ChatGPT

Operation Summer – Technical Specification

Executive Summary

Operation Summer is a daily content generation and delivery platform that provides summer educational activities for children through automated AI and cloud services. It integrates a **Flutter** cross-platform application (web and mobile) with a **Firebase** backend and a no-code/low-code **automation pipeline** (via n8n) to create and distribute personalized content each day. The system uses **OpenAI's GPT-4** for text generation, **DALL·E** for image generation, **text-to-speech (TTS)** for voiceovers, and the **Creatomate API** for automated video assembly. Google services (Calendar, Sheets, Drive) are integrated to enhance scheduling and data management, and the solution supports **automatic printing** of daily materials on a shared network printer.

Each daily "package" of content includes:

1. **Personalized Intro Video** – A short video for each child, featuring a GPT-generated script narrated via AI voice (TTS) and assembled by Creatomate.
2. **Coloring Page** – An image (line-art coloring page) generated by DALL·E, annotated with the child's name.
3. **Worksheet (PDF)** – A one-page educational worksheet (questions, puzzles, etc.) generated by GPT, tailored to the child's age, rendered as PDF.
4. **Group Project/Craft Suggestion** – A collaborative project idea for all children to do together (text-based).
5. **Physical Exercise/Challenge** – A fun physical activity or challenge for the group (text-based).
6. **YouTube Song Recommendation** – A child-friendly song (with a YouTube link) matching the day's theme.

The daily content is generated automatically every day (via a scheduled n8n workflow) and stored in **Firebase Storage**. The system then triggers printing of the printable materials (coloring pages, worksheets, and a summary page for group activities) on a local network printer. The Flutter application allows the user (parent or educator) to configure children's profiles, view the daily content digitally, adjust settings (e.g. scheduling, themes), and manage integrations (Google account linking, printer setup). This document provides a comprehensive specification for developers, detailing the architecture, components, data models, and integration steps needed to implement Operation Summer.

Architecture Overview

Architecture Summary: Operation Summer is built on a multi-tier architecture with a client-side Flutter app, a serverless Firebase backend, and an external automation/orchestration layer for AI content generation and printing. The system components and their interactions are outlined below:

- **Flutter Frontend (Web/Mobile):** Provides the user interface for configuring the system and viewing content. It uses Firebase Auth for authentication, Firestore for real-time data (child profiles, content

metadata), and Firebase Storage to retrieve generated media (videos, images, PDFs). The Flutter app runs on iOS, Android, and web (deployable via Firebase Hosting) from a single codebase.

- **Firebase Backend:** Acts as the primary backend and data store. It includes **Firestore** (a NoSQL database) to store structured data such as user accounts, child profiles, daily content metadata (links, statuses), and integration tokens. **Firebase Auth** handles user authentication (with support for email/password and Google Sign-In for OAuth). **Firebase Storage** holds large content files (videos, PDFs, images). Cloud Functions may be used for secure server-side operations (e.g., handling Google OAuth callbacks, or performing privileged tasks with Google APIs) and to enforce business logic or connect with external APIs if needed.

- **Automation & Integration Layer (n8n Workflow):** An n8n workflow (running on a server or local machine) orchestrates the daily content generation and distribution. At a scheduled time each day, the workflow triggers and performs the following:

- Calls **OpenAI GPT-4** APIs to generate the day's content (video script, worksheet content, project/exercise ideas, song suggestion).
- Calls **DALL·E** to generate the coloring page image based on the theme or prompt.
- Calls a **TTS service** (e.g., ElevenLabs or Google Cloud Text-to-Speech) to generate an audio narration for the video intro script.
- Uses **Creatomate's Video Generation API** to assemble the intro video by combining visual elements (optionally theme images or text) with the TTS audio. (Creatomate natively integrates with AI tools like ChatGPT scripts, DALL·E images, and ElevenLabs voiceovers [1].)
- Uploads the generated assets to Firebase Storage (video MP4, worksheet PDF, coloring page image/PDF, etc.) and updates Firestore with references and metadata.

- Initiates **printing** of the new content on the network printer (if the n8n instance has access to the printer) or triggers a downstream print service.

- **Google Services Integration:** The system optionally integrates with Google Calendar, Sheets, and Drive for enhanced functionality:

- *Calendar:* The app can create calendar events for daily activities (e.g., schedule the project time or exercise time on the user's Google Calendar).
- *Sheets:* The system can log daily summaries or track children's progress in a Google Sheet, or read configuration data (like a list of predefined themes or user-provided content ideas) from a Sheet.

- *Drive:* Generated content files (PDFs, images, video links) can be automatically uploaded to the user's Google Drive (e.g., to a designated "Operation Summer" folder) for backup or sharing.

- **Network Printer:** A printer on the local network receives print jobs for the daily materials. Because Google Cloud Print is deprecated [2], printing is handled via the local network using OS-level printing (CUPS/Windows Print Spooler) or third-party print utilities. The n8n workflow (or a companion local service) will send the PDFs/images to the printer queue using standard protocols (IPP/LPR or vendor SDK).

Data Flow Overview: Each day, the automation layer generates content and populates Firebase. The Flutter frontend receives a notification or simply detects new content via Firestore update and presents it to the user. The content includes downloadable links from Firebase Storage (protected by Firebase security rules). The printing is triggered automatically by the workflow, but the user can also manually re-trigger printing from the app if needed (for example, if a printout was missed). The Google Calendar integration ensures the parent's schedule reflects the planned activities, and all content is archived (in Firestore/Drive/Sheets as configured).

Scalability & Hosting: All cloud components (Firestore, Storage, Functions, Hosting) reside in a Firebase project (which auto-scales). The n8n service can be self-hosted (on a cloud VM, local server, or even a Raspberry Pi on-premise for direct printer access). Multiple users (families) can be supported by segregating data in Firestore per user account. Flutter's cross-platform nature allows easy distribution to multiple platforms, and Firebase scales to potentially many users, though initial usage might be within a single household or a small group.

Security: Each user's data is protected via Firebase Auth – users must sign in to access their Operation Summer content. Firestore security rules ensure users can only read/write their own documents. Sensitive API keys (OpenAI, Creatomate, etc.) are kept out of the client; they reside in n8n or Cloud Functions (server side). OAuth tokens for Google integrations are stored securely (in Firestore or Firebase Auth custom claims) with refresh tokens to allow offline access. Content generated by AI is moderated for safety (e.g., using OpenAI's content filters) to ensure suitability for children. Printing occurs within the local network to avoid exposing content to external services (unless using a third-party cloud print service by choice).

Strategic Component Breakdown

The system can be broken down into several core components/modules, each responsible for specific functionality. Below is a high-level list of these components and the custom development required for each:

- **1. Flutter Application (UI/UX):** The cross-platform client app providing user interface for Operation Summer. *Must be built from scratch.* This includes screens for user authentication, child profile management, viewing daily content (video player, image/PDF viewer, etc.), settings for integrations (Google account link, printer setup info), and manual controls (e.g., trigger content generation or printing if needed). We will design a responsive UI suitable for both web and mobile.

- **2. Firebase Backend Configuration:** The Firebase project setup and data structure. *Partially configuration, partially code.* We need to design Firestore collections (for users, children, daily content, etc.), set up Firebase Auth providers (email login, Google OAuth), configure Firebase Storage buckets/paths, and implement Cloud Functions for any server-side tasks (especially handling Google OAuth callbacks or any admin tasks that should not be done on client). Security rules for Firestore/Storage must be written to protect data.

- **3. Content Generation Pipeline (n8n Workflow):** The automation sequence that generates all pieces of content daily. *This involves custom logic and possibly small code snippets in n8n.* It integrates with external APIs (OpenAI, TTS, Creatomate, Firebase) in a specific sequence. We must build or configure n8n nodes for: scheduling trigger, OpenAI (GPT-4 text generation and DALL·E image generation) [3], possibly a TTS node or HTTP call to a TTS API, an HTTP Request to Creatomate API, and Firebase integration (likely via HTTP APIs or using a code node with Firebase SDK) to upload

results. Also included here is logic for error handling (e.g., retry on failures, fallback to alternatives) and notifications (possibly send a push or email when done).

- **4. Video Generation (Creatomate Integration):** A sub-component of the pipeline focusing on producing the intro video. *This involves creating a video template in Creatomate and using their API.* We need to design a template that takes dynamic text (script) and an audio track (TTS) to produce a personalized video for each child. The development includes setting up the template on Creatomate's platform and writing the API call to render the video with provided parameters. We might also integrate an image into the video (like a title card or the child's name overlay). Creatomate's API supports setting template modifications (like inserting voiceover text and image URLs) to generate the output [4] [5].

- **5. Google API Integration Module:** Handles connecting with Google Calendar, Sheets, and Drive. *Will require OAuth flow and API calls.* Development tasks include setting up OAuth consent (scopes for Calendar events insertion, Drive file creation, Sheets editing), storing tokens, refreshing tokens, and making calls to Google APIs:

- Calendar API: create events for activities.
- Sheets API: log daily content or read configuration data.

- Drive API: upload files (PDFs, images) to a user's Drive folder. This module will likely be implemented with Cloud Functions or in the Flutter app (with Google APIs client libraries), plus the necessary UI for users to sign in and manage permissions.

- **6. Printing Integration:** Ensures the daily PDFs/images are automatically printed. *Custom integration needed.* Because no off-the-shelf cloud print exists, we need to implement a solution such as:

- Running n8n on a machine that has access to the printer, using an Execute Command or custom node to send the print job to the printer.

- Alternatively, a small custom print server (Node.js service or script) that listens for new files (triggered by Firestore or n8n) and prints them via OS commands or a library. Development involves writing the logic to send files to printer (e.g., using Node libraries like **pdf-to-printer** [6] or system commands through CUPS/Windows spooler).

- **7. Data Models & Cloud Functions (Backend logic):** This includes any backend logic not covered in n8n. For example, Cloud Functions to:

- Process OAuth callbacks (to securely receive Google auth code and exchange for tokens, storing them).
- Provide an API endpoint for the Flutter app to trigger content generation on demand (this could invoke the n8n webhook).

- Enforce any complex security rules or perform post-processing (e.g., if using Firestore triggers to notify n8n or send push notifications). Data modeling involves defining the structure for Firestore documents (e.g., `users/{userId}/children/{childId}`, `users/{userId}/dailyContent/{date}` etc.) and any necessary indexes.

- **8. Deployment & DevOps:** Setting up the CI/CD pipeline, testing frameworks, and deployment strategies. *This includes scripts and configuration.* We need to ensure:

- Flutter app can be built and deployed (web app to Firebase Hosting, mobile apps to stores or distribution).
- Firebase Functions and security rules are deployed properly.
- n8n is deployed (e.g., via Docker) and configured with required environment variables (API keys, service account for Firebase).
- Continuous Integration (CI) for automated testing and Continuous Deployment for pushing updates (especially for the app and cloud functions).

Each of these components is elaborated in the next section, with tactical implementation details for each module.

Tactical Development Plan

We now break down the development plan for each major module, covering UI, data models, logic, and integration points in detail.

1. Frontend: Flutter Application

The Flutter app is the primary interface for users. It will be developed as a **cross-platform** app targeting web, iOS, and Android. Key aspects include UI design, state management, network/API calls, and integration with Firebase services.

1.1 UI Design & Screens: The app should have a clean, child-friendly but parent-oriented interface (since parents/educators will likely use the app to set up content for kids). We will implement the following screens/pages (Flutter `Widget` trees):

- **Login/Onboarding Screen:** Allows user to sign up or log in using Firebase Auth. Provide options for email/password or "Sign in with Google". If using Google for OAuth to also access Calendar/Drive, we'll handle additional consent (more on that in integration section).

- **Child Profiles Screen:** Allows the user to add and manage child profiles. Each profile includes at minimum the child's name and age. Possibly also a photo (optional) and preferences (could be favorite themes or interests to personalize content). This screen will list all children and an "Add Child" form. Data binds to Firestore (collection: `children` under the user's document).

- **Daily Content Overview Screen:** The home screen, showing today's generated content. If multiple children, it may allow selection of a child to view their specific materials.

- **Intro Video card:** An embedded video player (using a Flutter video_player plugin or an HTML `<video>` for web) that plays the personalized intro video for the selected child. The video file will be fetched from Firebase Storage (via a secure URL).
- **Coloring Page card:** A thumbnail or preview of the coloring page image (PNG/PDF). Possibly tapping it opens a full-screen view or downloads the PDF. The child's name is displayed with the image.

- **Worksheet card:** A preview or icon of the worksheet PDF. Tapping it can open a PDF view (using a PDF viewer plugin) or simply prompt download.
- **Group Activities card:** A section listing the project idea, exercise challenge, and the song recommendation for the day (with perhaps a clickable link to open the YouTube video). These can be text displayed in the app. For the song, we can show the title and maybe a thumbnail if we use YouTube Data API, but a simple link is sufficient.

- **Print Status/Action:** An indicator of whether the materials have been printed. If automatic print succeeded, show "Printed" (maybe with timestamp). If not, or if user wants to re-print, provide a "Print Again" button. This button will trigger a backend call (to Cloud Function or to n8n webhook) to send the print job again.

- **Settings/Integration Screen:** Allows configuration of various settings:

- Google Account linking: show whether a Google account is linked for Calendar/Drive/Sheets, and initiate OAuth if not. Allow user to enable/disable specific integrations (e.g., a toggle for "Add activities to Google Calendar", "Backup content to Google Drive", "Log to Google Sheet").
- Printer setup: instructions or status for printer connection. Because printing is handled by the backend, we might simply indicate if the system has a known printer. (If running n8n locally, perhaps the user would configure printer name in an env variable or config file, not via app. Alternatively, allow specifying a printer name or IP in the app, which the backend could use if dynamic. But likely simpler: assume a default printer or configuration on the server side.)
- Notification settings: optionally, toggles for receiving a push notification each day when content is ready.
- Theme preferences: optionally allow user to input or choose themes for certain days or exclude certain topics if desired (could be a multi-select of topics kids like, to guide GPT).

- Manage account (sign out, etc).

- **Past Content Archive Screen (optional):** The user might want to view previous days' content. We can implement a simple list of past dates (or a calendar view) that have content, allowing them to tap a date and see what was generated. This would fetch data from Firestore (or the user's Google Drive if we only keep archives there). This is a nice-to-have to ensure content is not lost after the day. We'll include this if time permits.

1.2 State Management: We will likely use a state management approach like **Provider** or **Riverpod** for managing global state (like the logged-in user, list of children, and today's content data) throughout the app. Flutter's reactive UI with streams from Firestore can update UI in real-time as data comes in (e.g., when the daily content generation writes to Firestore, the app could automatically show the new content, especially on web). Specifically: - Use a `StreamBuilder` or `Firestore Listener` for the daily content document to update the UI when content is ready. - Maintain an in-memory model of children and current selected child (so that switching child updates the displayed content). - Use a separate provider for integration settings and statuses (e.g., if Google account linked).

We will keep widget tree rebuilding efficient by splitting into small widgets and using keys where necessary. Navigation between screens can use Flutter's Navigator (with route names or using a higher-level router if needed). All pages will be accessible via a Drawer or BottomNavigation if appropriate (e.g., Home, Profiles, Settings).

1.3 Data Models (Flutter-side): Define Dart classes to represent our domain objects: - `UserProfile` (if needed beyond Firebase Auth basic user info). - `Child` with fields: `id`, `name`, `age`, maybe `preferences` (list of strings), and possibly a reference to their content or last content date. - `DailyContent` with fields corresponding to the generated content for a given day (or for each child): - If per child: fields for video URL, worksheet URL, coloring image URL, etc., plus maybe theme and the group items (project, exercise, song) might be duplicated per child or stored once. - We might split: e.g. a `DailyContent` doc per day per user containing group items and maybe list of children's items, or separate child-specific docs. For simplicity, consider each child has their own daily content entry, plus one common entry for group activities. We'll clarify in data model section below. - `Settings` model for things like default print times, calendar integration toggles, etc. (These could also be part of the user's Firestore document.)

These models will mostly mirror Firestore documents. We can use JSON serialization or manually map Firestore data to Dart objects using constructors.

1.4 Networking & API Calls: The Flutter app itself will rely primarily on Firebase for data, which means minimal direct REST API calls. Firestore and Storage have SDKs that we will use: - **Firestore:** Use `cloud_firestore` Flutter plugin to read/write data. For example, to get today's content: a query like `FirebaseFirestore.instance.collection('users/{uid}/dailyContent').doc(todayDate).snapshots()` to listen for updates. - **Storage:** Use `firebase_storage` plugin to get download URLs for files. We might use Firebase Security Rules to allow read access only to authenticated users on their files, and then use the SDK to fetch the file or an URL. If files are not too large, we could download them and display. For video, better to obtain a URL and feed it to the video player. - **Cloud Functions:** If we implement certain actions as callable functions (e.g., to trigger generation or printing manually), we'll use `firebase_functions` plugin to call them. For OAuth with Google, if using Cloud Functions for callback, the app might not directly call that but open a browser. More on that later. - **External APIs:** The app might not directly call OpenAI or Google APIs (those are handled by backend or n8n), except possibly for YouTube data if we wanted to fetch a thumbnail for the song (non-essential).

1.5 Flutter Component Tree Example: *(This illustrates how UI widgets might be structured in the app.)*

- **MainApp (MaterialApp)**
  - ├── **AuthGate** (decides whether to show Login or Main UI based on auth state)
  - └── **HomeScreen** (if logged in)
    - ├── **DrawerNavigation** (with links to Profiles, Settings, Archive, etc.)
    - └── **DailyContentScreen** (default view, shows today's content)
      - Child selector dropdown (if multiple children)
      - **VideoCard** – contains VideoPlayer or thumbnail + play button
      - **ColoringCard** – shows child's coloring image (thumbnail) and name
      - **WorksheetCard** – shows worksheet title or icon and allows open/download
      - **GroupActivitiesCard** – lists project idea, exercise, and song link (maybe with an icon or small image for YouTube)
      - **PrintStatusBar** – indicates printed status and offers reprint action

- **ProfilesScreen** (list of Child profiles)
  ├── List of **ChildTile** (name, age, edit button)
  └── **AddChildButton** (floating button or list tile to add new)

- **AddEditChildScreen** (for adding or editing a child profile) – form fields, save to Firestore.

- **SettingsScreen**
  ├── Google Integration Section (shows Google account email if linked, and toggles: e.g., [ ] Sync to Calendar, [ ] Save to Drive, [ ] Use Sheets log)
  ├── Printer Section (if applicable, perhaps a text like "Printer configured: MyPrinter on 192.168.x.x" or instructions "Ensure the server is connected to printer X")
  ├── Notification Section (e.g., [ ] Daily reminder notifications)
  └── Account Section (Sign out button, etc.)

- **ArchiveScreen** (Past Content)
  ├── Could be a calendar picker or list of dates
  └── When a date is selected, similar UI as DailyContentScreen but for that date's data (if we store historical content).

1.6 UX Considerations: Keep interactions simple – e.g., the daily content screen is mostly read-only content display (except print button). Ensure that if content is not yet generated (say user opens app early), the UI should indicate "Content is being prepared" and possibly show a loading state until Firestore updates. Likewise, handle error states (if generation failed, show a message and allow retry). The design should use friendly illustrations/icons since this is about kids' content (even if kids might not directly use the app, a playful theme is fine).

Additionally, ensure responsiveness: on mobile, cards stack in a scroll; on tablet/web, maybe two-column layout (video and coloring side by side, etc).

2. Backend Services & Data Models

The backend uses **Firebase** for most services. Here we detail the data model (Firestore) and any server-side logic (Cloud Functions, security rules, etc).

2.1 Firestore Data Model: We propose the following collections and documents structure in Cloud Firestore (in a **multi-tenant** approach keyed by user):

- **Users Collection (`users`):** Each authenticated user (parent) may have a document here (with ID = user's UID). This can store high-level info and settings.
- Fields: `displayName`, `email` (for convenience, though available via Auth), `googleLinked` (bool), `googleUid` or email of linked Google account, `calendarEnabled` (bool), `driveEnabled` (bool), `sheetsEnabled` (bool), `printerName` or `printerIP` (optional), etc. Also perhaps default times for activities or a `dailySchedule` template.

- **Children Collection (`users/{userId}/children`):** Stores child profiles for that user.

- Each child document fields: `name` (string), `age` (number), `preferences` (array of strings, optional), `lastGeneratedDate` (date of last content generated), `videoUrl`, `worksheetUrl`, `coloringUrl` (these last three could be in a separate content collection instead; see below).

- Alternatively, keep content separate. It might be cleaner to separate profile from content, so child doc just has static info (name, age, etc).

- **Daily Content Collection (** `users/{userId}/dailyContent` **)** (or possibly per child):

- This is a bit tricky because some content is per child and some is group-wide. We have options:
    1. **Option A:** One document per day per user that contains all info for that day, structured with subfields or subcollections for each child. For example, doc with ID = date (e.g., "2025-07-15") containing:
    2. `theme` (string, optional theme of the day).
    3. `projectIdea` (string), `exercise` (string), `songTitle` (string), `songUrl` (string).
    4. `childrenContent`: map of childId to an object with that child's content:
        - e.g., `childrenContent.child123.videoUrl = ...`, `childrenContent.child123.worksheetUrl = ...`, `childrenContent.child123.coloringUrl = ...`.
    5. Or we can use subcollection: `dailyContent/{date}/children/{childId}` docs with those URLs and maybe text summary if any.
    6. **Option B:** Separate collection per child for their daily items, plus a common one for group. For example:
    7. `users/{userId}/children/{childId}/dailyContent/{date}` documents with fields: `videoUrl`, `worksheetUrl`, `coloringUrl`, (and possibly any child-specific content text if needed, though most generated text is group-level).
    8. And `users/{userId}/dailyContent/{date}` for group stuff (theme, project, exercise, song). This way, child-specific content is easy to query (e.g., you could get all content for one child if needed). The Flutter app when showing today's content for a child would read both the child's dailyContent doc and the group dailyContent doc for that date.
    9. **Option C:** A hybrid, where each child document has fields updated daily for the latest content (like last video URL, etc.) and maybe an array of past content references. But it's generally better to not overwrite history if we want an archive.

We will implement **Option B** for clarity and historical tracking: - A **GroupDailyContent** document per date (under user) with group-wide info. - A **ChildDailyContent** document per child per date (under each child subcollection) with individual assets.

Example:
`users/UID/dailyContent/2025-07-15`
Fields: `{date: 2025-07-15, theme: "Space Adventures", project: "Build a model rocket...", exercise: "Moon rock scavenger hunt...", songTitle: "Planet Song", songUrl: "https://youtu.be/...", generatedAt: timestamp}`.

`users/UID/children/child123/dailyContent/2025-07-15`
Fields: `{videoStoragePath: "videos/UID/2025-07-15_child123.mp4", worksheetPath: "worksheets/UID/2025-07-15_child123.pdf", coloringPath: "coloring/UID/`

`2025-07-15_child123.png", status: "ready", ...}` (We can store just storage paths or full download URLs. Storing paths is safer, then client can get the download URL via Firebase SDK ensuring security rules apply. Alternatively, store download URLs if we set them to be accessible via token.)

Additionally, a top-level `dailyContent` might not be strictly necessary to store (we could derive group tasks by looking at one child's doc that includes theme, etc.), but explicitly storing group content in one place avoids duplication and inconsistency if multiple children.

Security & Access: Firestore security rules will ensure that: - Only authenticated users can read/write their own documents (`userId == request.auth.uid`). - A user can read their children and dailyContent, but not others'. - Possibly allow read-only access to dailyContent for the mobile app without logging in if we wanted an "agent" mode, but not needed. Everything stays behind auth.

- **Integration Tokens Collection (optional, could be part of user doc):** We might store Google OAuth refresh tokens and related info in Firestore as well. For example: `users/{userId}/integrationTokens/google` doc containing `refresh_token`, `access_token` (if storing temporarily), `scopes`, `token_expiry`, etc. Since this is sensitive, we must secure it (only server or that user can access). Alternatively, use Firebase Functions with **Secrets** or store in an encrypted form in Firestore. Given only our app uses it, storing in Firestore is acceptable with strong security rules (only the user and privileged cloud functions can read). We will mark this for careful handling.

2.2 Firebase Storage: We will use a single Firebase Storage bucket (default) with structured folders for different content types: - `videos/{userId}/{date}_{childName}.mp4` for intro videos. - `worksheets/{userId}/{date}_{childName}.pdf` for worksheets. - `coloring/{userId}/{date}_{childName}.png` (or .pdf) for coloring pages. - `group/{userId}/{date}_activities.pdf` for the combined group activities page (if we choose to generate a PDF containing the project, exercise, song info for printing).

Naming convention includes date and child name (or ID) for clarity and to avoid collisions. ChildName in filename can be nice for human admin, but we will rely on IDs and meta to ensure uniqueness in case of duplicate names.

Storage security: Only allow owners to read. For writing, our server (n8n with service account or Cloud Function) will do the uploading with admin privileges, or we can create a Firebase Storage token for direct upload if needed. Typically, we will use Firebase Admin SDK in n8n, which bypasses security rules (as admin), so that's fine. Then end-users (through the app) will only read, which can be done if we either (a) make the files public-but-obscure (not ideal) or (b) have the app use Firebase SDK which includes the user's auth token to authorize the download. Using the SDK's `getDownloadURL()` on a storage ref will check rules and return a time-limited URL if permitted.

2.3 Cloud Functions (API and Logic): We will implement a few Cloud Functions for cases that require server-side execution not covered by n8n: - **Google OAuth Callback Function:** When the user links a Google account, Google will redirect to a callback URI. We can use a Firebase Function as an HTTPS endpoint (e.g., `/oauthCallback`) to receive the `code`, then use Google APIs to exchange it for access and refresh tokens. This function will then store the refresh token in Firestore (or in Firebase Auth custom claims if we want, but Firestore is fine) and perhaps mark the user's integration toggles true. This function must be deployed with the appropriate Google OAuth client credentials configured (client ID/secret). -

Manual Trigger Content Generation: In case the user wants to regenerate or generate off-schedule (say they sign up in midday and want content immediately), we will provide a callable function (e.g., `generateTodayContent(userId)`). This function can either directly invoke the same logic as n8n (not ideal to duplicate logic) or, better, send a webhook to n8n to trigger the workflow on-demand. Simpler: the function could create a Firestore entry or message that n8n is watching. Alternatively, skip Cloud Function and let the Flutter app call a special **n8n webhook URL** directly (if n8n is exposed via a REST endpoint). However, exposing n8n to the internet requires security (API key or Basic Auth). We can protect it by requiring a secret or check in n8n. But using a Cloud Function as proxy is another layer of security. For this spec, assume we either: - Use a Cloud Function that calls n8n's webhook (the function itself being secured via Firebase Auth). - Or directly call an authenticated n8n webhook from the client (embedding a token). The Cloud Function approach is cleaner in a multi-user scenario because n8n can then run with elevated perms and we can log. - **Print Trigger Function (optional):** If the user hits "Print Again" in the app, a function could handle that by either: - Calling a local network service (if available via some API) to trigger print, or - Setting a flag in Firestore that n8n monitors (e.g., a subcollection "printJobs" where adding a doc causes the local n8n to pick it up and do the printing). For simplicity, we might have the app simply add a Firestore doc like `users/UID/printQueue/{date}` and n8n's workflow could watch that collection (via a trigger node or polling) to execute the print. This avoids exposing local network to cloud. We'll document this strategy in the printing integration.

- **Notification Function (optional):** If using FCM for push notifications to the app (for "content ready"), we'd use Cloud Messaging. A Cloud Function triggered on the Firestore write of new content can send an FCM message to the user's devices. This is optional; we can easily add it later for better UX.

All Cloud Functions will be implemented in Node.js (using Firebase Admin SDK for Firestore, and Google client libraries for OAuth exchanges, etc.). They will be deployed via Firebase. Environment configuration (like API keys or OAuth secrets) will be stored using Firebase **Functions Config** or **Secret Manager**, not in code.

2.4 Firebase Security Rules: We will implement rules roughly as: - For Firestore:

```
match /users/{userId}/... {
  allow read, write: if request.auth.uid == userId;
}
```

This covers all subcollections. We'll refine for specific cases (maybe read-only on dailyContent for others is never needed, so locked to owner only). Also ensure integration tokens are perhaps writeable only by server if needed (we can enforce that normal clients cannot directly write a `integrationTokens` doc, only the function with admin privileges can – might do this by requiring a custom claim or a specific field presence). - For Storage: We can structure rules such that:

```
match /videos/{uid}/{file=**} {
  allow read: if request.auth.uid == uid;
  allow write: if request.auth.uid == uid;  // though writes mainly by server
}
```

Similar for other folders (worksheets, etc.). If using admin for writes, we don't even need to allow writes for clients at all, only reads. So we might say `allow write: if false` on those to prevent any unknown uploads, and let admin SDK bypass rules.

- For Cloud Functions (HTTP): We will secure endpoints by either checking Firebase Auth token in requests (if the app calls them) or using secret keys for the OAuth callback (less needed – though we might include a state parameter in OAuth to prevent CSRF).

2.5 API Contracts for Backend Services: The Flutter app interacts mainly via Firestore and a few Cloud Function calls. We define these "API" contracts:

- **Firestore Data API:** (not REST, but via SDK)
- Get Children: `GET users/{uid}/children` – returns list of children documents.
- Add/Update Child: `PUT/POST users/{uid}/children/{childId}` – create or update a child profile.
- Listen Daily Content: subscribe to `users/{uid}/dailyContent/{today}` and `users/{uid}/children/{childId}/dailyContent/{today}` for updates. The data contains the URLs or paths for content and text fields for group content.

- (If archive screen) Query past content: e.g., list `users/{uid}/dailyContent` documents or query by date range.

- **Cloud Function: generateContentNow** (callable):

- **Request:** `{ date?: string }` (if date omitted, assume today, or allow generating for a given date).
- **Response:** `{ success: bool, message: string }` – likely returns immediately that generation has started or queued (since actual generation is asynchronous and handled by n8n). We might return a quick acknowledgement and rely on Firestore updates for result. If we want to be fancy, we could have the function actually perform the generation by calling GPT, etc. But given complexity, better to offload to n8n and just return started status.

- **Security:** Only authenticated users can call, and function will verify the UID matches context.

- **Cloud Function: printContentNow** (callable or HTTP):

- **Request:** `{ date: string, childId?: string }` – if childId omitted, print all content for that date (all children and group page).
- **Response:** `{ success: bool, message: string }`.
- This might simply set a Firestore entry that triggers printing (as described). So the function could create `users/{uid}/printQueue/{date}` doc with a field `requestedAt`.
- Alternatively, if we had a direct line to printer, the function would need to have network access to printer which it typically doesn't (running in Google Cloud).

- So likely, this function approach would also rely on an external agent. We might skip implementing it as a direct print and instead do the Firestore queue approach which the app can do without a function. So this function might not be necessary if app itself writes to Firestore for print requests.

- **HTTP Function: oauthGoogleCallback**:

- **Endpoint:** e.g. `/api/oauth_callback` (we'll configure Google OAuth redirect to this).
- **Request:** `GET /oauth_callback?state={state}&code={code}`.
- **Response:** An HTML or redirect that maybe closes the window or informs the user linking is done. (If this is initiated from the app, maybe via a WebView or external browser, we might not have a great way to return to app except maybe a custom scheme. We might instead do the OAuth flow via a plugin that handles it. But spec-wise, we can plan a web callback and then require user to go back to app.)

- **Function Logic:** Validate `state` (to ensure request is legit and tied to a user, perhaps store state in Firestore or in a session), use Google API client to exchange `code` for tokens, store refresh token in `users/{uid}/integrationTokens/google` and update `users/{uid}` fields like `googleLinked=true`. Possibly issue some client notification or update so app knows linking succeeded. Perhaps easier: once token is stored, we can set a field in user doc, and the app (which might be polling or just waiting) sees that and continues.

- **OAuth Scopes:** We will request scopes:

  - `https://www.googleapis.com/auth/calendar.events` (to insert events on the user's calendar; this allows read/write of calendar events).
  - `https://www.googleapis.com/auth/drive.file` (to allow creating and managing files in the user's Drive – this scope restricts access to only files created by the app, which is safer than full drive access).
  - `https://www.googleapis.com/auth/spreadsheets` (for Google Sheets read/write; if we use drive.file and the sheet is created by app, that might suffice, but to update arbitrary sheets or read user-provided sheet, this broader scope is needed).
  - We will include `access_type=offline` and `prompt=consent` in the OAuth URL to ensure a refresh token is returned [7]. The refresh token is necessary to perform daily updates without user present (since our app runs daily in the early morning, it needs offline access to Google APIs).

- **Token storage:** Save `refresh_token` (and maybe the first `access_token` and expiry). The function won't keep client secret or tokens in response; it will just persist and maybe redirect the user to a success page.

2.6 Google API Usage Details: Once OAuth is set up, actual usage: - *Calendar:* On each content generation, if calendar integration is enabled, we will create events for that day's activities. We can decide on a scheme, e.g., create two events: - "Operation Summer – Project: [project title or short description]" at a specific time (maybe morning or a preset time the user sets, e.g., 10 AM). - "Operation Summer – Physical Activity: [exercise name]" at another time (e.g., afternoon). - Alternatively, just one all-day event or a reminder like "Operation Summer activities for today are ready" with details in description. But scheduling times could help structure the day. - We'll retrieve the user's primary calendar ID (usually "primary") via Calendar API and insert events using the Events.insert endpoint. Include link to the song or any relevant info in the description. The event times could default or be configurable in app (maybe user can set preferred times for project and exercise in settings). - *Drive:* After generating content, if enabled, we upload the PDFs and images to Drive. We would: - Use Drive API to create a folder "Operation Summer" in the user's My Drive

(once, store the folder ID in the integration token doc or user doc). - Each day, create a subfolder for the date or just upload files with date in name into that folder. Since Storage already holds them, we have two options: * The n8n workflow could directly upload to Drive (with an OAuth credential). But n8n doesn't have the refresh token; only our Firebase does. We could send the content to Drive via a Cloud Function because the Cloud Function can use the stored refresh token to get an access token and call Drive. However, Cloud Functions are not triggered by the n8n content generation by default. We can create a Firestore trigger function: when dailyContent doc is created, if `driveEnabled=true` for user, then function reads the file from Storage (download or directly transfer if possible) and uploads to Drive. * Alternatively, at the end of n8n workflow, call a webhook to a Cloud Function to do the Drive upload. Or incorporate an HTTP request in n8n with the Drive API and an access token. But getting an access token in n8n would require pulling the refresh token from somewhere. We could store the Google refresh token also as a secret in n8n or retrieve via Firestore (n8n can call Firestore REST to get it). This is possible since we trust n8n environment. To reduce complexity, we might leverage Cloud Functions for Google interactions because they have easier access to Google client libs and our stored tokens. - The specifics can be decided during implementation; the spec will note both approaches. For now, assume we'll handle Google API calls via Cloud Functions triggered by events (like new content ready). - *Sheets:* If enabled, the system could append a row to a Google Sheet with columns like Date, Theme, Project Idea, Exercise, Song, Child1 WorksheetTopic (if we generate a specific topic), Child2..., etc., or maybe number of pages printed or any metrics. This could be for the parent's record or to share with, say, a teacher. Implementation similar to Drive: either directly in n8n using an HTTP request to Google Sheets API or via a cloud function. - If doing via Cloud Function, we'd have a function that triggers on new dailyContent and uses the Google Sheets API (or Google's Node.js client) to append a row to a specific spreadsheet (the sheet ID could be stored in user's integration settings once created). - We could also allow the user to provide a Google Sheet ID in settings (like if they have a template or existing sheet to use). - If the user did not provide one, the app could create a new spreadsheet via the Sheets API or Drive API when integration is first turned on.

2.7 Content Moderation & Quality: We will incorporate measures to ensure the AI-generated content is appropriate: - Use OpenAI's moderation API to screen the GPT outputs (script, worksheet content, etc.) for disallowed content. If something is flagged (which is unlikely if prompts are well-crafted for kids, but just in case of a hallucination), we can either retry or replace with a safe fallback (perhaps have some pre-curated content as backup). - Ensure DALL·E prompts are phrased to avoid anything inappropriate (we'll explicitly request "child-friendly" style). - If DALL·E fails due to content or unavailability, we can fall back to a static image or a stable diffusion model if available (though that's additional complexity, maybe not needed if DALL·E is stable). - For text variety, we use **GPT-4** for higher quality narrative and reasoning. In case GPT-4 API quota is hit or response is too slow, fall back to **GPT-3.5**. We can implement this in the n8n workflow: e.g., a try-catch around GPT-4 node, on error call GPT-3.5 node with adjusted prompt. Or use the OpenAI node's parameters to specify model accordingly. - TTS fallback: If using an external TTS (like ElevenLabs which has very natural voices), ensure we have a fallback such as using Google Cloud TTS (which is slightly more robotic but reliable). If one fails, try the other. Alternatively, because Creatomate can handle voice via ElevenLabs integration natively [1], if that fails we could try to pre-generate an MP3 using Google TTS and then feed that into Creatomate (they allow uploading your own audio as well, via a URL).

3. Automation Workflow (n8n) – Daily Content Generation

The n8n workflow is the heart of content generation and assembly. We will build it as follows:

3.1 Workflow Trigger: Use n8n's built-in scheduling trigger (Cron node) to execute every day at a specified time (e.g., 5:00 AM local time). This ensures content is ready by morning. The schedule can be configured; for now, assume daily.

Alternatively, we could use an Inject Webhook to allow manual trigger, but the daily cron is primary. (We will also set up a mechanism to trigger via API for manual runs as described earlier, possibly by having n8n also expose a webhook that the Cloud Function or app can call.)

3.2 Data Fetch / Prep: The workflow's first step could be to fetch the list of active children and any relevant preferences or theme configurations from Firestore. How to do this: - n8n doesn't have a native Firestore node at present, but we can use the HTTP Request node to call Firestore's REST API (with a service account) to get the data. Alternatively, include a code step that uses Firebase Admin SDK to query children. - Another approach: we can store the children info in n8n static data or an external JSON. But better to fetch live so it's up to date if user added a child. - We will likely integrate a service account JSON key into n8n (set as an environment or credential) and use the Google APIs directly in code. Or use the Firestore REST API (which requires an OAuth2 token or using the service account JWT to get one). - Simpler: since n8n is often self-hosted, including Firebase Admin Node.js SDK might be tricky. We can use the HTTP node to call a Cloud Function that returns children list. However, let's plan to use Firestore REST: - We have a service account JSON (we can store it in n8n). Use a Code node to generate a Google OAuth2 token for Firestore or use the service account's JWT to call the Firestore REST endpoint. This is a bit complex; another way is to use Firestore's new REST (which might allow using the service account key as Bearer). - Alternatively, skip Firestore: The n8n could instead be triggered separately per user. But we have potentially multiple users. If we have to generate for all users, the n8n workflow could loop through all user records. - If number of users is small (like one family or a few), it's fine to just do a loop. If it grows, we might refine. - For now, design n8n to handle multiple users sequentially: It could query a Firestore collection of users (with active subscription) and for each generate content.

Given scope, we'll assume one user or handle a few in loop: - Query `users` collection for all users who want daily content (maybe filter by a field e.g., `active=true`). - For each user, query `children` subcollection for child details. - Possibly also fetch any theme preferences or past theme to avoid repetition (maybe we have a list of used themes in Firestore or we can derive from past content). - Then for each user, branch into content generation steps.

3.3 GPT-4 Content Generation: Using the OpenAI node in n8n, we will prompt GPT-4 to produce the textual content. We have to decide on prompt strategy: - One comprehensive prompt vs multiple specialized prompts: - *Option 1:* One prompt that asks for all required outputs in a structured format (script, worksheet questions, project, exercise, song). This ensures a consistent theme across all pieces. We'd need to parse the response. - *Option 2:* Separate prompts for each type of content, possibly with a shared theme or context. For example: prompt GPT for a theme and summary first, then feed that theme into other prompts for worksheet, etc.

Option 1 might look like: `"You are a creative educational AI generating content for a children's summer program. Today, create a theme and the following items for children: 1) A short friendly greeting script for a video (include the child's name <Name> and cheerful tone, about 3-4 sentences). 2) A single image description for a coloring page (one sentence, outline style). 3) A worksheet with age-appropriate questions or problems (e.g., if age 7, a simple math or reading task, provide 3-5`

`questions). 4) A fun group craft project idea (one or two sentences). 5) A physical activity challenge (one sentence). 6) A children's song related to the theme (song title and artist or describe, preferably something findable on YouTube). Ensure all items revolve around a common theme. Respond in JSON format with keys: script, coloringDescription, worksheet, project, exercise, song."` - This could yield a JSON or at least a structured answer. Using GPT-4, it will likely comply with format if asked. - Then n8n can parse the JSON (maybe using a Code node or an JSON parse node if text).

Option 2 might be: - Prompt 1: "Suggest a fun educational theme for today for kids aged [X...] and provide a title for it." -> yields e.g., "Space Adventures". - Then use that theme in subsequent prompts: * Prompt script: "Write a short video introduction script for a program called Operation Summer for a child named [Name] (age Y) on the theme '[Theme]'. Make it cheerful and encouraging." -> gets a script. * Prompt coloring: "Give a one-sentence description of a coloring book page image fitting the theme '[Theme]' that a child [Name] age Y would enjoy, in the style of a simple line art drawing." (We might actually feed this directly to DALL·E instead of GPT). * Prompt worksheet: "Create 5 simple [math/word/etc depending on age] questions for a worksheet for a [Y]-year-old child about the theme '[Theme]'. Format as a list." * Prompt project: "Suggest a group craft project related to '[Theme]' that a group of kids can do together in one afternoon." * Prompt exercise: "Suggest a fun physical activity or game related to '[Theme]'." * Prompt song: "Suggest a children's song (and artist if possible) related to '[Theme]'. Preferably something that can be found on YouTube." - This yields multiple API calls though, which might be slower and more cost. GPT-4 can handle multi-output, so Option 1 is appealing to reduce calls and keep coherence.

We'll opt for **Option 1** (single call returning structured response). If formatting is an issue, we could do a slight combination: ask for all text in one go but not necessarily JSON, then use prompt engineering to reliably split. But JSON is clean if GPT-4 follows it. We must be prepared to handle if GPT output isn't perfectly formatted JSON (maybe use a regex or re-ask).

In n8n: - Use **OpenAI Chat** node (with model GPT-4) with the system and user prompts defined as above. Possibly set temperature moderately (like 0.7) to get creative but relevant output. - If fails or times out, on error route: use another OpenAI node with model GPT-3.5 (with same prompt or simplified). - Once we have the text output, use a **Function** or **Set** node to parse out the pieces. If JSON, we can use JSON.parse in a Code node. If not JSON, maybe instruct GPT to separate with easily split tokens or use multiple OpenAI nodes with roles. But likely JSON is fine.

We will then have variables: `scriptText`, `coloringDescription`, `worksheetText`, `projectText`, `exerciseText`, `songSuggestion` (the last might be "Song Title - Artist").

3.4 DALL·E Image Generation: For the coloring page, we need an image prompt. We can either use the description from GPT or craft our own prompt incorporating child's name: - We want a black-and-white line drawing style (coloring page). DALL·E 3 is quite good with prompts. We might use something like: `"a coloring book page outline drawing of {description from GPT}. No colors, only black outline, for kids to color. Include the child's name '{Name}' in the image."` However, as noted, DALL·E often struggles with rendering text accurately. It might produce gibberish or stylized text for the name. Relying on it to put the name may not yield perfect results. Instead: - We can generate the image without the name and later overlay the name text ourselves on the PDF print. - So prompt more reliably: `"a coloring book style line art outline drawing of [subject related to theme]. black and white, no shading, simple, for a children's coloring page."`

Possibly include "with the word [Name] drawn nicely at the top" – we can try but not guaranteed. Better to not rely on it. - We will have the `coloringDescription` from GPT (e.g., "a rocket ship blasting off"). If it's short and good, use it. If GPT gave a full sentence, might strip to keywords. - Use **OpenAI Image Generation** node (DALL·E API) via n8n's OpenAI node (it supports image generation operations [8]). Provide the prompt. Set desired size (maybe 1024x1024 for quality, or 512 if concerned about token cost—images cost ~ was it priced per image? anyway, we want decent resolution for printing on A4: 1024 or even 2048 if allowed, but DALL·E likely max 1024 or maybe 1024 is max). - This returns an image URL (hosted on OpenAI servers). We will need to download this image to then upload to Firebase Storage (and possibly to embed in PDF). - n8n can do HTTP GET to fetch the binary, then an AWS S3 or Firebase call to upload. Possibly easier: we can directly use a Firebase Storage API to upload from URL (some libraries allow pulling from URL). - But likely simplest: in n8n use HTTP Request node: GET image -> gets binary data -> then another HTTP or code to upload: - We might leverage Firebase Storage's REST API: There's a **REST endpoint** to upload files (via a POST with the file content and a Firebase Auth or storage token). If our n8n has a service account, it can use Google OAuth2 or a simple approach: Firebase Storage allows uploads via an authenticated URL with an `Authorization: Bearer <ID-token or Admin token>`. We could generate a custom token via service account to act as admin. Alternatively, use the Google Cloud Storage JSON API with service account JWT. - Another approach: use Google Cloud Storage Node library in a code node. If we include Google Cloud SDK in n8n environment (with proper creds), we can do `bucket.uploadFromBytes` etc. - Considering complexity, perhaps easier: use a Cloud Function to do the upload after n8n posts the image to it. But that's more moving parts. - We'll lean toward doing it directly in n8n with a small custom code using a pre-signed URL. Actually, Firebase Storage can issue a short-lived upload URL via their REST by calling a `getSignedURL` but that's usually server side Node. Alternatively, skip fancy: we can encode the image to base64 and send to Firestore as part of a doc – not good for large images. - Let's assume we will integrate the Firebase Admin SDK in the n8n environment for straightforward uploading. We'll detail that in integration section.

- Summarizing: DALL·E node -> returns image URL -> HTTP node GET -> Code node to upload to Storage -> produce `coloringUrl` or path.

3.5 Worksheet PDF Generation: The GPT output for the worksheet (likely a list of questions or a short text) needs to be converted into a nicely printable PDF. Approaches: - Use a third-party PDF generation API as cited (like **CraftMyPDF**, **PDFMonkey**, etc.). For example, CraftMyPDF has an API and n8n might have an integration node; similarly, PDFMonkey can be triggered via HTTP. These allow designing a template PDF with placeholders. - Or dynamically generate PDF via a script: e.g., n8n Code node with a library like PDFKit. But adding dependencies to n8n's Code node is limited unless pre-installed in environment. - Another approach: use Google Docs or Slides API: Create a Google Doc from a template with placeholders for questions and export as PDF. But that's heavy and requires Google API usage from n8n or function. - Simpler (since we only need basic text): use **HTML and headless browser**: n8n might allow an HTML to PDF conversion (maybe via a headless Chrome or an integration like an HTML to PDF API). - If using an API like CraftMyPDF, we'd need to design a template in their system (perhaps one that takes a list of questions and child name) and call their API with JSON (this costs but they have some free tier). - Alternatively, for simplicity in spec: treat the worksheet as plain text PDF: - We can create a very basic PDF: Title at top (like "Worksheet for [Name], Age X, Theme: [Theme]"), then the list of questions or tasks. - Could even just take the GPT text and save as a `.txt` file and then in printing, print that text? But nicer to have PDF for consistency. - Let's say we will incorporate a Node.js PDF library on the n8n host. If that's not trivial, fallback could be to call a small Cloud Function that uses a library to generate PDF from text (since Cloud Functions can have dependencies like PDFKit). - Actually, Cloud Function might be easier: pass the text and child name

to a callable function which returns a PDF file or stores it. But Cloud Functions have memory/time limits, though generating one PDF is fine.

Given this is a spec, we can list both possibilities (external API vs custom generation) and lean on one: We will attempt to use **pdfkit** via a small custom script in n8n. Possibly, one can use a JS code node with a base64 library to create a PDF. Actually, to use pdfkit, it must be installed. n8n's Code node by default doesn't allow requiring arbitrary modules unless configured (`NODE_FUNCTION_ALLOW_EXTERNAL`). We could configure n8n to allow pdfkit (just like the Dymo example for printing allowed external modules [9]). If n8n is running in Docker, we might bake in pdfkit.

Alternatively, consider using Google Drive as a creative workaround: Create a Google Doc using the Drive API with the worksheet content, then export as PDF. But that's a lot of overhead for each day.

Given time, using an external specialized API might be easiest from integration standpoint. **CraftMyPDF** (or PDFMonkey) would require account and designing a template. Possibly overkill for one page of text.

So plan: implement either in code or with a very simple API: - There is a service **PDFGeneratorAPI** or **Documint** too, but let's keep spec focusing on code.

Approach: Use n8n **Function** node with **pdfkit**: - Prepare the worksheet text (the GPT output might already be formatted somewhat). - In code node, use something like:

```
const PDFDocument = require('pdfkit');
const doc = new PDFDocument({ margin: 50 });
doc.fontSize(18).text(`Worksheet for ${childName} (Age ${age}) - Theme: ${theme}
`, { align: 'center' });
doc.moveDown();
doc.fontSize(12).text(worksheetText);
const buffers = [];
doc.on('data', buffers.push.bind(buffers));
doc.on('end', () => {
  const pdfData = Buffer.concat(buffers);
  // save pdfData to binary output
  return pdfData.toString('base64');
});
doc.end();
```

Something akin to that. Actually, n8n code node returning binary might need special handling (maybe return as base64 and use a subsequent node to convert). - Alternatively, simply output base64 and then use an n8n binary convert node to actual file. - Save the PDF file similarly to Storage (or first to local then upload).

If we can't do that in n8n easily, as backup mention: - Possibly send `worksheetText` to an **HTTP node** calling a small self-hosted script or Cloud Function that returns PDF. - But let's assume the above works with n8n configuration (the Medium article we saw shows that for Dymo they allowed an external npm, similarly we can allow pdfkit via environment and require it [9]).

3.6 Assembling Group Activities Page: To print the project, exercise, and song, we might want to put them on a single summary page (so that the printer yields a page containing these group instructions). - This can be done similar to the worksheet PDF. It's mostly text (project idea could be a short paragraph, same for exercise, and a line for song). - Format: Title like "Today's Group Activities - [Date] [Theme]", then "Project: ..." paragraph, "Exercise: ...", "Song: ..." (maybe include the song link as text or a QR code if fancy). - Including a QR code for the YouTube link could be a good idea so that someone can scan it from the paper. That would need a QR code generator. We could use Google Chart API (construct a URL for QR) or an npm like `qrcode` in our PDF generation. If too much, just print the URL (which might be long, but maybe provide the video ID or a short link). Possibly better: if the song is well-known, just printing title might be enough to search manually. - But adding a QR code in PDF is doable (pdfkit can place an image if we generate one). - For simplicity, we might skip QR for now or mention as enhancement.

So, we will similarly generate a PDF for group activities.

Alternatively, we could combine the worksheet and group text onto one PDF to minimize separate prints – but since each child gets their own worksheet, group tasks are separate anyway. So we'll end up with: - X worksheets (for X children) – each personalized. - X coloring pages – each personalized (though coloring page itself not personalized beyond name label). - 1 group activities page – common.

So total PDF pages = 2*X + 1. If printing one by one, that's fine, or if needed we could also collate them into one big PDF to send as one print job. But that might collate differently per child. Probably fine as separate jobs.

3.7 Creatomate Video Assembly: With script text and a generated voice audio, we create the intro video. - If using Creatomate's direct integration with ElevenLabs (for voice) and OpenAI (for images), we can simplify by just providing text and optional an image URL to the Creatomate API. According to documentation, we can design a template with placeholders: - One placeholder for a voiceover (which uses ElevenLabs voice, we specify text). - Possibly a placeholder for a subtitle text (auto-generated from voice). - A placeholder for an image or background. - The child's name could be a text element in template or even spoken in the script (the script will include the name). - We can have multiple scenes or just one scene with a static background and the voiceover. - For example, a template could have: * A background video or color. * A text element that can show the script text (if we want text on screen). * A voiceover element (ElevenLabs) which speaks the script. * Some simple animation or music could be pre-added in template for polish (optional). - We will obtain a `template_id` after creating on Creatomate platform. - **Creatomate API call:** The n8n workflow will include an HTTP Request node: - URL: `https://api.creatomate.com/v1/renders` - Method: POST - Headers: Authorization with Bearer API Key (the API key from our Creatomate account, stored securely). - JSON Body:

```
{
  "template_id": "YOUR_TEMPLATE_ID",
  "modifications": {
      "Voiceover-1": "<script text>",
      "Image-1": "<optional image URL if template has an image placeholder>" ,
      "Text-1": "<child name or other text placeholders if any>"
```

```
  }
}
```

(The exact keys like Voiceover-1, Image-1, etc., correspond to element names in the template). - Creatomate will handle generating the video with ElevenLabs voice (if the template's voiceover element is configured for that). Their platform notes that it integrates seamlessly with ElevenLabs and can auto-generate the voiceover and even subtitles [1] . - The response from Creatomate could be immediate with a render ID and then we might need to poll until the video is ready, or it might return a final video URL if quick. According to examples, it seems one can either wait or get a callback. Simpler: do a synchronous wait for completion by polling in n8n (maybe an HTTP node to check status if needed). - Once ready, it provides a URL to the video file or the file itself. Likely a URL to an AWS S3 or something. - We then need to download that video file (HTTP GET) and upload to Firebase Storage (like we did for images). The video might be a few MB (depending on length, but script is short ~ 30 seconds maybe, video likely <5 MB). - If for some reason we don't want to use Creatomate, alternative would be to generate the audio separately and then combine it with an image to a video ourselves (for example using FFmpeg on the server). But that's complex to do in n8n environment, so using Creatomate as intended is best.

- **Video content considerations:** The script is likely a greeting like "Good morning, [Name]! Today we have a fun day about [Theme]. You'll get to do some cool things, like [project mention] and a [exercise]. Have fun learning!". So maybe 20 seconds.
- We might not have a unique visuals for theme (unless we wanted to include the DALL·E image or another AI image illustrating the theme). We could consider generating one more image via DALL·E: e.g., a colored illustration of the theme to use in video background. However, that's an extra step and our DALL·E already gave coloring page (which is b&w, not great for video).
- Perhaps skip: just use a static background or a simple animation from template.
- Or find a stock image relevant to theme using an API? Too much. A compromise: the video can just be text-on-solid background with voice. It's basic but serves the purpose of a personalized greeting.
- We can iterate later to improve visuals.

3.8 Integration of Workflow with Firebase: After generating all pieces (script, images, PDFs, video), we will: - Upload each file to Firebase Storage as planned. - Create/update Firestore documents: - Write the group dailyContent doc with theme, project text, exercise text, song info. - Write each child's dailyContent doc with storage paths or download URLs of their assets. Also possibly store the raw text of worksheet (in case we want to display it in app) or the script text (not really needed in app), but maybe store it for reference. - Possibly mark a flag like `status="generated"` and timestamp. - The Flutter app listening will get these updates and present to user. Also a Cloud Function might trigger on this Firestore write to send notification or do Google Calendar/Drive/Sheets integration: - Alternatively, n8n at the end could call Google APIs too (we discussed that earlier). But dividing responsibilities, it might be cleaner that: * n8n does content generation and Firebase upload. * A **Cloud Function trigger** on `dailyContent/{date}` document creation does the Google Calendar events and Google Sheets log. (It has easy access to the data now in Firestore and to the Google refresh token from Firestore as well.) * For Google Drive, we might prefer to do it in Cloud Function as well (maybe the same one or another) because it's similar context. That means after n8n stores content in Storage, the function can fetch it and push to Drive. * This way n8n doesn't need Google credentials at all. * However, triggers might need careful control to not run multiple times or on partial data. We might want to ensure Firestore doc is fully written. Perhaps n8n can write the group doc last as a signal (so the function triggers when group doc arrives, then it knows the children docs and files exist). * Or use a field like `complete=true` to indicate generation done. * Alternatively, call a

callable function at end of n8n to handle Google integration (explicit call rather than trigger). * For spec completeness: we can mention using Firestore triggers for automation to avoid tight coupling with n8n.

3.9 Error Handling & Retry: The n8n workflow should handle errors gracefully: - If GPT call fails (due to rate limit or network), catch error and attempt a retry after a short delay, or fallback to simpler model. - If DALL·E fails (returns no image or an error), retry with a tweaked prompt (maybe remove the child name, or use a simpler description). If still fails (rare unless content policy), we could use a placeholder image (maybe a generic coloring page from a set). - If TTS fails (if using external), try an alternative TTS API. - If Creatomate fails to render (maybe due to API error), we could try again or as fallback skip the video and just have the text available. - Each of these should not block the others indefinitely – design workflow to continue even if one part fails, so that at least partial content is delivered: - For example, if video fails, still produce PDFs and mark video as unavailable for that day (app can show "video not available"). - Use branching logic in n8n for each API call success/failure. - Logging: We can add nodes to log errors to a Firestore `logs` collection or send an email to admin if something goes wrong (since for production readiness, we want to know if daily job fails).

3.10 Testing the Workflow: Before production, test the n8n workflow with a sample user and child. Use a fixed date or on-demand trigger to observe that all pieces are generated correctly and files appear in Firebase. We might use n8n's workflow testing and also unit-test parts by substituting GPT with sample output.

4. Google API Integration (OAuth and Usage)

Integration with Google services requires careful OAuth2 implementation and token management:

4.1 OAuth2 Consent & Setup: We will register the application in Google Cloud Console to obtain a Client ID and Client Secret for OAuth. Since our platform is Firebase-based, we have a few options: - Use Firebase Auth's built-in Google sign-in for basic profile info (this we do for login perhaps), *but* for accessing Google Calendar/Drive/Sheets, we need additional scopes which the Firebase Auth token won't cover by default. - The recommended approach is to perform a separate OAuth flow for the additional scopes: - Possibly utilize the `google_sign_in` Flutter plugin which can accept scopes. On mobile, it can handle OAuth sign-in flows. On web, Firebase Auth can be invoked with scopes as well by using the OAuth provider. Actually, Firebase Auth's Google provider can request additional scopes by setting `customParameters` before signIn. This way, the sign-in flow will ask for extra permissions (like Calendar access) and Firebase Auth might then allow retrieving the OAuth tokens (Firebase Auth can give an OAuth access token for Google if scopes were requested, but not 100% certain if refresh token is retrievable via Firebase Auth). - Alternatively, handle it outside Firebase Auth: open a WebView or external browser to Google's consent screen (with our own client ID) and then catch the redirect in a deep link. - Simpler: since the user likely will sign in with Google anyway (it's a common case), we can piggyback on that to get offline access. Firebase Auth does allow linking with Google credentials; however, obtaining the refresh token for use outside Firebase might be tricky. We might end up with only an access token (short-lived) that Firebase provides. - The robust method: run a separate OAuth flow through a Cloud Function as we described: user clicks "Connect Google", Flutter opens `https://accounts.google.com/o/oauth2/v2/auth?...&scope=...&access_type=offline&state=<uid>` in a browser (maybe using `url_launcher` plugin). After user consents, Google calls our function URL. The function then ties it to the user and stores token. The app might poll or we can have updated state in Firestore.

We choose to implement the **Cloud Function OAuth callback method** to avoid complexity on Flutter side. So: - In Flutter, user taps “Link Google Account”, we open a URL like:

```
https://accounts.google.com/o/oauth2/v2/auth?
   client_id=XYZ.apps.googleusercontent.com&
   redirect_uri=https://<our cloud function domain>/oauthCallback&
   response_type=code&

scope=https%3A%2F%2Fwww.googleapis.com%2Fauth%2Fcalendar.events+https%3A%2F%2Fwww.googleapis.com%
   state=<uid>|<session_token>&
   access_type=offline&
   prompt=consent
```

The `state` will include the Firebase UID of user (and a random token for CSRF). We store that token in Firestore or in memory to verify. - Google shows consent for Calendar, Drive, Sheets. - After allow, Google calls our function with `code` and `state`. - The function verifies state (matches user), uses Google’s OAuth2 token endpoint to exchange code for tokens. It receives `access_token`, `refresh_token`, `expires_in`, etc. - The function stores: - `refresh_token` securely for that user, - perhaps the `access_token` and expiry if we want to use immediately (e.g., we might want to make an immediate test call like get the user’s calendar list to verify or create the “Operation Summer” calendar events or folder). - We then update user’s doc: e.g., set `googleLinked=true, googleEmail=...` (we can actually call Google’s People API or decode the ID token to get email). - Or if the user used the same Google for login, we already have email and can double confirm. - The function can then either redirect the browser to a static "Linked successfully, you can close this window" page. For web, we can provide a page via Hosting or directly respond with a small HTML. - The Flutter app side: we might have set up a listener or we can poll Firestore for `googleLinked` field true. Or if using a dynamic link, we could catch the redirect in-app. But simplest: tell user to return to app manually and we show status updated. - Now that we have a refresh token, subsequent calls: - For Calendar events: The daily content trigger function will use Google’s Node.js API client with the refresh token to create events. (The refresh token can be used to get a new access token whenever needed; Google libraries do this automatically.) - For Drive/Sheets: same process, use refresh token. - We must ensure to specify `token.access_type=offline` so refresh token is not one-time. Google’s refresh tokens typically don’t expire unless user revokes or if a new one is granted (there’s a limit like 50 per client, but our app will use one per user).

4.2 Calendar Integration Implementation: - Cloud Function on new content (or integrated in n8n) will: - Compute times for events (maybe read from user settings or default schedule). - Use Calendar API (authenticated with user’s token) to insert events. If needed, first find or create a specific calendar? We could just use primary calendar. - Possibly mark events with a specific color or title prefix "Operation Summer: ...". - Add details: For craft project event, description could mention the project details. For exercise event, description with exercise details. Song maybe not an event but could include in one of descriptions or as separate "Music break" event. - If the user doesn’t want calendar events, they can disable in settings (we only do if `calendarEnabled=true`). - We handle errors like token expiration (in which case refresh and retry), or if token is revoked (the API will return 401; in that case, we should set `googleLinked=false` and notify user to re-link).

4.3 Drive Integration Implementation: - If enabled, we ensure an "Operation Summer" folder exists: - possibly create one named "Operation Summer [Year]" or just "Operation Summer". Use Drive API to create folder (if not exists). - Store that folder ID in Firestore (so we don't create duplicates). - Upload files: * Use Drive API's files.create with uploadType=multipart or media upload. Provide the file content (which we have in Firebase Storage - we could either download it and then upload to Drive, or use the public URL if the file is accessible. But our files are protected. So likely need to download). * The Cloud Function can use Firebase Admin to get the file from Storage (since it's in same Google Cloud project, we can use Admin SDK to get file as buffer). * Then use Drive API to upload that buffer. * Or a trick: we could add the service account as having access to the file in Storage and then use Drive's copy from URL if that exists (but Drive can't directly pull from a URL without us providing content). * So yes, likely download then upload. - Upload the group PDF, each child's worksheet PDF, and each coloring image. Possibly store them in subfolder by date for neatness: * Create a date-named subfolder in Drive folder and put files inside. * Or prefix filenames with date and child name and just put all in one folder. E.g., "2025-07-15 - Worksheet - Alice.pdf". * A subfolder per date could clutter if 90 days = 90 subfolders, but it's fine. Alternatively, one folder for all content might become large but manageable. We can do either; subfolder might help the user browse per day. - Alternatively, only upload a combined PDF if we produce one with all materials for the day. But since we keep them separate for printing, we'll upload separate files for fidelity.

4.4 Sheets Integration Implementation: - If enabled, choose or create a Google Sheet for logs: - Perhaps create a spreadsheet "Operation Summer Log" with a sheet named "Daily Content". - Columns: Date, Theme, Project, Exercise, Song, Child1 Worksheet Topic or Score (if one day we track results), etc. Could be simple now: just record what was given. - Each day append one row with the data. (If multiple children, either list them in one cell or have separate columns per child). - Possibly record whether printed. - The purpose is a journal of activities; maybe useful for the parent to remember what was done or share with others. - Implementation: Cloud Function uses Sheets API (appends row). We need the Spreadsheet ID stored. We can create the spreadsheet on first link: use Drive API to create a Google Sheet file, get ID, store in user's doc or integration doc. - Then each day use Sheets API `AppendValues` to add the row.

4.5 Token Refresh Strategy: - We rely on the refresh token to get new access tokens whenever needed. Google's Node libraries manage this automatically (the first call with a refresh token will fetch a fresh access token behind the scenes). - If we use raw HTTP requests, we'd need to call Google's OAuth token endpoint with `refresh_token` and our client secret to get a new access token. We can automate that with a small helper in Cloud Function if not using Google's lib. - Ensure to handle token errors: If refresh token is revoked or expired (rarely Google refresh tokens can expire if not used for ~6 months or something), we should catch that error and mark integration invalid so user can re-auth. - Keep the refresh token securely: Firestore but readable by Cloud Function is okay. Alternatively, store in Google Cloud Secret Manager and store reference in Firestore – might be overkill. - Since only our project's backend uses it, Firestore is fine. Just ensure rules so that user's client cannot read the refresh token field (we can do a rule that disallows reads on `integrationTokens` or such even to the user, since app doesn't need it, only functions do).

- If using Firebase Auth's OAuth linking instead of our own flow (just as an aside, since it's possible to link Google provider with scopes), Firebase can store a refresh token in the user's credential, but retrieving it for use might not be straightforward. So we stick with custom flow.

5. OpenAI Usage: Prompts, Models, and Fallback Logic

We have partially covered prompt design in the n8n workflow section, but here we formalize the usage of OpenAI:

5.1 Model Selection: We will primarily use **GPT-4** (specifically the version available via OpenAI API, e.g., `gpt-4` or `gpt-4-0613` if specifying) for text generation due to its superior ability in generating creative, coherent, and instruction-following content. GPT-4 will generate the structured daily content (script, worksheet content, etc.).

For cost or availability reasons, we implement fallback to **GPT-3.5 Turbo**: - If the GPT-4 API call fails (HTTP error or timeout) or if a certain number of daily requests is exceeded, we catch the error and then call `gpt-3.5-turbo` with the same prompt. - We might also use GPT-3.5 for less critical or less complex generation if needed (for example, if we decided to separate prompts, maybe using GPT-3.5 for straightforward ones like song suggestion to save tokens). - The system should log if fallback happened (for monitoring quality).

5.2 Prompt Structure: The prompt to GPT will be carefully structured to ensure age-appropriate and coherent results. We will use either a single prompt or multiple. As decided, likely a single prompt with a clear format request. We will include in the prompt: - The child's name and age (so the AI knows to tailor language complexity). - A note that this is for educational fun content, to ensure tone is friendly and not too formal. - The list of required items, possibly numbered, and request JSON output for easy parsing. - Example (as in 3.3):

```
System role: "You are an educational assistant that creates fun daily content
for children."
User message: "Generate today's content for Operation Summer for a child named
Alice (age 7).
Choose a creative theme kids would love. Provide:
1) A short greeting script for a welcome video, mentioning the child's name and
introducing the theme.
2) A description of a simple coloring page image related to the theme (one
sentence).
3) An age-appropriate worksheet - 5 questions or exercises related to the theme
(in plain text).
4) A group craft project idea related to the theme (one sentence).
5) A physical activity/game idea related to the theme (one sentence).
6) A children's song related to the theme (provide song name and artist).
Respond in JSON with keys: theme, script, coloring, worksheet, project,
exercise, song."
```

- The assistant (GPT) should output JSON like:

```
{
  "theme": "Space Adventures",
  "script": "Good morning, Alice! Today we're going to explore the wonders of
```

```
space...",
  "coloring": "a rocket ship blasting off surrounded by stars",
  "worksheet":
"1. What planet is known as the Red Planet? ... (and so on with 5 questions)",
  "project": "Make a paper mache solar system with balloons and paint.",
  "exercise": "Do a 'moon jump' competition to see how high you can jump!",
  "song": "\"Twinkle Twinkle Little Star\" - Traditional"
}
```

- We will test and tweak the prompt as needed to get a well-formatted response. If JSON is tricky, we could ask for a delimiter format instead and parse.

5.3 Content Filtering: We use OpenAI's policies to keep content appropriate. By context and instructions, GPT should produce kid-friendly content. As an extra step: - Use the **OpenAI Moderation API** on the combined text output (or at least on the script and worksheet which are longer) to ensure nothing harmful. n8n has an OpenAI node operation "Classify Text for Violations" [10] which can flag issues. If flagged, we will not deliver that content. We could attempt a second generation with a stricter prompt (like explicitly instruct "avoid any sensitive topics"). - It's expected to be rare to get disallowed content given the domain, but we plan for it.

5.4 DALL·E Prompt: We will create prompts for DALL·E that yield coloring pages: - Format: `"Black and white coloring book page, line art drawing of [subject]."` If we have the theme and maybe the coloring description from GPT, we combine them: e.g., theme "Space Adventures" and GPT gave "a rocket ship blasting off surrounded by stars", our prompt: `"Black and white coloring book outline of a rocket ship blasting off into space with stars around. No shading, simple lines, for kids to color."` - We may omit the child's name in the prompt to DALL·E to avoid weird text; we will handle the name ourselves in print. - We use the `n8n OpenAI -> Generate Image` node with size 1024x1024. The node will return an image URL (which is accessible via internet for a short time). - If needed, we can specify style like "in a cartoon style" or "in a cute style" but as it's a coloring page, line art is enough.

5.5 TTS (Text-to-Speech): We have a few options: ElevenLabs API, Google Cloud TTS, Amazon Polly, or even Azure TTS. The mention in Creatomate suggests using **ElevenLabs** (they integrate it easily). We will likely use ElevenLabs for high-quality voices: - If using Creatomate's integration, we just pass text and it uses presumably a default or chosen voice we set in template (maybe we can pick a child-friendly voice). - If not using Creatomate for TTS, we could call ElevenLabs API ourselves in n8n, which would return an audio file (MP3). - In that case, we'd then supply that audio to Creatomate as a URL (maybe by uploading audio to a temporary location). - But since Creatomate can do it, easier to let them handle TTS if possible. - Alternatively, Google Cloud TTS is easy to call and cheaper. But voices may sound less natural. For a "friendly narrator" vibe, ElevenLabs or Azure neural voices are better. - Because we have to possibly pay for ElevenLabs as well, in early stage we might use Google TTS which might be covered if small usage. However, the spec will assume ElevenLabs for quality. - We will include in the documentation that TTS can be swapped or configured.

5.6 OpenAI API Keys and Costs: The OpenAI API key (for GPT and DALL·E) will be stored securely on the server (as n8n credential). We must ensure not to expose it to client or in Firestore. n8n credentials are encrypted typically. Similarly, the Creatomate API key and ElevenLabs key are in n8n credentials. - Monitor

usage: We can implement some usage tracking – like log how many tokens or images per day to gauge costs, possibly storing a count in Firestore or using OpenAI's usage info. - In case of reaching limits, perhaps scale down (e.g., only use GPT-4 for certain length or use 3.5 temporarily).

5.7 Fallback Summary: - GPT-4 -> fallback to GPT-3.5 for text. - DALL·E -> fallback to stable image or second attempt (perhaps try a simpler prompt or use a smaller size). - ElevenLabs -> fallback to Google TTS if needed. - Creatomate -> fallback to simpler method: we could at least send the TTS audio and maybe the app could show a text if video fails. But realistically, if video fails, user still has other content; video is nice-to-have. We might send a notification "Video generation failed today" to address it.

6. Video Generation Automation (Creatomate Integration)

6.1 Creatomate Template Design: We will create a template using Creatomate's online video editor: - **Duration:** ~20-30 seconds. - **Resolution:** likely 1080p or 720p (not specified, but 720p should suffice for quick loading on mobile, 1080p if we want full HD since might watch on bigger screen). - **Elements:** For example: - Background: a color gradient or stock image (maybe something neutral or fun, or even dynamic backgrounds provided by Creatomate). - Text element: Could display the child's name and maybe the theme or a title ("Good Morning, [Name]!"). But since voiceover will already say it, on-screen text is optional. We could use it to reinforce reading. We could create a text element bound to placeholder "Name" or even dynamic content from the script. - Voiceover: The key element – we will add an **Audio** element configured to use ElevenLabs with a specific voice (like a friendly female/male voice appropriate for kids). This element is linked to a placeholder text (like "VoiceoverText"). In the template, that placeholder can be given the actual script via API. Creatomate then calls ElevenLabs to generate the audio and sync it in the video [11]. - Subtitles: Possibly add an auto-generated subtitle track (Creatomate can generate subtitles from the voiceover [12] [13]). This is a nice touch so kids could read along. We can enable it by linking subtitle element to the voiceover source. - Imagery: Optionally, if we want to show something relevant to theme, we could include 1-2 image placeholders. For example, we could have one static image covering the frame or as a picture-in-picture. We could feed in either: * The coloring page (but that's line art, maybe not visually engaging for a video). * A different AI-generated image (maybe we could use DALL·E to also create a colored illustration of the theme). To avoid another API call, perhaps skip initial version. * Alternatively, use an emoji or icon of the theme. E.g., if theme is space, maybe show an emoji rocket or some small graphic. Harder to automate nicely. * Maybe better: have a generic intro sequence that doesn't rely heavily on dynamic images. Could just have confetti animation or something with the audio. - Music: Possibly add a gentle background music track in template (if voice is clear, a soft background music can make it lively). Creatomate might allow adding a background audio track. Or we skip to keep it simple.

- Once template is created, we note its `template_id`.

6.2 API Use: As outlined, we call Creatomate's render endpoint with: - `template_id` - `modifications`: * The voiceover placeholder gets the script text. (If script is longer than template timeline, template might auto-extend or maybe we designed timeline to fit a certain length by making it responsive to voiceover length.) * If we have a text element for child name or theme, set those too. * If we decide to include an image placeholder and want to use the coloring image or some default: - We could, for example, include the coloring image to show at the end as a "Coming up next" preview. But since coloring is B&W, maybe not so attractive on video. - Another idea: Instead of an AI image, we could incorporate the YouTube song's thumbnail or something, but retrieving that automatically is extra steps (YouTube API to get thumbnail by song name? not guaranteed). - Probably simplest: not to use dynamic image at first. But we can keep a

placeholder for a theme image and either fill it with a relevant stock image manually chosen per theme. We don’t have a library of theme images though. - Perhaps no image or use a static fun illustration (like a sun wearing sunglasses, given "Summer" vibe) on all videos to brand it. That can be baked in template (no need to modify). - **Rendering**: The Creatomate API after called will process. We might have to poll the API. They might provide a `renderId` and an endpoint to check status. The example in docs suggests one call can be blocking until done (maybe not, likely asynchronous with polling). * If asynchronous, n8n might need to periodically check or perhaps Creatomate can send a webhook to an endpoint. However, easiest path: use a loop in n8n: - After POST, use a loop node that checks a GET status endpoint every few seconds until done or a timeout. - Or possibly the initial POST response includes the final video URL if it was quick. The docs show an example where they just do one call and presumably the output video is available at a URL in the response (maybe they wait to return until done, since short videos might generate in a few seconds). * We’ll verify with their docs: They have a synchronous mode; indeed, the code example in [1] was just one call and then “Output Video” was shown – possibly they waited. If the processing takes longer than typical HTTP request, maybe their API holds the connection open. But likely not too long since a short video can be done in maybe 5-10 seconds. * We'll plan to poll using a render ID if needed. (We can find this from their docs if needed, but in spec, just say we will handle retrieving result.) - Once ready, we get either a direct file data or a URL to download (some API return a CDN link). - We then proceed to upload to Firebase Storage (`videos/uid/date_name.mp4`). - Save that link or path in Firestore.

6.3 Video Output Example: A final video might say (audio): "Good morning, Alice! Today we're going on a Space Adventure. You'll get to build your own rocket and do a moon jump challenge. Have fun and let's blast off into learning!" with some upbeat tone. On screen perhaps the text "Space Adventure" and child’s name appears with some animation.

6.4 Testing Video Gen: We will test with a couple of sample scripts manually via Creatomate’s editor to ensure the voice is clear and the timing fits. Adjust template if voiceover runs longer than visuals (maybe allow it to auto-resize or just have a static background so duration doesn’t matter).

6.5 Performance Considerations: Each video generation is an API call to Creatomate which in turn calls ElevenLabs (which can take maybe a second or two to synthesize) and then composes video. For multiple children, these could be done sequentially or in parallel: - If we have, say, 3 children, generating 3 videos might take a bit. Could we parallelize? Possibly with multiple branches in n8n (split by child). But n8n might handle each in sequence anyway unless we explicitly fork. Could do a Function that triggers multiple API calls concurrently, but simpler is sequential given it's at 5am and time is not super critical if it's done by 6am. - It should still be fine (maybe 10-20 seconds each, so a minute or two total). - Meanwhile other text generation is quick. DALL·E might take ~5 seconds per image. GPT maybe a couple seconds. So overall, maybe the pipeline finishes within 1-2 minutes for a few kids. Well within morning schedule.

6.6 Error Handling in Video Gen: If Creatomate returns an error (like template ID wrong, or text too long): - We should ensure script length fits the template’s allowed voice length. If child’s name is long or GPT gave a longer script than usual, maybe it's fine because the voiceover will just lengthen video. - If fails, we can attempt a second try maybe with a simpler template or skip voice (like maybe as a last resort, produce a text-only video or just none). - At least log the error to troubleshoot template issues.

7. Printing Integration Strategy

Printing the daily materials automatically is a unique challenge. We outline a strategy to ensure the PDFs/images are printed each day with minimal user intervention:

7.1 Environment for Printing: The printing will be executed by a **local agent** because cloud services cannot directly reach a LAN printer (and Google Cloud Print is discontinued [2]). We assume: - The **n8n workflow will run on a machine that has printer access**. This could be a spare PC, Mac, or Raspberry Pi on the same network as the printer, or even the printer host machine itself. - Alternatively, we deploy a separate small service on a local machine for printing and have n8n communicate with it.

Given we are already using n8n for generation, the simplest path: run n8n on the local machine where printer is installed. That way, n8n can directly execute print commands.

7.2 Printing via n8n: n8n provides a few ways to run custom actions: - **Execute Command Node:** This node can run shell commands on the host machine. We can use it to call OS printing utilities. E.g.: - On Linux/macOS with CUPS: `lp -d <PrinterName> /path/to/file.pdf` will send the PDF to the printer queue. - On Windows: could call `print` command or use PowerShell `Start-Process -FilePath "file.pdf" -Verb Print`. - There are cross-platform Node packages (like **pdf-to-printer** [6] or **node-ipp** for direct IPP printing) which can be invoked via a script. - **Custom Node with Libraries:** As referenced, the n8n community suggested using a Node package for Dymo printing [9]. Similarly, we could install `pdf-to-printer` which supports Windows and Unix (there's an updated version that works on Unix too [14]). In an n8n Function node, with `NODE_FUNCTION_ALLOW_EXTERNAL=pdf-to-printer`, we can do:

```
const { print } = require('pdf-to-printer');
await print("/tmp/worksheet_alice.pdf", { printer: "MyPrinterName" });
```

This abstracts the OS differences and uses underlying OS drivers. - Another approach: use an HTTP node to call a local print server. We could implement a mini Express server (like in the Medium article example) that exposes an endpoint, and n8n calls it with the file. The article "Super simple PDF printing service with node.js" does exactly that – an Express endpoint that receives PDF bytes and uses pdf-to-printer to print [15]. We might not need this extra layer if we can do directly in n8n, but it's an option for decoupling if needed.

7.3 Automated Trigger of Print: We want printing to happen right after content is generated: - The n8n workflow, after uploading files, can immediately add a step to print them. Because n8n is running locally, it can access the files either by downloading from Firebase or, since it just uploaded them, it has them in memory or at least knows the paths. - Actually, after uploading to Firebase, the files might not necessarily reside on local disk. We might need to download them from the URLs. Alternatively, we could save a local copy during generation and then use that to print (and still upload from that copy). - For example, when generating PDFs via code, we have the PDF buffer; we could write it to a local temp file as well as upload. - When downloading images from DALL·E, we had them in memory to upload; could save to file as well. - Or use the fetched binary directly to print without saving (some print libs might accept buffer). - It might be easier to save each file to a known temp directory (like `~/operation_summer_temp/<date>/childName_worksheet.pdf` etc.) and then call print on those files. After printing, optionally delete them. -

Another angle: some printers support printing via URL or email. For instance, HP ePrint allows sending an email to a printer-specific address. If our user had that, we could email the PDFs. But that's not general and depends on printer. - We stick to direct printing on local.

7.4 Printer Configuration: We will need the printer's name or address: - If the printer is installed on the n8n host OS, the OS will have a printer name (like "HP_OfficeJet" etc.). We can specify that in the print command. Possibly expose this as a config in n8n or environment variable so it's not hard-coded. - If the printer is networked but not installed, we could try direct IP printing via IPP. But it's simpler to just install it on OS. - Document in deployment that the user must ensure the printer is installed on that system and test a manual print from that system first.

7.5 Printing Order: We can print all files one by one: - Perhaps print the group activities page last (so it's on top of stack or bottom? Up to how they want to collate). - Or just any order, user will sort them out. - Could even print each child's set together: first coloring then worksheet for child1 (two pages), then child2, etc., then group page. That might be logical grouping. Or group page first so they see overall then individual sheets. - Minor detail, but we can do child by child: * Print child's coloring page, then worksheet, * then next child's, etc., * then print group page. - Alternatively, if we had combined per child (like one PDF combining coloring+worksheet for that child), we could print that in one go each child. But we kept separate. Not too hard to combine with a PDF library if needed, but separate is fine. - We will incorporate a small delay between print commands if needed to avoid spooling issues, though usually it's fine to send multiple jobs quickly.

7.6 Print Node Implementation in n8n: We will likely use the Execute Command node for simplicity: - Example node config (for Linux): Command: `lp`; Parameters: `-d Printer_Name {{ $json["filePath"] }}` (if we have filePath from previous node). - We will do this in a loop for each file. - If using Windows, the n8n instance might be running under something like Windows Subsystem for Linux or we use a different command. We might assume a Linux environment for now (if using Raspberry Pi or a small Linux server). - We will mention cross-platform considerations: On Windows, the `lp` command might not exist; could install Cygwin or use `print`. - Because detailing cross-OS is beyond spec scope, we can assume Linux environment (common for an always-on service). - If using the Node library approach: - We add a Function node after generation that does require('pdf-to-printer') and calls print for each file. - We must ensure environment variable `NODE_FUNCTION_ALLOW_EXTERNAL=pdf-to-printer` is set for n8n (like in its config). We'll document that in deployment.

7.7 Handling Print Failures: There are many real-world issues that can happen (printer offline, out of paper, etc., which we can't fully solve in software). But we can: - Catch if the command returns an error code and log it or mark in Firestore that printing failed. - If printing fails, maybe keep the files and next time try again or alert user (app can show "Printer error, please check printer"). - Possibly send an email or push notification to user if printing fails so they know to print manually. - Provide the user the ability to reprint via the app (which we have: pressing "Print Again" triggers either a Cloud Function or simply triggers n8n again for that day's files).

7.8 Alternative Approaches: (For completeness) - Use a cloud print service like **PrintNode** or **Google Cloud Print alternatives** [16]. PrintNode, for example, provides an API and you run a local agent that connects to their service. We could integrate that by sending files to PrintNode API, which then prints via the agent. This could simplify our code but introduces an external dependency/cost. We will not use it now, but note that as an alternative if scaling to many printers (like a product given to many customers, we might prefer that

approach so we don't manage each local environment). - Use CUPS API: If the printer is on a CUPS server, one could send an IPP request. There's an NPM `ipp` that could send the job. But since we already have OS spool accessible, not needed.

7.9 Summary of Print Implementation: - The recommended implementation: **n8n on local machine**, after content generation: 1. Use **Function node** to write files to local disk (if not already) – not strictly necessary if we can print from memory, but easier. 2. Use **Execute Command node** for each file: e.g., `lp -d MyPrinter /tmp/operation_summer/2025-07-15_Alice_color.png`. 3. Check exit status; if error, set a flag or output to Firestore log. 4. Optionally, after successful prints, delete temp files to save space. - The app, via Firestore or a function, could get feedback of print status. We might update the Firestore dailyContent doc with a field `printed: true` or `printError: "error message"` accordingly. So the app can display “Printed” or “Needs attention”.

8. Firebase Project Structure & Configuration

We detail how the Firebase project will be set up and organized:

8.1 Firebase Project Setup: Create a Firebase project (e.g., named "operation-summer"). Enable the following services: - **Authentication:** Enable Email/Password and Google as sign-in providers. For Google, since we also need additional scopes, we’ll just use it for identity for now. Ensure to add the app’s OAuth client (if any) or domain to Authorized domains as needed (for web sign-in). - **Cloud Firestore:** Set to production mode (with security rules) and a proper location (choose something like us-central or where appropriate). - **Firebase Storage:** Enable default bucket. - **Cloud Functions:** Initialize with Node.js environment. - **Firebase Hosting:** (for web app and possibly for the OAuth callback landing page if needed). We will host the Flutter web build and possibly some static pages.

8.2 Firestore Collections Recap: (From section 2.1) - `users` (doc per user with profile/integration info) - `users/{uid}/children` - `users/{uid}/dailyContent` (group doc per day) - `users/{uid}/children/{childId}/dailyContent` (child-specific docs per day) - Possibly `users/{uid}/integrationTokens` (doc(s) for tokens) - Possibly `users/{uid}/printQueue` or `printJobs` if implementing print trigger via Firestore.

We should also consider indexing: - If we allow queries like “list all dailyContent for user sorted by date”, Firestore requires an index for certain composite queries. If each dailyContent doc ID is the date, we can do an orderBy **name** (since doc IDs are lexicographic, ISO date works lexicographically mostly if format YYYY-MM-DD). - Or we store a timestamp field and index on that. - If multi-user, maybe not needed beyond per-user queries which are fine (no cross-user queries). - If listing children alphabetically or by age, might index age.

We will define in `firestore.indexes.json` if needed: e.g., an index on `users/{uid}/dailyContent` by `date` (if date is a field) or we can use the default by document ID if structured.

8.3 Firebase Storage Structure Recap: - We might not need separate buckets; one bucket with folder per user is enough. - If content volume is low, one bucket is fine. If scaling huge, could consider multiple buckets but not needed here. - Set Storage rules as described:

```
rules_version = '2';
service firebase.storage {
  match /b/{bucket}/o {
    match /videos/{userId}/{filename} {
      allow read: if request.auth != null && request.auth.uid == userId;
      allow write: if request.auth != null && request.auth.uid == userId;
    }
    ... (similar for worksheets, coloring, etc.)
  }
}
```

Actually, since writing is done by admin (n8n with service account), the rules don't stop admin. But if theoretically a malicious user tried to write via client, this will block if not their file. We might refine to allow write for admin or perhaps allow no client writes at all: - We could do: `allow write: if false;` for all, since the app isn't expected to upload any content from client. The only writes come from server which bypass rules. This might be the safest approach to ensure users can't upload arbitrary files (unless we plan to allow user to upload profile pics for children or something). - If profile pics or any user file upload feature is needed, we'd have to allow write for those specific paths. - For now, likely no user-uploaded content except maybe profile avatar. If we want profile picture, we can allow writes to `users/{uid}/profilePictures/{childId}.jpg` on rule that auth.uid == uid. That's a minor addition if needed.

8.4 Cloud Functions Structure: - Organize functions code in multiple modules (for clarity): - `auth.js` (for OAuth handling function), - `googleIntegration.js` (functions to handle creating calendar events, etc., possibly triggered on Firestore changes), - `manualTrigger.js` (function for generate on demand, if implemented), - Possibly `notify.js` for sending notifications (if we do that). - We'll use environment config for secrets: - `functions.config().google.client_id` and `...client_secret` for OAuth. - Or store in Secrets (Firebase has a way to store these and fetch at runtime). - OpenAI keys etc. not needed in functions if all done in n8n. - Use `firebase-functions` SDK to define triggers. E.g., for Firestore trigger on content creation:

```
exports.onContentGenerated = functions.firestore
  .document('users/{uid}/dailyContent/{date}')
  .onCreate(async (snap, context) => { ... });
```

Inside, do Google Calendar/Drive/Sheets stuff.

- **Deployment**: Manage via `firebase.json` and `firebase-cli`. Possibly separate function into multiple to not block each other:
- If doing everything in one onCreate, note that writing to Drive and to Sheets and to Calendar could be done in parallel with `await Promise.all(...)` since they are independent API calls. But ensure not to exceed function time (should be fine, each call a few seconds at most).
- If heavy, could separate triggers or chain them (but one trigger is fine).
- Mark these functions to not run multiple times incorrectly (if Firestore does repeated onCreate if we do retries).

8.5 Additional Firebase Services (if any): - **Firebase Cloud Messaging (FCM):** Not mentioned, but we could incorporate to send a push notification to user's device. If doing so: - We'll enable FCM in project. - In Flutter app, use `firebase_messaging` to get device token and store in Firestore (like in user doc). - Cloud Function `onContentGenerated` can then send a notification "Today's Operation Summer content is ready!" via FCM. - This is a nice addition for UX, albeit optional. We mention as optional in spec. - **Firebase Crashlytics or Analytics:** Could integrate for app stability and usage tracking, though not required by user, but a production-grade app might include it for debugging. We might mention briefly that we can use Crashlytics to capture any runtime errors in app, and maybe use Analytics to see engagement (like how often content viewed). - **Firebase Performance Monitoring:** Also optional for measuring app performance, likely overkill here.

8.6 Dev & Prod Environments: Possibly have different Firebase projects for dev vs prod. We should allow configuration such that developers can run the app on a test project. - So we'll not hard-code project IDs; use `google-services.json` / `GoogleService-Info.plist` for the app. - The n8n and cloud functions similarly could target a dev environment (with different credentials). - For CI, we might set up separate config or use Firebase's feature to use different alias on deploy.

9. Deployment Plan (Flutter, Firebase, n8n, CI/CD)

9.1 Flutter App Deployment: - **Web:** We will deploy the Flutter web app to Firebase Hosting. Steps: 1. Run `flutter build web --release` which generates `build/web` directory with `index.html`, etc. 2. Configure `firebase.json` for hosting:

```
"hosting": {
  "public": "build/web",
  "ignore": [ "firebase.json", "**/.*", "**/node_modules/**" ],
  "rewrites": [ { "source": "/**", "destination": "/index.html" } ]
}
```

This rewrite allows Flutter's client-side routing to work (all paths serve index.html). 3. Run `firebase deploy --only hosting` to upload. - The app will then be accessible at a Firebase-provided domain or custom domain if set. - We need to ensure the OAuth redirect domain for Google includes the hosting domain (for web OAuth if using it). - **Mobile:** - For Android, generate an APK/AAB via `flutter build apk` or `appbundle`. Sign it with a key. Submit to Google Play or sideload. - For iOS, run `flutter build ios` to get an .ipa after configuring signing in Xcode, then distribute via TestFlight or App Store. - If this is just for a small group, we can distribute AdHoc or via Firebase App Distribution for testing. - We should ensure the Google Services config files (google-services.json/plist) are included and the Flutter app is configured with the Firebase project details (via `firebase_core` initialization).

9.2 Cloud Functions Deployment: - Use Firebase CLI: `firebase deploy --only functions`. This will upload our Node.js code. - We will set environment config for any secrets: - e.g., `firebase functions:config:set google.client_id="XYZ" google.client_secret="ABC" openai.key="..." creatomate.key="..."` etc. (OpenAI/Creatomate keys might only be used in n8n, not needed in functions, but if we choose to move some generation to functions in future, good to have). - Alternatively use `firebase functions:secrets:set`. - Ensure in code we retrieve those via

`functions.config()`. - After deploy, test that functions (like OAuth callback) are reachable and working. (We might simulate an OAuth callback by hitting it with a known code or something in dev).

9.3 n8n Deployment: n8n being a separate system requires: - Setup a **server** or environment to run n8n continuously. Options: - Run as Docker container on e.g. a local machine (with restart policy). - Use n8n cloud or similar – but that usually wouldn't have printer access and storing secrets. For printing, local is best. - Let's assume we run it on a local mini PC with Ubuntu. - Steps: 1. Install Docker and run n8n image, or install n8n via npm globally. 2. Configure n8n: - Set it up with environment variables for needed access: * `N8N_BASIC_AUTH_USER` and `N8N_BASIC_AUTH_PASSWORD` if we want to secure the editor (since it might be accessible via a port). * `NODE_FUNCTION_ALLOW_EXTERNAL=pdf-to-printer` (to allow using that package in Function nodes) [9]. * Possibly `N8N_ENV_FIREBASE_KEY` or similar with service account credentials (unless we include those in workflow). * Or mount a credentials file if using service account for Firestore. * Set `WEBHOOK_TUNNEL_URL` if we want to expose any webhook externally (for manual trigger), or we can keep n8n local only. - We'll create credentials within n8n for: * OpenAI (API Key). * If n8n has a Firebase credential type or just use HTTP with service account. * If we want n8n to handle any Google API calls directly (less likely). * Creatomate API Key. * Possibly a generic HTTP credential for any authenticated calls. - Create the workflow as designed, test it within n8n editor. - Activate the workflow so it runs on schedule. 3. To run n8n on system startup, if using Docker, use a system service or simply a cron @reboot or so. If using npm, maybe use PM2 process manager. 4. Ensure n8n has access to internet (for OpenAI, etc.) and to the printer (should see the printer if OS has it). 5. We may also schedule regular backups of n8n workflows or use its database (likely we use default SQLite if not many workflows). - Security: Because n8n will hold API keys, ensure the machine is firewalled (no open n8n editor to internet without auth). - We might not need to expose n8n's interface publicly at all if comfortable triggering manually via other means. But to allow the cloud function to call a webhook on n8n for manual triggers, we'd need an accessible URL or use something like a polling approach (cloud function sets Firestore, n8n sees that). - We could implement that by having n8n watch a "commands" collection for a doc like `generateNow` and then run generation. But easier to just call a webhook. Could also run an n8n tunnel or use a service like webhookrelay if needed. - Given complexity, we might skip manual triggers and rely on daily schedule primarily.

9.4 Domain & Hosting Considerations: If this app will be used externally, using a custom domain for Firebase Hosting might be nice (like operationsummer.example.com). Not necessary but could be done by verifying domain, etc.

9.5 CI/CD Pipeline: - We will set up a git repository for the code (both Flutter and functions). Possibly separate repos or monorepo. Could be monorepo with `frontend/` and `functions/` directories. - Use **GitHub Actions** (or GitLab CI, etc.) to automate tests and deployment: - Example: On push to main branch: * Run Flutter analyzer (`flutter analyze`) and tests (`flutter test`). * Run `npm run lint` and maybe tests for functions if any. * If all good, build flutter web and deploy to Firebase Hosting via CLI (use Firebase service account or token). * Deploy Cloud Functions similarly (could be combined or separate workflow). * This requires storing Firebase deploy token or service credentials in GitHub secrets. - Alternatively, use manual deploy for production to avoid accidental overwrites. For now, could do CI in staging environment and manual promotion to prod. - For mobile apps, setting up CI to build and perhaps automatically upload to app stores is more involved (might use Codemagic or GitHub Actions with Mac runners for iOS). Possibly out of scope, but mention: - Could use fastlane or codemagic for automating builds and uploads when tagging releases.

9.6 Testing and QA: - We will aim for **test coverage** on critical parts: - Flutter: Unit tests for any logic (like maybe some utils for date formatting or filtering content). Widget tests for UI components to ensure they build correctly with dummy data. - Cloud Functions: Write tests using firebase-functions-test SDK to simulate events (like simulate Firestore doc create and check if our function calls Google API functions (could mock Google API calls)). - The n8n workflow itself is harder to unit test in isolation, but we can test the sub-routines (like we could create small Node scripts mimicking what n8n code nodes do). At minimum, we test the prompts (maybe using OpenAI's sandbox) to ensure output format. We will likely rely on manual testing for the workflow. - We'll create some **integration tests**: - e.g., a script that triggers content generation for a test user and then checks Firestore to see that documents were created with expected fields, and maybe check that files exist in Storage (Firebase Admin SDK can list files). - This could be part of CI if we have a test environment (but it would call external APIs costing money, so careful with automated tests hitting OpenAI). - Perhaps limit automated integration tests to using mock OpenAI (not trivial). So perhaps do those tests manually in staging environment.

9.7 Monitoring & Maintenance: - Use Firebase console to monitor function invocations, logs (for errors). - Use n8n's own logs or add error catching to send emails. - Possibly integrate Sentry or similar for Flutter app to catch runtime issues (especially as it may run on various devices). - Plan for content updates: Over time, maybe we want to refine prompt or add new features. We should make prompt text easily adjustable (maybe not hard-coded in n8n nodes but in some config or at least clearly commented). - Manage API keys and usage quotas: - Monitor OpenAI usage in their dashboard, ensure we don't exceed monthly limits. - Monitor Google API usage (should be fine, a few calls per day per user). - If usage grows, consider caching or reusing some content or use cheaper model where possible.

File/Folder Structure Recommendations

To keep the project maintainable, we outline a recommended structure for both the Flutter project and the backend code:

Flutter Project Structure: We will use a **feature-first** organization (per Flutter best practices) where each feature has its own directory, containing its screens, widgets, models, etc. [17]. This makes it scalable as we add features.

```
/lib
|-- main.dart          # entry point, sets up Firebase and runs the app
|-- app.dart           # MaterialApp and route setup
|-- constants/         # constant values, e.g., theme styles, strings
|-- models/            # data model classes (if not feature-specific)
|-- services/          # general services (e.g., maybe a firebase_service.dart
to encapsulate Firestore calls)
|-- providers/         # global state management (if using Provider or Riverpod,
could have providers here)
|
|-- features/
|   |-- auth/
|   |   |-- login_screen.dart
|   |   |-- auth_service.dart (if handling auth logic like sign-in flows)
```

```
|   |
|   |-- profile/
|   |   |-- children_screen.dart
|   |   |-- child_form_screen.dart
|   |   |-- child_model.dart   (could be in models/ as well)
|   |
|   |-- daily_content/
|   |   |-- daily_content_screen.dart
|   |   |-- content_model.dart (maybe a model representing daily content)
|   |   |-- widgets/
|   |   |   |-- video_card.dart
|   |   |   |-- activity_list.dart
|   |   |   `-- ... (other UI components)
|   |
|   |-- settings/
|   |   |-- settings_screen.dart
|   |   |-- integrations_manager.dart (logic to handle link/unlink Google, etc.)
|   |
|   |-- archive/ (if implemented)
|   |   |-- archive_screen.dart
|   |   |-- archive_detail_screen.dart
|
|-- widgets/           # shared generic widgets (e.g., a common PrimaryButton)
|-- utils/             # utility functions (e.g., date formatting, open URL for
YouTube, etc.)
```

Explanation: - We separate features such as Auth, Profile (child management), Daily Content display, Settings, Archive. Each feature can have its own subfolder with screens and any related logic. - For state management, if using Provider, we might have ChangeNotifier classes, which can reside either in `providers/` or within features. E.g., a `ContentProvider` to manage loading daily content could live in `daily_content/` or `providers/`. - The `services/` folder can contain wrappers for Firebase calls or other API interactions. For example, a `GoogleApiService` to handle making requests (though most Google actions are done by backend, the front-end might not call directly except for OAuth). - Keep UI code (screens and widgets) separate from logic (providers/services) to follow MVVM-like separation.

Firebase Functions Structure: We will structure the Cloud Functions codebase with multiple source files for clarity, using ES modules or CommonJS as supported:

```
/functions
|-- package.json
|-- firebase.json (for deploy config if separate, but usually in project root)
|-- /.env (if using environment file for local emulation, not for production
secrets though)
|-- src/
|   |-- index.ts (or index.js) # main entry where we import and export the
functions
```

```
|   |-- oauth.ts              # contains the oauthCallback function definition
|   |-- integrations.ts       # contains functions for handling calendar/drive/
sheets updates
|   |-- generate.ts           # contains the callable function for manual
generation if any
|   |-- print.ts (optional)   # if a function related to print or notifications
|   |-- ... (other helper modules if needed)
```

We prefer TypeScript for Cloud Functions to catch errors early, but JavaScript is fine too. The above shows TypeScript structure (we'd have to compile TS -> lib/). If using TS:

```
|-- tsconfig.json
|-- lib/
|   |-- index.js (compiled output)
|   |-- ...
```

If using plain JS, just keep under `functions/index.js` requiring other `.js` files.

n8n Workflow and Config: n8n doesn't have a file structure since it's configured via its DB/UI, but we can export the workflow JSON and keep it in version control for reference:

```
/n8n_workflows/
|-- daily_content_workflow.json
|-- (possibly other workflow JSONs if multiple)
```

And maybe a README on how to import them.

We should also maintain environment setup for n8n:

```
/n8n/
|-- .env (to configure n8n container, containing credentials or at least
references)
|-- Docker-compose.yml (if using docker)
```

This is not part of code repository usually, but for operations.

Other Files: - Root `README.md` for the project. - Possibly in Flutter project an `assets/` folder if we have any local assets (images, etc.). We might not have many local assets, maybe just the app logo or an icon. - Add in Flutter pubspec dependencies: e.g. `firebase_core`, `firebase_auth`, `cloud_firestore`, `firebase_storage`, `firebase_functions`, `provider`, `video_player`, `pdf_viewer` (for viewing PDFs, maybe use `syncfusion_flutter_pdfviewer` or similar), etc. - Ensure to not commit any secrets (like Google OAuth client secret) in code.

10. Flutter Component Trees (UI Implementation Details)

We provided an example component tree in section 1.5. Here we add some more detail to key screens:

DailyContentScreen Tree (expanded):

```
DailyContentScreen (StatefulWidget)
└─ StreamBuilder<DailyContent> (listens to Firestore for today’s content for selected child & group)
   └─ (if data not yet available) LoadingIndicator or Text("Generating...")
   └─ (if data available) ListView (scrollable column) containing:
      ├─ VideoCard (if videoUrl exists)
      |    ├─ Container or Card with a thumbnail (we might generate a thumbnail using video_player plugin, or simply an icon)
      |    └─ PlayButton (tapping opens a VideoPlayerScreen or uses a video_player within the card to play inline)
      ├─ ColoringCard
      |    ├─ Image.network (with Firebase Storage download URL for coloring image, or use FirebaseStorage ref with image widget if using firebase_ui)
      |    └─ Text(childName + "'s Coloring Page", overlaid or below image)
      |    (maybe a Download button to save image to device)
      ├─ WorksheetCard
      |    ├─ Icon (PDF icon) or thumbnail (Generating thumbnail for PDF is complex; maybe just show an icon and title)
      |    └─ Text("Worksheet") and maybe number of questions
      |    (tapping opens PDFViewerScreen which loads from Storage URL)
      ├─ GroupActivitiesCard
      |    ├─ Text("Today's Group Activities")
      |    ├─ Text("Project: " + projectIdea)
      |    ├─ Text("Exercise: " + exerciseIdea)
      |    ├─ Row("Song: " + songTitle, [Open YouTube] button or link icon)
      ├─ PrintStatusWidget (if printed field in Firestore is true -> shows "Printed   at [time]"; if false -> "Not printed  ", with a button "Print Now")
```

(The actual UI may use Flutter Cards or just Padding and Column to style each section.)

If multiple children, perhaps above list is per child. We might instead have a TabBar or dropdown at top for each child:

```
DailyContentScreen
└─ Column
    ├─ DropdownButton (or TabBar) for child selection (list of child names)
    ├─ contentListView (as above, but filtered to that child’s content plus group content)
```

We have to ensure switching child triggers StreamBuilder to listen to that child's doc. Could achieve by using a StreamBuilder that depends on `selectedChildId`. Or use a separate stream for group doc and combine with child doc stream.

ProfilesScreen Tree:

```
ProfilesScreen (StatelessWidget or Stateful if we load via Future)
└─ StreamBuilder<List<Child>> (Firestore collection of children)
   └─ ListView
       ├─ ListTile or Custom ChildCard for each child
       |    ├─ leading: (optional image avatar if we have profile pic)
       |    ├─ title: child.name
       |    └─ subtitle: "Age: X"
       |   (onTap -> maybe open edit, or on trailing icon -> edit)
       └─ ListTile (Add Child)
            └─ trailing: IconButton(add)
```

AddChildScreen Tree:

```
AddChildScreen (StatefulWidget)
└─ Form
    ├─ TextField Name
    ├─ TextField Age
    ├─ (optional) Multi-select chip or dropdown for interests (not required,
could skip)
    ├─ Save Button (onPressed -> call Firestore add)
    (If editing, fields prefilled and we update existing doc on save)
```

SettingsScreen Tree:

```
SettingsScreen (StatefulWidget)
└─ ListView
     ├─ SectionTitle("Google Integration")
     ├─ ListTile (Google account link status)
     |    └─ trailing:
     |         if linked -> Text(linkedAccountEmail, style: green)
     |                   -> also a "Unlink" button maybe.
     |         if not linked -> ElevatedButton("Link Google Account")
     ├─ if linked:
     |    ├─ CheckboxListTile [ ] Add events to Google Calendar
     |    ├─ CheckboxListTile [ ] Save content to Drive
     |    ├─ CheckboxListTile [ ] Log activities to Google Sheet
     |         (these update in Firestore user doc or local state and then
```

```
saved)
     |
     ├─ SectionTitle("Printer")
     ├─ (We might not know printer status from app side. We could show
instructions:)
     |    Text("Printing is handled by the server. Ensure the printer is on and
connected to the Operation Summer server.")
     |    (Potentially show a last print status which we have from Firestore;
e.g., "Last print: success on [date]" or error if any)
     |
     ├─ SectionTitle("Notifications")
     ├─ SwitchListTile "Daily Ready Notification"
     |    (toggles whether to subscribe to FCM topic or store device token in
Firestore for notifications)
     |
     ├─ SectionTitle("Account")
     ├─ ListTile "Sign out"
```

We divide by sections just for clarity in UI.

We will apply appropriate theming (via ThemeData in MaterialApp) to ensure a consistent look.

Navigation: We might use a Drawer to navigate between main sections (Home, Profiles, Settings, Archive), or use a BottomNav if only few sections. Possibly: - BottomNavigationBar with tabs: Home, Profiles, Settings (and maybe Archive under Home as a subview or could be a tab). - Or simpler, a hamburger menu Drawer for rarely used pages like Settings, and have Home with child content by default.

Given typical usage, after initial setup, user mainly looks at daily content. So that is the main screen. Profile and Settings are secondary.

11. API Contracts for Backend Services (Recap and Additional Context)

We described Cloud Function endpoints earlier. Here we ensure clarity and completeness:

11.1 Cloud Function HTTP API: - `GET /oauthCallback` – (Not a RESTful resource in traditional sense, but it's an endpoint). No auth required (Google calls it), but uses `state` to link to user. - Returns: a simple HTML page. (We might not expose a JSON API here). - Side-effect: stores tokens and updates user doc as described.

We might add a convenience: - `GET /api/triggerGenerationNow?uid=XYZ` – Could be a secure endpoint to trigger immediate gen for a user (if, say, an admin calls it or some monitoring). But since we have other means, we might not implement this.

11.2 Cloud Function Callable: - `generateContent` (callable from app): - Input: JSON with optional date or child. - Auth: Firebase Auth automatically included; in function we verify context.auth.uid. - Behavior: posts a message to n8n or Firestore to start generation, or directly calls generation logic if implemented in CF (but

we offloaded to n8n, so likely it signals n8n). - Response: for example `{ started: true }` or if fails `{ error: "..."}` - The app will likely just trust it started and wait for Firestore updates.

- `printContent` (callable):
- Input: maybe date (if wanting reprint older day) and optional child.
- Behavior: Similar, create a print request.
- Response: immediate ack.

- In practice, for reprint, perhaps easier for app to just set Firestore flag as earlier stated. But either works.

- We will use Callable for these to leverage Firebase Auth and not open a separate public HTTP.

11.3 Firestore Data: While not a REST API, treat each collection as an API resource: - `Children`: accessible under the user, basic CRUD via Firestore SDK. The app will call `add` for new child, `update` for edits. Firestore ensures offline sync as well if network issues. - `DailyContent`: read (no writes from client except maybe marking something like they completed tasks, which could be a new feature like user marking tasks done – not in spec but possible. For now, client doesn't write to dailyContent). - Actually, client might update a field in dailyContent to acknowledge something? But not needed. - Print status might be updated by backend (function or n8n) and read by client.

11.4 Third-Party API Contracts:

We also define how we interact with external APIs (for development reference):

- *OpenAI Chat Completion API*:
- Endpoint: `POST https://api.openai.com/v1/chat/completions`
- Request body: `model: "gpt-4"`, `messages: [{role: "system", content: "..."}, {role: "user", content: prompt}]`, `temperature: 0.7, max_tokens: e.g. 500`.
- Response: JSON with `choices[0].message.content` containing the answer (which in our case is JSON string or structured text).

- In n8n, the OpenAI node abstracts this, so we might not manually call REST.

- *OpenAI Image API*:

- Endpoint: `POST https://api.openai.com/v1/images/generations`
- Body: `{ prompt: "text", n:1, size:"1024x1024" }`.
- Response: contains `data[0].url`.

- In n8n, the OpenAI node's "Generate Image" uses this under the hood.

- *Google Calendar API*:

- Endpoint: `POST https://www.googleapis.com/calendar/v3/calendars/primary/events`
- Auth: Bearer access_token.

- Body: e.g.

```
{
  "summary": "Operation Summer: Project - Build a Rocket",
  "description": "Help the kids build a model rocket using household
items.",
  "start": { "dateTime": "2025-07-15T10:00:00", "timeZone": "America/
Chicago" },
  "end": { "dateTime": "2025-07-15T11:00:00", "timeZone": "America/
Chicago" }
}
```

Similarly another for exercise at 3pm.

- Or maybe all-day events with just date (depending on preference). But scheduled times are more structured.

- *Google Drive API*:

- Create folder: `POST https://www.googleapis.com/drive/v3/files` with body `{"name": "Operation Summer", "mimeType": "application/vnd.google-apps.folder", "parents": [<parent folder id if any>]}`.
- Upload file: `POST https://www.googleapis.com/upload/drive/v3/files?uploadType=multipart` with multipart form containing metadata JSON part: `{"name":"2025-07-15_Alice_Worksheet.pdf", "parents": ["<OperationSummerFolderID>"], "mimeType":"application/pdf"}` and second part file content binary.
- We will use Google's Node client so it simplifies to calls like `drive.files.create({ requestBody: {...}, media: {...} })`.

- *Google Sheets API*:

- Append row: `POST https://sheets.googleapis.com/v4/spreadsheets/{spreadsheetId}/values/Sheet1!A1:append?valueInputOption=RAW`
- Body: `{"values": [[ "2025-07-15", "Space Adventures", "Build a rocket", "Moon jump", "\"Twinkle Twinkle Little Star\"", "Alice (7): 5 math Qs", "Bob (5): 5 counting Qs" ]]}` (just an example row).
- Or we use the Node client: `sheets.spreadsheets.values.append({ spreadsheetId, range: ..., valueInputOption: "RAW", insertDataOption: "INSERT_ROWS", resource: { values: [...] } })`.

- *Creatomate API*:

- As described earlier, one POST to `/v1/renders` with JSON containing `template_id` and modifications. Example from docs:

```
{
  "template_id": "xxxxx-xxxx-xxxx",
  "modifications": {
    "Voiceover-1": "Hello, this is the text to speak",
    "Text-1": "Hello, Alice!",
    "Image-1": "https://example.com/image.png"
  }
}
```

- Response: likely contains a `renderId` and maybe a `status: "queued"` or maybe it blocks until finished and provides `renderUrl`.
- If needed, follow-up GET to `/v1/renders/{id}`.

- We'll rely on their documentation to implement correctly. They mention using a single HTTP request can produce the video [18] [11], meaning possibly they wait until done and give output or a link.

- *ElevenLabs API (if used separate)*:

- Endpoint: `POST https://api.elevenlabs.io/v1/text-to-speech/{voice_id}`
- Body: JSON with text and settings.
- Needs an API Key in header.

- We might not call directly if Creatomate handles it.

- *pdf-to-printer (not an API but a function)*:

- If using the library, just note that it invokes underlying OS commands.

11.5 Communication Between Systems: - The Flutter app communicates with Firebase (Firestore/Storage/Functions) – we ensure network rules allow that (should by default). - n8n communicates with Firebase using service account: we need to provide the service account credentials to n8n. Possibly set `GOOGLE_APPLICATION_CREDENTIALS` env pointing to the JSON file, so that any Google client libraries pick it up. Or directly use them in code nodes. - Alternatively, simpler: use Firestore REST API with a service account's REST token: - Could do JWT auth: create a custom token and exchange for OAuth2 token for Google APIs. Firestore REST accepts OAuth2. - If that's too heavy, bring in Firebase Admin SDK which uses service account key directly. Possibly easier if we embed small Node code with `admin.initializeApp({ credential: cert(serviceAccount) })` and then `admin.firestore()` usage. But doing that inside n8n might conflict if n8n is itself using some firebase internally (likely not). - However, n8n function nodes run in isolated context, it might be okay to initialize admin SDK in one. We must ensure the service account JSON is accessible (could store its JSON in an environment var or in a file on disk). - For simplicity, maybe use HTTP nodes to write to Firestore via the Firestore REST API: * Example: `POST https://firestore.googleapis.com/v1/projects/<project>/databases/(default)/documents/users/UID/dailyContent/2025-07-15` with body data and an Authorization header with a Bearer token. That token can be generated by service account by JWT sign or maybe use the IAM access token approach. Might be easier to just use the Admin SDK as said. - We'll assume use of Admin SDK in code node since we already consider using Node libs for printing, etc.

11.6 Security and Auth Recap: - Only authenticated users can call functions and read their data. - The service account in n8n has full read/write, but that is outside of client control (we trust our server). - Ensure n8n's service account key is kept secret. If the n8n host is secure, fine. Alternatively, we could limit its role (but service account probably is Editor for entire project which is normal for this usage). - Google OAuth tokens provide limited access (only the scopes we requested). - In the Flutter app, prevent any sensitive info from being hard-coded or logged (like refresh tokens – but we don't expose them to app anyway). - If multiple families use this system, enforce strict separation in Firestore and never use any read that isn't scoped by user. We did that by always including uid in path.

11.7 Scalability Notes: - If tomorrow we have, say, 100 users (families) with 2-3 kids each: - n8n workflow might need to loop through 100 users daily. That means ~ maybe 100 GPT-4 calls, 100 images, etc. That might be slow or expensive. We could consider optimizing: * Possibly schedule different users at different times (some at 5:00, some at 5:10 etc.) to distribute load. * Or run multiple n8n instances (not ideal). * But at 100, GPT-4 calls serially might be ~100 * 5s each = 500s ~ 8 minutes, which is borderline but maybe okay. But adding image and video generation, could be a couple minutes per user easily, which means hours for 100 sequentially. * So if that scale, better approach is parallelize user processing. Could run multiple workflows or threads. n8n might not easily do multi-thread in one workflow, but we could set up multiple cron workflows each filtering a subset of users (not elegant). * Alternatively, ditch n8n and use Cloud Functions or Cloud Run to parallelize generation for each user triggered by a scheduler or pubsub events. But since this is initial spec, we assume small scale and n8n sequential is fine. - The backend (Firestore, Storage) can handle that scale easily. Google APIs also fine as usage is low per user. - The printing part: if 100 different printers (one per user in their home), our printing solution doesn't scale globally easily unless each user runs their own local n8n or some agent. If it were productized, we might have to integrate something like PrintNode so that each user installs an agent and our cloud triggers prints via the PrintNode API. That would be a different architecture. - But likely it's intended for one site or a few sites (like a small school or a community center), not a commercial product for many separate environments. So we proceed with single environment assumption.

11.8 Internationalization: - Not explicitly asked, but note if we want to adapt to other languages, OpenAI could generate content in other languages if prompted accordingly. But probably out of scope; assume only English for now.

12. Deployment Steps & CI/CD (Summary)

We have touched on deployment in section 9, here is a concise checklist style summary for production deployment:

- **Firebase Setup:**
- Create Firebase project "operation-summer".
- Enable Authentication (email/Google) and set up authorized domains.
- Add iOS and Android apps in Firebase console (get config files) if mobile app is to be used.
- Initialize Firestore and Storage in locked mode, then set security rules as per design.
- Deploy initial dummy rules (or open rules for development if needed, but then secure them).
- Set up Cloud Functions: `firebase init functions` (choosing TypeScript or JS). Write functions code. Set config for OAuth client secrets: `firebase functions:config:set google.client_id="..."`.
- Deploy functions: `firebase deploy --only functions`.

• Test functions (e.g., manually hit OAuth URL, see if token stored).

• **Flutter App:**

• In Flutter code, set up firebase (call `Firebase.initializeApp` with options or use the default if using google-services config on mobile).
• For web, ensure the Firebase config snippet is included (either via `index.html` or using FlutterFire CLI to configure).
• Test app in debug mode connecting to Firebase (maybe use Firebase Emulator for Firestore during dev, optional).
• Build release versions. For web: `flutter build web`.
• Deploy web: `firebase deploy --only hosting` (make sure hosting is configured to serve build/web).
• For mobile: gather release builds, upload to stores (this might be a longer process with store listing etc. if intended to be on app stores).
    ◦ Possibly for a smaller user base, they might skip store and just use web or side-load.

• Post deploy, do a sanity test: create a user, add a child, and manually trigger generation or wait for schedule to see if everything flows.

• **n8n Setup:**

• Provision the machine (with Node or Docker).
• If Docker: `docker run -d --name n8n -p 5678:5678 -v ~/.n8n:/home/node/.n8n -e N8N_BASIC_AUTH_USER=user -e N8N_BASIC_AUTH_PASSWORD=pass -e NODE_FUNCTION_ALLOW_EXTERNAL=pdf-to-printer -e GOOGLE_APPLICATION_CREDENTIALS=/files/serviceAccount.json -v /path/to/serviceAccount.json:/files/serviceAccount.json n8nio/n8n:latest` (This runs n8n with basic auth and mount for persistence and service account).
• If not Docker, install and run as a service similarly.
• Open n8n UI in browser, create credentials:
    ◦ Add OpenAI credentials (API Key).
    ◦ Possibly add HTTP credentials for Firebase (but using service account we might call admin SDK in function nodes).
    ◦ Add any needed for creatomate: actually, creatomate doesn't have an official n8n node, so likely we use HTTP node with Bearer token, so no special credential needed beyond storing API key in the HTTP header (which we can store in a credential for reuse).
    ◦ ElevenLabs if separately used can be stored similarly.
• Import or create the workflow:
    ◦ Cron trigger -> Function nodes & HTTP nodes as per our design.
    ◦ Use function node at start to initialize Firebase Admin: something like

```
const admin = require('firebase-admin');
if (!admin.apps.length) {
  const serviceAccount =
JSON.parse(process.env.FIREBASE_SERVICE_ACCOUNT_JSON);
  admin.initializeApp({ credential:
```

```
admin.credential.cert(serviceAccount) });
}
return { payload: "Firebase initialized" };
```

(We either pass the JSON via env or mount file. We showed mounting file and using `GOOGLE_APPLICATION_CREDENTIALS`, so we could directly do `admin.initializeApp()` without passing credentials if env var is set, admin SDK might pick it up. If not, do as above reading JSON from file path.)

- Then another node to fetch data, then GPT, etc. Build out the logic.

- Set the schedule in Cron node (e.g. every day at 05:00).
- Activate workflow.
- Test run it (maybe temporarily change schedule or trigger manually via n8n UI).
- Check Firestore/Storage/printer to verify outputs.

- Troubleshoot any errors and iterate.

- **CI Setup:**

- Use a git repo, integrate a pipeline as earlier described. Perhaps not strictly needed if it's a small project, but since spec asks, we mention it.
- Ensure secrets (Firebase deploy token, etc.) in CI.
- Possibly separate jobs: one for lint/test, one for building Flutter, one for deploying functions, one for deploying hosting, triggered appropriately.

13. Optional Test Coverage and CI/CD Notes

Test Coverage Targets: - Aim for ~80% unit test coverage for critical pure functions in Flutter (like any computation logic). - For UI, use widget tests for at least main screens to catch build errors (maybe target 50% of widgets have basic render test). - Cloud Functions: target > 70% coverage via simulation of events (we can simulate Firestore events and test that the function attempts certain API calls by mocking Google API clients). - The n8n workflow cannot be easily unit-tested, but we will test it end-to-end in a staging environment. We can also add small unit tests for sub-modules if we break some logic into separate scripts. - Use static analysis: `flutter analyze` should show 0 errors, maybe use `dart analyze` in CI and `eslint` for functions.

CI Pipeline Outline: - On each commit or PR: 1. Run Flutter tests and analysis. 2. Run Functions tests and lint. 3. Possibly build the Flutter web and ensure build succeeds (but not deploy on every PR). - On merging to main or a release branch: 1. Build and deploy Flutter web to a staging site or directly to prod if confident. 2. Deploy functions to Firebase. 3. (For mobile, we might manually handle store releases, or use CI to at least build artifacts and attach to a GitHub release for manual upload). - Use environment separation: maybe have `firebase use staging` for merges to main, and manually promote to prod project for actual production run, depending on project needs.

Continuous Delivery: - Could implement automatic daily or weekly builds of the app if content or logic changes frequently, but since this is not a user-facing content app (the content is dynamic from AI, not from code changes), not necessary beyond code changes. - We will maintain version control and semantic versioning for app releases.

Summation: Following this technical specification, developers (or an AI agent) should be able to implement Operation Summer. The design balances the capabilities of Flutter/Firebase for realtime user interaction with the power of cloud automation and AI services to deliver rich, personalized content daily. All critical aspects from architecture to implementation details and integration points have been covered, providing a clear blueprint for building and deploying the system.

---

1 11 18 How to Add an AI Voice Over to a Video using an API? - Creatomate
https://creatomate.com/how-to/api/add-voiceover-to-a-video

2 16 Migrate from Cloud Print - Chrome Enterprise and Education Help
https://support.google.com/chrome/a/answer/9633006?hl=en

3 8 10 OpenAI node documentation | n8n Docs
https://docs.n8n.io/integrations/builtin/app-nodes/n8n-nodes-langchain.openai/

4 5 12 13 How to Create AI Voice Over Videos using an API - Creatomate
https://creatomate.com/blog/how-to-create-voice-over-videos-using-an-api

6 15 Super simple PDF printing service with node.js. | by Philipp Bauknecht | Medialesson | Medium
https://medium.com/medialesson/super-simple-pdf-printing-service-with-node-js-c9e958aa019b

7 google api - What does "offline" access in OAuth mean? - Stack Overflow
https://stackoverflow.com/questions/30637984/what-does-offline-access-in-oauth-mean

9 Send file to printer - Questions - n8n Community
https://community.n8n.io/t/send-file-to-printer/27796

14 gps1mx/pdf-to-printer2: Print PDF files from Node.js and Electron.
https://github.com/gps1mx/pdf-to-printer2

17 Flutter Best Practices to Follow in 2025 - Aglowid IT Solutions
https://aglowiditsolutions.com/blog/flutter-best-practices/