, Chore Title, Description, Due Date, Points, Status, and perhaps a hidden column for the Firestore ID. - For instance, say the Google Sheet "Chores" has: - Column A: Child Name - Column B: Chore Title - Column C: Description - Column D: Due Date - Column E: Points - Column F: Status - Column G: FirestoreID (hidden or not human-edited)

When our function adds a row, it will generate a Firestore ID, put all data in, and fill the FirestoreID column. This allows the Apps Script to send that ID back when a row is edited, so we know exactly which document to update in Firestore. This two-way key link is crucial for robust sync.

- If multiple sheets are needed (one per child or one for chores vs lessons), we can handle that accordingly. But one sheet for all chores could suffice if we include the child name column.
- Using Zapier or other third-party integrators was an option (they provide Firebase–Sheets integrations [21]), but we decided to implement it ourselves for full control.

Data Consistency Considerations: Because both Firestore and Sheets can be edited, conflict resolution is simple: last write wins. Our system updates one side immediately when the other changes, so theoretically they won't diverge by much. If simultaneous edits occur (rare in this scenario), Firestore (with offline and last write) and the nature of spreadsheets (last edit in cell wins) will each have their own resolution. The eventual consistency should suffice for a family environment.

Having Firestore as the "source of truth" means we trust the app's data primarily [10]. The sheet is more like a convenient UI and backup. If something goes out of sync, we could provide a manual "Import from Sheet" or "Export to Sheet" button to reconcile, but ideally our automated triggers prevent that need.

Task Status Updates and Points Tracking

Status Updates: When a chore's status changes (especially to "done"), a few things happen: - UI: The chore card moves to the Done column (or is removed from To-Do/In-Progress). We might animate this (e.g. fade out from old spot, slide into new column). - Data: Firestore is updated (`status = done, completedAt = now`). This triggers any backend logic like point awarding. - Google Sheet: The Cloud Function will update the status cell to "Done" for that chore (and maybe gray out the row via Apps Script formatting, if we set that up in the sheet). - Calendar: If this chore had an associated calendar event (like a reminder), we might mark it as done there or delete the event to avoid clutter. Or maybe we leave it; up to design. Could integrate with Google Tasks, but that's out of scope.

Points Tracking: Each chore contributes to a child's point total. The design is: - Each child document has a `points` field that aggregates all completed chore points. - When a chore goes from not-done to done, we add its points to the child's total. If a chore is undone (say a parent reopens it), we subtract the points. - We ensure idempotence: if a chore is already done and somehow a duplicate update triggers, we don't double-add points. Using a Cloud Function that checks `change.before` vs `change.after` can ensure we only add when status transitions to done. - Alternatively, we could calculate points on the fly (sum of all chores where status=done). But that could be heavy if chores accumulate. Still, for likely moderate number of chores, it's fine. Yet having a stored total is convenient and efficient. - We will implement a Cloud Function as described to handle the increment. It can use `admin.firestore().runTransaction` to increment points safely to avoid race conditions if two tasks finish at once. - The `points` field on the child profile will update in Firestore, and since the app is listening (likely via a stream on the child doc or a specifically on that field through profile stream), the UI (like the Dashboard or profile header) will update the displayed points in real-time.

Rewards and Usage of Points: Though not explicitly requested, it's worth documenting how points might be used: - The app could allow parents to set up rewards that cost certain points (e.g. 100 points = ice cream treat). If implemented, those would be additional collections and UI, but currently we just track points. - Points can also be used for positive reinforcement: e.g. a big number on the dashboard, maybe leveling up or a badge when reaching milestones.

Task Validation: We might consider requiring parent approval for chores completion (especially if using points and wanting to ensure kids just not cheating by marking everything done). A possible approach is to have a status "done (pending approval)" or simply trust the kid and allow parent to unmark if needed. This wasn't specified, so initially we assume trust or that the parent is overseeing the app usage.

Integration with External Tools (optional expansions)

- **Voice Assistants / IoT:** While not in spec, one could imagine connecting with Google Assistant ("How many chores do I have left?") or printing chore charts with a single command. Our architecture could accommodate such additions via Cloud Functions or APIs (for instance, a Google Assistant action could fetch Firestore data for chores).
- **Mobile notifications:** If desired, we could send a push notification when a new chore is assigned or when a chore is marked done (to inform the parent). This would use Firebase Cloud Messaging. Not in scope now, but our backend design with Cloud Functions could easily plug that in (trigger on chore creation or completion and send a message).

In summary, the Chore Board component provides a fun, interactive way to manage tasks, with robust backing from Firestore for real-time sync and Google Sheets for external editing. The use of Flutter's drag-and-drop (or a library) gives a Trello-like experience, which should be engaging for kids. All changes are persisted and mirrored to keep everyone (and every system) up-to-date.

Print Pipeline – PDF Generation and Automated Printing

One standout feature of Operation Summer is the ability to automatically print materials (like worksheets or chore charts) without manual intervention. This "Print Pipeline" involves generating PDFs (if not already available), queuing them, and sending them to a **local printer** via CUPS (Common Unix Printing System) or similar. This section details how the printing workflow is orchestrated using Firebase Cloud Functions and a local print service, as well as the print job identification and handling.

Overview of Printing Workflow

1. **Initiation:** A print job can be initiated either by user action ("Print All" button in the app) or by an automatic schedule (every morning).
2. **Cloud Function Queuing:** When triggered, the Cloud Function prepares the content to print. It identifies which PDF files (or other documents) need printing:
3. In the case of "Print All" for daily lessons, it will gather all PDF attachments for today's lessons for the selected child. These files likely reside in Firebase Storage (each lesson might have an `attachmentPath`). The function will get secure download URLs for each file.
4. If a dynamic PDF (like a summary page) needs generation, the function generates it on the fly (using a PDF library) and either attaches it to the response or stores it temporarily.
5. The function then creates a new document in the `printJobs` collection with details of the print request. For example: `files: [url1, url2, ...]`, `status: "queued"`, `printer: "HomePrinter"`, `timestamp: ...`. (Alternatively, we could skip Firestore and directly call a local endpoint, but using Firestore as a queue makes it easier to monitor and more robust if the local service is offline and comes online later.)

6. The function returns a success acknowledgment to the app, so the UI can show "Printing started" message.

7. **Local Print Service:** We set up a small **local service** on the network where the printer is connected. This could be a script running on a Raspberry Pi, a home server, or even a user's PC. The service's responsibilities:

8. Continuously monitor the `printJobs` collection in Firestore for new entries (this can be done by using the Firestore SDK in an admin context, with a listener on the collection where `status == 'queued'`).
9. When a new job appears, the service retrieves the job info: list of file URLs (or paths). For each URL, it downloads the file to a temporary local path.
10. The service then sends each document to the printer via CUPS. If on a Linux-based system (like a Raspberry Pi running CUPS), it could use the CUPS command-line (`lp` command) to print the file [22]. For example, `lp /tmp/file1.pdf` will send the PDF to the default printer (or a specified printer name with `-d printer_name`).
11. We can also use CUPS' IPP (Internet Printing Protocol) interface. CUPS can expose a printer on the network; our service could issue an HTTP request to CUPS with the PDF. But since the service is on the same machine as CUPS (likely), using the command-line or CUPS API is straightforward.

12. The service updates the Firestore job document's status to "printing" (when it starts) and "completed" when done. If an error occurs (printer not responding, or file error), it sets status "error" and maybe logs details (possibly in a subfield or separate log collection).

13. **Printer Setup (CUPS):** The local printer must be installed and configured on the device running the service. Using CUPS, we ensure the printer is accessible via a known name. CUPS can handle PDF printing natively (most modern printers and CUPS filters accept PDF). If any conversion is needed (like to PostScript), CUPS does it internally.

14. If the printing environment is Windows or Mac instead, the service could call OS-specific print commands (Windows: use `PrintDocument` or run a powershell command; Mac: also has `lp` via CUPS since macOS uses CUPS under the hood).

15. The solution is cross-platform in concept, but we assume a Linux environment with CUPS for concreteness, as it's commonly used for IoT printing setups.

16. **Alternative Direct Print (Not via Firestore):** For completeness, another design is to skip the Firestore `printJobs` and have the Cloud Function directly attempt to contact the printer. However, a Cloud Function runs on Google's servers with no direct access to the local network. Google Cloud Print (the old service to do such printing) was deprecated in 2021 [23], so we cannot rely on that. We must route through a device on the local network. Our approach basically turns that device into a Cloud Print proxy listening to Firestore events. This is essentially a custom implementation of a Google Cloud Print alternative using CUPS [23].

17. **Confirmation to User:** The app can get feedback about the print job status if needed:

18. The simplest way is to trust that once queued it's handled. But we can do better: the app can listen on the specific `printJobs/{jobId}` document (since we got the jobId after the function call returns) to see updates. For example, if status changes to "completed", we can notify the user "Print completed". If "error", we display an error message ("Please check the printer").
19. This requires the app to have read access to that printJobs doc. We can structure security such that only the parent user who initiated it can read it, or we can put printJobs under the user's node in Firestore (like `/users/{uid}/printJobs/{jobId}`) so it's naturally accessible to them. This

might be wise, to avoid any multi-user security concern (though in a single-family app, it's not a big issue; still good practice).

Auto-Print Setup and CUPS Configuration

To set up auto-printing with CUPS: - **CUPS Installation:** The device (e.g. Raspberry Pi) needs CUPS installed and the printer added. This typically involves installing CUPS and using its web interface or command-line to add the printer (via IP, USB, etc.) [24]. We ensure the printer can print PDFs driver-correctly by testing manually. - **Access:** We may open CUPS to accept jobs only from localhost for security. Our service running on the same device will be acting as a client to CUPS on localhost. - **Local Service Implementation:** This could be a small Node.js script using the Firebase Admin SDK, or a Python script using the Google Firebase library. Pseudocode:

```
const admin = require('firebase-admin');
admin.initializeApp({...serviceAccountCreds...});
const db = admin.firestore();
db.collection('users/uid/printJobs').where('status','==','queued')
  .onSnapshot(snapshot => {
     snapshot.docChanges().forEach(change => {
        if(change.type === 'added') {
           const job = change.doc.data();
           handlePrintJob(change.doc.id, job);
        }
     });
  });

async function handlePrintJob(jobId, job) {
  // update status to 'printing'
  await db.doc(`users/uid/printJobs/${jobId}`).update({ status: 'printing' });
  for(const fileUrl of job.files) {
    // download file
    let filePath = await downloadFile(fileUrl);
    // send to printer
    execSync(`lp -d ${job.printer || 'DefaultPrinter'} "${filePath}"`);
  }
  // update status to 'completed'
  await db.doc(`users/uid/printJobs/${jobId}`).update({ status: 'completed' });
}
```

This is a rough idea. Proper error handling should be included: if `execSync` fails or printer is offline, catch and update status to 'error'. We might also include retry logic or a mechanism to avoid infinite blocking. - **Security:** The service account used to initialize the Admin SDK should have access only to the necessary parts of Firestore (in practice, admin SDK bypasses rules, but running it on a trusted device is fine). We could further restrict by only listening to a specific user's jobs if multi-family was a scenario. Here, presumably one family, so not a problem. - We should keep the service running on boot of the device. On a Raspberry Pi, this could be a systemd service or a PM2 process that starts on boot, ensuring prints can happen anytime.

Print Job Identification and OS-Level Printing

Each print job includes the list of **printable files**. These files might be: - PDF worksheets (most common). - Could also be images (jpg/png). CUPS can print images directly as well (it converts them). - Possibly text or HTML (if we had any) – but we can convert those to PDF in the Cloud Function beforehand to maintain formatting.

If multiple files are listed, the local service will send them one by one. If order matters (likely print in listed order), we ensure to respect it.

"Print All" Implementation Details: In the app, when the user taps Print All: - The app calls a Callable Cloud Function `requestPrintAll(childId, date)`. - This function might: - Query Firestore for `lessons` of that child with `date == today` (and perhaps any chores flagged for printing). - Get each lesson's attachment path (if stored in Storage, get a signed URL or use Storage's getDownloadURL via the Admin SDK). - It creates the `printJobs` doc as described. - It could also directly return, for example, `jobId` to the app. - We might also want the function to bundle small documents together. For example, if there are many small PDFs, some printers handle a single combined PDF better than many separate jobs. But combining PDFs in Cloud Function might be heavy on memory/time if there are many pages, so sending separate is fine. CUPS will queue them quickly anyway. - If a "Print All" action should also include a summary page or cover page, the function can generate it and include as the first file.

Edge Cases & Reliability: - If the printer is off or out of paper, the print job will fail. The service can detect an error code from `lp` and mark accordingly. We might notify the user to check the printer. The user can then fix it and perhaps press a "retry print" button in the app which could simply set the status back to "queued" or make a new job. - If the local service is down (device off), the Firestore job will remain "queued". We should handle that scenario: - The Cloud Function could set a TTL on the printJob doc (maybe schedule a Cloud Function in e.g. 5 minutes to check if it's still queued and alert user). - But more simply, if the service comes online later, it will eventually pick it up. We might just inform user "Print job queued; please ensure the printer device is on." - We will also consider **printer selection**: If the family has multiple printers (maybe a color and a B&W, or upstairs/downstairs), we could have a printer field (we included one in printJobs). The app can allow the parent to set a default printer name (perhaps stored in the user doc). The local service can either ignore jobs not for it or ideally, the app only creates jobs for the printer that service handles. Assuming one printer, not an issue. If multiple, possibly one service could handle all if it has drivers configured for all.

Security of Local Service: The local print service is effectively an extension of our backend running at home. We must ensure it's not openly accessible. By using Firestore as the queue, we avoid needing to open any network ports on the home network. All communication is outbound (the service polls Firestore via secure channel). This is secure as long as the Firebase project is protected and the service account isn't compromised. We won't expose any direct endpoint on the internet for printing (which could be abused). So this design is secure and firewall-friendly.

Testing the Pipeline: We will test the pipeline in stages: - Ensure Cloud Function correctly generates or fetches PDFs and places a printJobs entry. - Simulate the local service (could even run on a developer PC connected to a printer) to ensure it picks up the entry and prints. - We will test PDF content compatibility (fonts, sizes) with the printer. - If printing from different OS (Android devices, etc., might require enabling external storage to share PDF via plugin, but since we do via cloud, that's fine). - Possibly implement a debug mode where instead of printing, the local service just saves the PDFs to a folder to verify content.

By implementing the print pipeline as above, we achieve **automatic printing** of needed documents. Parents benefit by not having to manually download and print files each day – the system essentially functions like the deprecated Google Cloud Print but specifically tailored [23] to our use case, using CUPS under the hood as recommended by Google for replacement [25].

Code Planning – Modules and Components

To build Operation Summer, we break the work down into distinct modules/systems (strategic planning) and detail the components or tasks needed for each (tactical planning). Below is a breakdown of the major modules and what needs to be built or configured for each:

1. Authentication & Profile Management Module

Strategic Goal: Implement user authentication and support multiple child profiles per parent. - *Frontend Components:* - **Login/Signup Screen** – Build UI for signing in (and sign up if needed). Integrate Firebase Auth (email/password and Google sign-in button). - **Profile Selector** – On the dashboard (and possibly as a persistent part of AppBar), create a widget to select which child's data is being viewed. This could be a dropdown of child names or a horizontal list of avatar icons. - **Profile Management Screen** – A screen or dialog where the parent can add/edit child profiles (name, age, maybe photo). On saving a new profile, call Firestore to create a new child document. If implementing separate child accounts, include an option to invite or create account (which might trigger a cloud function). - *Backend & Integration:* - **Auth Setup** – Enable providers in Firebase (Email/Password, Google). If Google sign-in, configure OAuth client IDs for iOS/Android as required by Firebase. - **Profile Data** – Cloud Functions if needed to enforce any invariants (for instance, limit number of child profiles per parent if we want). Possibly a function to handle creating a child Auth account and linking to parent (if multi-account route). - **Security Rules** – Write rules as described to ensure only authorized access. Test with a scenario of both parent and child roles if applicable.

2. Chore Management Module

Strategic Goal: Provide functionality to create, list, update, and sync chores. - *Frontend Components:* - **ChoreBoardScreen** – As detailed, with columns and drag/drop. Implement using Flutter's Draggable/DragTarget or integrate a `KanbanBoard` package [1]. Handle state updates on drop (optimistically update UI, and Firestore write). - **ChoreCard Widget** – Displays chore info. Possibly includes a checkbox or swipe-to-complete gesture in addition to drag. - **Add/Edit Chore Dialog** – Form for input. Possibly reuse for editing. - **Points Display** – A widget (maybe in Dashboard or Profile header) that shows the accumulated points. This subscribes to the child's Firestore doc for live updates. - *Backend & Integration:* - **Firestore Structure** – Set up the collections for chores. Ensure indexes if needed (e.g., composite index on childId and status if using a single collection, or on dueDate for date queries). - **Cloud Function: Points Award** – A triggered function on chore status change that updates points as described. - **Cloud Function: Sheet Sync** – Firestore onWrite trigger for chores -> update Google Sheet row. Use `googleapis` for Sheets inside the function [17]. - **Apps Script** – Write a Google Apps Script for the sheet for onEdit trigger. This script should call a webhook (we'll deploy an HTTP Cloud Function to receive it). Alternatively, the script could call the Firestore REST API directly using an API key, but for security the webhook (with a secret) is better. We will implement this script and document setup steps for the admin user. - **Testing** – Test drag-drop logic, ensure Firestore updates reflect correctly, verify points increment. Test that sheet sync works both ways (edit a chore in app -> sheet updates; edit sheet -> app data updates via function, maybe test by reading Firestore or UI).

3. Calendar & Scheduling Module

Strategic Goal: Integrate calendar features with Google Calendar and app data. - *Frontend Components:* - **CalendarScreen** – Use Syncfusion SfCalendar (or alternative). Implement month and week view toggling. Show events from combined sources. - **EventDataSource** – Code to fetch events from Google Calendar API. Likely create a Dart service class `CalendarService` with methods like `getEvents(DateRange)`, `addEvent(Event)`, `updateEvent(Event)`, `deleteEvent(id)`. This will wrap the googleapis calls. - **Event Details/Creation UI** – Possibly a simple bottom sheet or dialog when tapping an empty slot or pressing "+" to add an event (fields: title, date/time, etc., maybe a picker to choose which calendar if multiple). - **Drag/Drop Handling** – In `onDragEnd` of calendar: determine event and new time, call appropriate update function. If using Syncfusion, implement `onTap` or `onLongPress` to maybe start drag or open context. - *Backend & Integration:* - **Google API Setup** – Create OAuth client IDs for each platform, set redirect URIs, etc. The Flutter google_sign_in plugin requires configuration (e.g., an iOS URL scheme, Android SHA1). Document these steps. - **Cloud Function (optional)** – If we implement any backend process like adding events for chores, code that as a Firestore trigger or scheduled job. For instance, an onCreate of chore with dueDate -> call Google Calendar. This would require storing user credentials or using a service account's delegated access (complex), so likely skip or do on client. - **Testing** – Use a test Google Calendar to verify: events load, creating an event via app shows up in Google Calendar, dragging event in app changes it in Google Calendar (check via web or Google Calendar app), and vice versa (if event changed externally, verify app reload picks it up). Also test offline: what happens if no internet – ensure app doesn't crash, maybe show cached chores events or a message.

4. Learning Content (Lessons) Module

Strategic Goal: Manage daily lessons, PDF storage, and viewing. - *Frontend Components:* - **LearningHubScreen** – List of lessons per day. Possibly add a date picker or tabs for each week. - **LessonCard Widget** – Similar to chore card but for a lesson: shows title, maybe an icon if PDF is attached, and completion status. - **LessonDetailPage** – When a lesson is opened. Integrate a PDF viewer. We might use `flutter_pdfview` which requires the file path, so we'll need to download the PDF first (using Firebase Storage's `getData` or `getDownloadURL` and then `http` get). Alternatively, `printing` plugin's preview which can handle a network PDF. - **Print All Button** – In LearningHubScreen, perhaps in the AppBar or floating button. This triggers the print flow. - **Complete Lesson Toggle** – Perhaps a checkbox in LessonCard or a "Mark as done" button inside detail. This sets `completed=true` in Firestore (and potentially awards points or simply for tracking). - **Add Lesson (Admin)** – Possibly, a UI to allow the parent to add a lesson for a future date. Including uploading a PDF (integration with file picker and Firebase Storage upload). - *Backend & Integration:* - **Firestore & Storage** – Set up `lessons` subcollection and storage bucket security. Ensure that the authenticated user can upload/read their files. We might use Firebase Storage rules to allow only parent to write, children to read if needed. - **Cloud Function: Auto-schedule Print** – A scheduled function that looks at lessons of the day and enqueues print as discussed (the "morning auto print"). Cron schedule in firebase.json (using PubSub scheduler). - **Cloud Function: any PDF processing** – If we want, a function could generate a PDF for a lesson if the content is text-based (not likely here, since presumably lessons are provided as PDFs or links by the parent). - **Testing** – Upload a dummy PDF via app, ensure it appears and can be viewed. Test Print All triggers (maybe initially point the print service to print to PDF or another printer for safety). Check that multiple PDF printing works (the service prints each in order).

5. Printing Module

Strategic Goal: Implement the end-to-end pipeline for printing. - *Components/Services:* - **Cloud Function** `printAllMaterials` – As described, to handle incoming print requests. Implement using Node.js: gather Firestore data, get Storage URLs, write printJobs doc. Use Firebase Admin SDK to write to Firestore. - **Cloud Function** `dailyAutoPrint` – Scheduled trigger that at a set time calls similar logic for each relevant profile (or for one profile if only one). - **Local Print Service** – This is not part of Firebase deployment but part of project deliverables. We need to develop and configure it: - Could be Node (with `firebase-admin` and using `child_process` to call system print) or Python (with `google-cloud-firestore` and maybe `subprocess.call("lp ...")`). - We will write this script and include it in the documentation (maybe in a `tools/` directory in the repo). - Setup instructions: e.g. “Install Node, run `npm install firebase-admin cups`, set environment variable for Firebase credentials, and run `node print_service.js` on startup.” - **Testing Harness** – Possibly a mode in the app or a separate test function to simulate print events for testing (like generate a dummy PDF and run through pipeline). - *Integration:* - **CUPS/Printer config** – Document how to get the printer name and ensure it prints via command. For example, instruct the user to test a print using `lp` manually. - **Error Monitoring** – We might add a mechanism: if a print job fails, maybe the Cloud Function could send the parent a notification or email. Could integrate with Firebase Crashlytics or logs. At minimum, log errors in Firestore (status = error with message field). - **Security** – We'll locate printJobs under `users/{uid}` so only that user’s service listens/has access. Since admin SDK bypasses rules, the local service can see all, but if single user, it’s okay. If multi-family scenario, we would filter by parentId or such. - *Deployment of Local Service:* - If the user (developer/parent) has a Raspberry Pi, we provide setup scripts or a Docker container perhaps. A Docker approach: run a container on a local machine that includes CUPS and a small Node server that listens for print jobs (like some projects have combined these). - But given the specificity, probably a custom script is fine.

6. Google API Integration Module

Strategic Goal: Wrap Google Calendar and Sheets API usage for reuse and clarity. - *Components:* - **GoogleAuthService (Flutter)** – Manages signing in and storing scopes, token refreshing. - **CalendarService (Flutter)** – As above, for event operations. Could use the `googleapis` Dart package to avoid manual HTTP. E.g. use `calendarApi = CalendarApi(client)` where client is from google_sign_in auth token [3]. - **SheetsService (Flutter or Cloud)** – If we allow any direct reading of Sheets in app (maybe to display something), but more likely all Sheets ops in Cloud Functions. So maybe not needed on client. - **SheetsIntegration (Cloud)** – In Cloud Functions code, use Google APIs Node.js client. Manage auth: likely use a service account or API key for Apps Script webhooks. Write helper functions to add/update rows. - **Configuration** – Provide a config file or use Firebase Remote Config to store things like spreadsheet IDs or calendar IDs for default usage, unless those are per user (in which case store in Firestore). - *Build from scratch vs existing:* - We will largely build from scratch using official client libraries. There’s no off-the-shelf Firebase extension for Calendar, so this is custom. - We do use existing packages (googleapis for Dart and Node, Syncfusion for UI if chosen). - *Tactical tasks:* - Register the app with Google API Console (for OAuth consent screen etc.) and link it to Firebase project. Without this, GoogleSignIn for Calendar scope might not work. - Ensure Google API quotas are sufficient and handle errors (like rate limit: unlikely to hit with personal use). - Implement caching or minimal fetching to reduce API calls (like don’t fetch the same events repeatedly within a short time; maybe keep them in state or cache for a session).

7. Deployment & DevOps Module

Strategic Goal: Set up development, testing, and deployment pipelines. - *Local Development:* - Set up Flutter dev environment. Use Firebase Emulator Suite for Firestore during development to avoid messing with production data (especially for testing rules and Cloud Functions). - Provide fake Google API keys for dev if needed or use test accounts. - Test on both iOS Simulator and Android emulator for platform-specific issues (especially Google sign-in, which requires correct config). - Use Flutter's hot-reload during UI build, but be mindful of not calling real print accidentally in dev. - *CI/CD:* - Use a CI service (GitHub Actions or similar) to run tests and possibly build releases. For example, set up an action to build the Flutter app for web (if we deploy web) or to at least analyze code (dart analyze, flutter test). - For Cloud Functions, maybe use the Firebase CLI in CI to deploy functions on merges to main. (Since this is likely a personal app, could also deploy manually, but we outline CI for completeness.) - Ensure secrets (service account keys, API keys) are stored safely in CI (using encrypted secrets). - *Deployment:* - **Firebase Project:** Create a Firebase project, enable Firestore, Auth, Storage, Functions, etc. Set up Firestore rules from our rules file. - **App Stores:** If we intend mobile deployment, plan for App Store (iOS) and Google Play (Android) processes. That includes code signing, provisioning profiles for iOS, and preparing store listings. Possibly not needed if app is for personal use, but if sharing with others, this is relevant. - **Web Deployment:** If building as PWA/web, we can deploy via Firebase Hosting. That requires building `flutter build web` and deploying the output via Firebase. Hosting can be set up with our custom domain or Firebase's domain. - *Device Testing:* - We will test on actual devices if possible (especially to test Google sign-in on release mode, as it can have configuration differences). Also test printing end-to-end from a physical phone to ensure the pipeline triggers as expected (maybe have the Pi or print server on and see that it prints after pressing the button on phone). - Cross-platform considerations: - iOS: must configure application Transport Security if connecting to any non-https (but we mostly use https for Firebase and Google, so fine). Google Sign-In on iOS needs URL scheme in Info.plist. - Android: need to add SHA1 fingerprint in Firebase console for Google Sign In to work and download `google-services.json`. - Web: need to add the OAuth client ID for web in Firebase config and authorized domains. - Differences in UI: ensure drag-and-drop feels okay on mobile touch vs desktop web (it should; Syncfusion handles touch DnD well). - Ensure that if the app is backgrounded or device locked, and a scheduled print triggers, since that's cloud-driven it should still occur (no dependency on app being open).

- *Monitoring:*
- Set up Firebase Crashlytics for the app to catch runtime errors.
- Set up Firebase Performance maybe to see if any call is slow.
- Logging in Cloud Functions: use console.log and monitor via Firebase console. Possibly integrate an alert if the daily print function fails.

By breaking it down into these modules and tasks, we can assign development priorities and ensure nothing is missed. Many systems (Firebase Auth, Firestore, Google APIs) can be developed somewhat in parallel given their loose coupling, but integration testing will tie it all together.

Deployment Considerations

Successfully launching Operation Summer requires proper configuration of Firebase, Google Cloud services, development environments, and a plan for continuous integration/delivery. This section covers what needs to be done to prepare the project for deployment and maintenance.

Firebase Project Setup

- **Project and App Registration:** We will create a Firebase project (via Firebase console). Within it, create iOS, Android, and Web app entries to get the appropriate config files:
- For iOS: Add an app with the iOS bundle ID (e.g., com.yourcompany.operationsummer). Download `GoogleService-Info.plist` and include in the Xcode project (Flutter will handle this in ios/Runner usually).
- For Android: Add app with package name (e.g., com.yourcompany.operationsummer). Download `google-services.json` and place in `android/app/`.
- For Web: Enable it to get the Firebase config snippet (we'll put it in `index.html` via Flutter web or in init in code).
- **Enable Firestore:** In Firebase console, enable Cloud Firestore in production mode (we will apply rules). Plan for regional selection (choose a region close to user, e.g., us-central or where applicable).
- **Enable Authentication Providers:** Go to Auth > Sign-in methods. Enable Email/Password and Google. For Google, configure the project's OAuth consent screen if not already (give it a name, support email, etc., since Google sign-in for external scope like Calendar requires an OAuth consent screen). Add the authorized domain (e.g., your Firebase app domain) for web sign-in.
- **Firebase Storage:** Enable storage. Set up a default bucket. We'll set rules to allow the parent user to upload and others to read as appropriate (could be open read for certain files if not sensitive, but likely restrict to auth users).
- **Cloud Functions:** Ensure billing is enabled if needed. Note: calling external APIs (Google Sheets/Calendar) might require the Blaze plan (due to making external network requests). Also, Cloud Scheduler for scheduled functions requires Blaze (paid) plan. We will upgrade to Blaze (the free Spark plan wouldn't allow scheduled triggers).
- **Firestore Security Rules:** Deploy the rules written (based on earlier in spec). Test them using the Firebase emulator or the Rules Playground (simulate a child user vs parent user).
- **Indexes:** If we identified any Firestore composite indexes, define them in `firestore.indexes.json` and deploy. For example, if querying chores by `status` and `dueDate`, or children by parentId if we had that, etc.
- **Testing on Emulator:** Use `firebase emulators:start` to ensure functions, firestore, etc., work together locally. Particularly test the printJob creation and see if the local service can connect to emulator Firestore (the service can be pointed to emulator for dev testing).

Google Cloud Service Configuration

Operation Summer uses Google APIs (Calendar and Sheets), which require some Google Cloud Console setup: - **OAuth Consent and Publishing:** Because we request sensitive scopes (Calendar and Sheets are both user data scopes, beyond basic profile), we need to configure an OAuth consent screen in the Google Cloud Console. This includes providing a name ("Operation Summer App"), privacy policy URL (if distributing publicly, else for personal use it's just yourself, you can mark it internal if using a Google Workspace domain or just go through verification if needed for external). For testing, we can add our own account as a test user. - **API Enablement:** In Google Cloud Console, enable the Google Calendar API and Google Sheets API for the project associated with the OAuth client. (If using Firebase's default project, you can find it in Google Cloud Console). - **OAuth Client IDs:** - We need one for iOS (with the bundle ID), one for Android (with package name and SHA1), and one for Web (with authorized JavaScript origins, e.g., http://localhost:port for dev and the production domain for hosting). Firebase can create these for you if you use Firebase's Google sign-in, but for Calendar/Sheets scopes, we ensure those scopes are added. - Using GoogleSignIn in Flutter might auto-use a default client ID if Firebase project is linked. However, to be safe, create explicit OAuth clients in the Google Cloud Console Credentials section for each platform. - Update the Firebase Auth Google provider configuration if needed with the web client ID (sometimes Firebase Auth uses its own client). - For iOS, add the reversed client ID in the Info.plist URL schemes (FlutterFire docs guide through this). - **Service Account (for Sheets sync):** Create a service account in Google Cloud (in IAM > Service Accounts). Grant it the role "Editor" or specifically "Sheets API Editor" if that exists (or just use it with the API). - Generate a JSON key for this service account and store it securely (we might put it in Cloud Functions as an environment secret). - Share the target Google Sheet with the service account's email (so it has access). - In Cloud Functions code, use that JSON (via environment or directly uploading as part of code is not secure) to authorize the Google Sheets API calls. - **Cloud Function Environment Variables:** Use `firebase functions:config:set` or .env to store: - The spreadsheet ID(s) (if one global one). - Perhaps the service account JSON (though better to use Google's default credentials if functions service account has domain-wide access; using JSON might be easier given we can directly impersonate). - Printer default name (maybe not needed as code can use a generic or allow override). - **CUPS and Local Setup:** On the local machine with printer: - Install required software: CUPS, Node/Python, Firebase Admin SDK. - Ensure the machine has internet to talk to Firestore. - If behind a firewall, open port 443 for Firestore (outgoing). - Test CUPS by printing a test page. - Deploy our local script (e.g., as a systemd service, we provide a config file). - Possibly set up dynamic DNS or static IP if we ever wanted direct access, but we opted against needing that.

Local Development Environment

For developers or maintainers: - **Repository Structure:** Use a monorepo style or separate for Flutter app and functions. Likely Flutter project at root, with `functions/` folder for Cloud Functions code. Provide a README for how to run. - **Dev Secrets:** Use `.env` files or `firebase emulators:config` for local dev to mimic config vars. - **Emulator Use:** Document how to run the app against local emulator (like set Firestore settings to localhost in debug mode). - **Unit Testing:** Write some unit tests for critical pieces, e.g., a Dart test for a function that calculates points, or a simple widget test for drag-drop logic if possible (though drag-drop might require integration test). - **Integration Testing:** Perhaps write a Flutter Driver or integration test scenario: create a child, add a chore, mark it done, ensure points updated. - **Linting:** Ensure code passes flutter analyze and any lint rules. Possibly use a formatter and CI check.

CI/CD Pipeline

- **Continuous Integration:** Set up GitHub Actions:
- One workflow to build and test the Flutter app on pushes/PRs. This would use `actions/checkout`, set up Flutter (using subosito/flutter-action or similar), run `flutter analyze`, `flutter test`, maybe `flutter build web` to ensure it compiles.
- Another job for Cloud Functions: run `npm lint` (if eslint configured) and perhaps deploy to Firebase Emulators to run tests (if any). Could even run an emulator suite and run a test script to simulate a sheet update and see function response (advanced).
- Optionally, if everything passes on main branch, automatically deploy to Firebase (Functions deploy, Hosting deploy for web).
- **Continuous Delivery:**
- For mobile apps, manual steps are involved (upload to stores). However, we can automate builds: e.g., use Codemagic or GitHub Actions with macOS runner to build iOS IPA and Android APK. If it's just internal use, might not need to publish to store at all (just install directly or TestFlight).
- For web, a CI step can deploy to Firebase Hosting. We might map a custom domain like operationsummer.example.com or use the default project.web.app domain.

- **Release Management:** Use Git tags or branches for production vs dev. Possibly maintain a config that points to a dev Firebase project for testing separate from prod.

Device Testing and Platform Differences

- **Android:** After building an APK, enable Google Sign-In by adding SHA1 of the release keystore to Firebase console. Also update the OAuth client in Google Cloud with that SHA1. Ensure printing code (though mostly cloud) doesn't require any Android permission (likely not, since not printing locally from device).
- Test on an actual Android phone the Google sign-in flow. It should show the OAuth consent asking for Calendar/Sheets access. Because these are sensitive scopes, on first time, Google will show a warning if the app isn't verified. If it's just personal, we can live with that or publish the app as an internal app in our Google account.
- **iOS:** Set up an Apple Developer profile for signing if needed. Add any needed permissions: likely none special. If using the camera or storage for picking PDFs, add those usage descriptions. We might need to allow arbitrary loads if loading PDF from a URL (perhaps not if using https).
- Test Google Sign-In on iOS. The GoogleSignIn SDK on iOS might require configuring URL schemes and maybe adding the scope in code. Possibly use the `googleapis` package (which might need a separate approach on iOS if the user isn't asked for those scopes by Firebase by default).
- Might use a fallback: use `url_launcher` to open a Google auth URL if needed, but the recommended way is via the plugin.
- **Web:** Test on web (Chrome, etc.), ensure that the popup for Google auth works (need to add localhost and deployed domain to Authorized domains in Firebase Auth settings). On web, printing pipeline still works since it's cloud; the user might expect clicking Print All to perhaps show their browser print dialog, but we are doing remote print. We should maybe clarify UI text like "Print to home printer" so they know it's not printing locally attached printer of the device but the one configured.
- On web, some plugins have limitations (e.g., `flutter_pdfview` might not work on web). We may need a web-specific approach to PDF preview (maybe just open the PDF in a new tab or use `<iframe>`).

- Drag-and-drop in web might conflict with the browser's native DnD for images, but Flutter's widgets should handle it internally via HTML5 drag events. We'll test that it doesn't try to drag data outside the app.

- **Responsive Design:** Ensure the UI works on tablet vs phone vs web sizes. Possibly adjust the layout: e.g. on a tablet or web, maybe you can show the Chore Board and Calendar side by side. Not required, but could be a nice enhancement. For now, focus on making each screen responsive enough (scrollable lists, etc., so no overflow).

By addressing these deployment considerations, we can confidently move from development to a live, usable application. The documentation and configuration steps also ensure that the project can be collaborated on (via GitHub or Notion for planning) and that others can set it up if needed. Each step (from creating OAuth credentials to starting the print service) will be clearly documented in the project README for future reference.

---

[1] dart - How to Create a Kanban App with Flutter and Enable Drag-and-Drop Feature? - Stack Overflow
https://stackoverflow.com/questions/77086625/how-to-create-a-kanban-app-with-flutter-and-enable-drag-and-drop-feature

[2] [18] Drag and drop in Flutter Event Calendar widget | Syncfusion
https://help.syncfusion.com/flutter/calendar/drag-drop

[3] [4] [5] [6] Google APIs | Flutter
https://docs.flutter.dev/data-and-backend/google-apis

[7] How to load an existing pdf and print using printing in flutter
https://stackoverflow.com/questions/60166702/how-to-load-an-existing-pdf-and-print-using-printing-in-flutter

[8] printing | Flutter package - Pub.dev
https://pub.dev/packages/printing

[9] Secure data access for users and groups | Firestore | Firebase
https://firebase.google.com/docs/firestore/solutions/role-based-access

[10] [11] [12] [13] [14] [15] [16] [17] [19] [20] Hybrid Database with Firebase Firestore and Google Sheets | by David Lopez | Medium
https://medium.com/@degl0412/hybrid-database-with-firebase-firestore-and-google-sheets-a7bde8e12b32

[21] Firebase / Firestore Google Sheets Integration - Quick Connect
https://zapier.com/apps/firebase/integrations/google-sheets

[22] c# - How to send a Label Print Job to a network CUPS/Raspberry Pi ...
https://stackoverflow.com/questions/63347725/how-to-send-a-label-print-job-to-a-network-cups-raspberry-pi-print-server-from-a

[23] [25] Migrate from Cloud Print - Chrome Enterprise and Education Help
https://support.google.com/chrome/a/answer/9633006?hl=en

[24] Chapter 6. Adding a printer to CUPS by using the web interface
https://docs.redhat.com/en/documentation/red_hat_enterprise_linux/9/html/configuring_and_using_a_cups_printing_server/adding-a-printer-to-cups-by-using-the-web-interface_configuring-printing