Get the source code on GitHub https://github.com/lsvekis/Google-Apps-Script-APIs-and-Gemini

📘 Introduction — Building AI-Powered Workflows with Apps Script, Gemini, and Real-World APIs

Welcome to this hands-on guide where you'll learn how to build powerful AI-enhanced workflows using **Google Apps Script**, **Gemini**, **external APIs**, and **Google Drive** as a document knowledge base.

Whether you're a beginner or an intermediate developer, this guide will take you step-by-step through building real, working "AI assistants" that combine:

- Local documents
- External API data
- Vector embeddings
- RAG (Retrieval-Augmented Generation)
- Chat memory
- Web app UIs
- Multi-workspace knowledge routing

This guide is designed to be **practical**, **approachable**, and **fully reproducible**. Every chapter includes complete code you can paste directly into Apps Script, along with explanations and real use cases.

🌟 What You Will Build

Across the lessons, you'll progressively create a suite of intelligent assistants:

Lesson 1 — Basic API Fetching

- Learn how to call free public APIs (Cat Facts, Bored API, Dad Jokes, Open-Meteo).
- Build a clean helper function that fetches data safely.
- Prepare your Apps Script environment with Script Properties.

Lesson 2 — Combine API Data + Drive Folder Content

- Read all text-based documents in a Drive folder.
- Merge that content with external API responses.
- Feed both into Gemini to generate contextual answers.
- Build a simple HTML Service web app UI.

Lesson 3 — Improve the Web App + Separate UI + Server Logic

- Create a polished chat UI.
- Call server-side AI functions asynchronously.
- Handle errors gracefully.
- Display rich AI responses.

Lesson 4A–C — Full RAG System

You'll expand your project into a full **Retrieval-Augmented Generation** engine:

4A — RAG Chunking & Embedding

- Chunk large documents into small pieces.
- Create embeddings for every chunk.
- Store them efficiently.

4B — RAG Index in Google Sheets

- Build a complete vector index in Sheets.
- Store chunk metadata + embeddings.
- Reload the index instantly for future queries.

4C — Chat Memory + Source Debugging

- Add short-term conversational memory via Apps Script Cache.
- Display the document sources used to answer each question.
- Create a more transparent, explainable assistant.

Lesson 5 — Resettable Chat + Full Debugging

- Add UI-level “Reset Conversation” button.
- Provide sources for every answer.
- Build a production-grade web chat interface.

Lesson 6 — Multi-Workspace AI

- Manage multiple folders (workspaces) as separate knowledge domains.
- Auto-switch knowledge based on user selection.
- Build a high-level intelligent router for content.

💡 Why This Guide Is Different

This guide doesn’t just give you snippets — it gives you *full projects*:

✔ Complete Apps Script projects
✔ HTML & JavaScript Web App interfaces
✔ RAG logic using Gemini embeddings
✔ Debugging tools and source transparency

✔ Real-world examples and tested lessons
✔ GitHub-ready project structures

You'll finish with a professional-grade AI assistant you can deploy at work, in your classroom, or inside your organization.

🧠 What You Will Learn

By the end of this guide, you'll understand:

1. How AI Models Use Context

- How Gemini interprets prompts
- What embedding vectors are
- How retrieval boosts factual accuracy

2. How to Use Google Apps Script as an AI Platform

- Building secure backends
- Caching and optimizing results
- Managing large text sources

3. How to Build Intelligent Web Apps

- Interactive chatbot UIs
- Connecting client-side JS → server-side Apps Script
- Real-time updates and state management

4. How to Design Scalable RAG Systems

- Chunking strategy

- Vector similarity search
- Multi-workspace architecture

5. How to Work with Public APIs

- Structuring API wrappers
- Handling errors safely
- Feeding API data into AI prompts

🛠️ Tools You'll Use

- **Apps Script** (JavaScript)
- **Gemini API** (text + embedding models)
- **HTML Service** (for web apps)
- **Google Sheets** (as a vector index)
- **Google Drive** (document sources)
- **CacheService** (chat memory)
- **Free Public APIs** (cat facts, weather, jokes, D&D, etc.)

📂 Folder Structure (Example)

Each lesson includes a GitHub-ready folder:

```
lesson1_api_basics/
```

```
lesson2_api_plus_docs/

lesson3_webapp_ui/

lesson4a_rag_embeddings/

lesson4b_rag_sheet_index/

lesson4c_chat_history_sources/

lesson5_debug_reset/

lesson6_multi_workspace_rag/
```

Each folder includes:

- `Code.gs`
- `Index.html` (where applicable)
- `README.md`

🎯 Who This Guide Is For

This guide is designed for:

- Educators teaching AI + Workspace automation
- Developers who want to build document-aware AI chatbots
- Anyone using Google Workspace professionally

- Students learning modern AI engineering fundamentals

No advanced programming skills are required — just basic JavaScript and a willingness to experiment.

📘 Lesson 1 — Getting Data from APIs with Google Apps Script

In this project, your Hybrid AI Orchestrator pulls in **real data** from the web using APIs and then lets AI (local model + Gemini) interpret that data.

To do that, we need to understand one key tool:

> **`UrlFetchApp` — the Apps Script class that lets you call APIs.**

In this lesson you'll learn:

- What an API is (quick recap)
- How to call an API using `UrlFetchApp.fetch()`
- How to handle JSON responses
- How to add headers (for APIs that need them)
- How to use **Script Properties** to swap APIs without changing code
- How our `fetchDataFromApi_()` helper fits into the bigger AI project

🔹 1. Quick Recap: What Is an API?

Think of an API as a **URL that gives you data instead of a web page.**

Example:

```
https://catfact.ninja/fact
```

If you open that in a browser, you'll see JSON:

```
{
  "fact": "Cats sleep 70% of their lives.",
  "length": 34
}
```

With Apps Script, we can call that URL with `UrlFetchApp.fetch()` and get the same JSON in code — and then pass it to AI.

🔹 2. Your First API Call in Apps Script

Create a new Apps Script project, and in `Code.gs` try this:

```
function testCatFactApi() {
  const url = 'https://catfact.ninja/fact';

  const res = UrlFetchApp.fetch(url);
  const text = res.getContentText();  // raw JSON string

  Logger.log('Raw response: ' + text);

  const data = JSON.parse(text);      // convert JSON string → object

  Logger.log('Cat fact: ' + data.fact);
}
```

Run `testCatFactApi()`:

- Open **Executions** or **View → Logs**
- You should see a random cat fact

This is the basic pattern we'll reuse:

1. `UrlFetchApp.fetch(url)`
2. `getContentText()`
3. `JSON.parse(...)`

🔹 3. Adding Headers (for APIs That Require Them)

Some APIs need extra info — for example, a header that says:

> "I want JSON, not HTML"

The **icanhazdadjoke** API is like that.

Example: Calling icanhazdadjoke with Apps Script

```
function testJokeApi() {
  const url = 'https://icanhazdadjoke.com/';

  const options = {
    method: 'get',
    headers: {
      Accept: 'application/json'
    },
    muteHttpExceptions: true
  };

  const res = UrlFetchApp.fetch(url, options);
  const code = res.getResponseCode();
  const body = res.getContentText();

  Logger.log('Status: ' + code);
  Logger.log('Body: ' + body);

  if (code >= 200 && code < 300) {
    const data = JSON.parse(body);
    Logger.log('Joke: ' + data.joke);
  }
}
```

> 🔍 `headers: { Accept: 'application/json' }`
> tells the API: "Please respond with JSON."

🔹 4. Using Script Properties to Configure the API URL

Instead of hard-coding the URL in your functions, you can store it in **Script Properties**.

That way:

- You can switch APIs (weather → cat facts → jokes)
- Without changing any code
- Perfect for demos & teaching

Step 1 – Add a Script Property

In Apps Script:

1. Click the **gear icon** (Project Settings)
2. Scroll to **Script properties** → **Add script property**
3. Add:
    - **Name:** `DATA_API_URL`
    - **Value:** e.g. `https://catfact.ninja/fact`

Step 2 – Read It in Code

Here's a reusable helper:

```
function fetchDataFromApi_() {
  const props = PropertiesService.getScriptProperties();
  const dataApiUrl = props.getProperty('DATA_API_URL');

  if (!dataApiUrl) {
    throw new Error('DATA_API_URL is not set in Script properties.');
  }

  const res = UrlFetchApp.fetch(dataApiUrl, { muteHttpExceptions: true
});
  const code = res.getResponseCode();
  const body = res.getContentText();
```

```
  if (code < 200 || code >= 300) {
    return {
      source: 'error',
      status: code,
      body
    };
  }

  try {
    return JSON.parse(body);
  } catch (e) {
    return {
      source: 'error',
      status: code,
      body,
      parseError: String(e)
    };
  }
}
```

Now, anywhere in your script, you can do:

```
function testDataApiProperty() {
  const data = fetchDataFromApi_();
  Logger.log(JSON.stringify(data, null, 2));
}
```

To switch APIs, just change **`DATA_API_URL`** in Script Properties — no code changes needed.

🔹 5. Plugging in Real Sample APIs

Here's how to use the free APIs we talked about, all via **`DATA_API_URL`**.

✅ Option A — Weather (Open-Meteo)

DATA_API_URL:

```
https://api.open-meteo.com/v1/forecast?latitude=43.65107&longitude=-79.347015&current_weather=true
```

Then:

```
function testWeather() {
  const data = fetchDataFromApi_();
  Logger.log(JSON.stringify(data, null, 2));

  const cw = data.current_weather;
  if (cw) {
    Logger.log('Temperature: ' + cw.temperature + '°C');
    Logger.log('Wind speed: ' + cw.windspeed + ' km/h');
  }
}
```

✅ Option B — Cat Facts

DATA_API_URL:

```
https://catfact.ninja/fact
```

Then:

```
function testCatFact() {
  const data = fetchDataFromApi_();
  Logger.log('Cat fact: ' + data.fact);
}
```

✅ Option C — Bored API

DATA_API_URL:

```
https://www.boredapi.com/api/activity
```

Then:

```
function testBoredApi() {
  const data = fetchDataFromApi_();
  Logger.log('Activity: ' + data.activity);
  Logger.log('Type: ' + data.type);
}
```

✅ Option D — Jokes (with Headers)

For `icanhazdadjoke`, you'll likely want a dedicated version that includes headers, like:

```
function fetchDataFromApi_() {
  const props = PropertiesService.getScriptProperties();
  const dataApiUrl = props.getProperty('DATA_API_URL') ||
'https://icanhazdadjoke.com/';

  const options = {
    method: 'get',
    headers: {
      Accept: 'application/json'
    },
    muteHttpExceptions: true
  };

  const res = UrlFetchApp.fetch(dataApiUrl, options);
  const code = res.getResponseCode();
  const body = res.getContentText();

  if (code < 200 || code >= 300) {
    return {
      source: 'error',
      status: code,
      body
    };
  }
```

```
  try {
    return JSON.parse(body);
  } catch (e) {
    return {
      source: 'error',
      status: code,
      body,
      parseError: String(e)
    };
  }
}
```

Then:

```
function testJoke() {
  const data = fetchDataFromApi_();
  Logger.log('Joke: ' + data.joke);
}
```

🔹 6. How This Fits Into the Hybrid AI Orchestrator

In your **Hybrid AI Orchestrator** project, the same `fetchDataFromApi_()` helper is used inside `chatHybrid(question)`:

```
function chatHybrid(question) {
  const q = (question || '').trim();
  if (!q) throw new Error('Question is empty.');

  // 1) Get data from API (this lesson)
  const data = fetchDataFromApi_();

  // 2) Ask local AI model to analyze it
  const localAnalysis = callLocalAi_(q, data);

  // 3) Ask Gemini to synthesize everything into a final answer
  const finalAnswer = callGeminiOrchestrator_(q, data, localAnalysis);
```

```
  return {
    answer: finalAnswer,
    raw: { data, localAnalysis }
  };
}
```

So this lesson is the foundation:

> **First, we learn how to get JSON data into Apps Script. Then, we plug that data into local AI + Gemini.**

🔹 7. Practice Tasks (Apps Script Exercises)

You can add these as end-of-lesson exercises in the blog:

🧪 Exercise 1 — Swap APIs Without Editing Code

1. Set `DATA_API_URL` to the **cat facts API**.
2. Run `testDataApiProperty()` and log the fact.
3. Change `DATA_API_URL` to the **Bored API**.
4. Run the same function — see how the data changes without touching code.

🧪 Exercise 2 — Build a Simple "Explain This" Function

```
function explainDataForAi_() {
  const data = fetchDataFromApi_();
  return JSON.stringify(data, null, 2);
}
```

Later, you'll feed `explainDataForAi_()` into Gemini as context.

🧪 Exercise 3 — Add Basic Error Handling

Update `fetchDataFromApi_()` to throw an error if `source === 'error'`, and handle that gracefully in `chatHybrid()`.

📘 Lesson 2 — Build an API + Drive + Gemini Web App (No Indexing, Just Context)

In **Lesson 1**, you learned how to:

- Call public APIs with `UrlFetchApp`
- Parse JSON in Apps Script
- Use Script Properties (`DATA_API_URL`) to swap APIs without changing code

Now in **Lesson 2**, we'll go one level up:

You'll build a **chat-style web app** where each question is answered using:

- Data from a **live API** (e.g., cat facts, weather, activities)
- Text from files in a **Google Drive folder**
- The **Gemini** model to combine everything into one answer

No embeddings, no indexing — just **direct context**:

- "Here's the API data"
- "Here's what's in the folder"
- "Here's the user's question"
- Gemini: "OK, let me respond."

🧩 What You'll Build

By the end of this lesson, you'll have a web app that:

1. Shows a **chat UI** (user + assistant bubbles).
2. When you send a message:
    - Reads all text from a specific Drive folder (Docs + text files).
    - Fetches JSON data from a public API (`DATA_API_URL`).
    - Builds a single prompt with:
        - Your question
        - The API JSON
        - The folder content
    - Sends that prompt to **Gemini**.
    - Returns a combined answer in the chat UI.

🔧 Step 1 – Project Setup

Create a new **Apps Script project**.

You'll need:

1. A **Gemini API key**
2. A **Drive folder** with some content:
    - One or more Google Docs
    - Optional `.txt` files
3. A public **JSON API URL**, e.g.:
    - `https://catfact.ninja/fact`
    - `https://www.boredapi.com/api/activity`

- An Open-Meteo weather URL

1.1 Set Script Properties

In Apps Script:

1. Click ⚙ **Project Settings**
2. Under **Script properties** → **Add script property** add:

- `GEMINI_API_KEY` → your Gemini API key
- `DATA_API_URL` → your chosen API URL

Example values:

- `GEMINI_API_KEY` = `AIza...` (whatever your key is)
- `DATA_API_URL` = `https://catfact.ninja/fact`

1.2 Get Your Folder ID

- Open your Drive folder in the browser
- The URL will look like:

```
https://drive.google.com/drive/folders/1AbCdEfGhIjKlMnOpQrStUvWxYz
```

Copy the long ID part (`1AbCdEfGhIjKlMnOpQrStUvWxYz`).

We'll plug that into the code as `DEFAULT_FOLDER_ID`.

🧠 Step 2 – Backend Logic (Code.gs)

Create a file named **`Code.gs`** and paste this full script.

✏️ Don’t forget to replace PASTE_YOUR_FOLDER_ID_HERE with your real folder ID.

```
/**
 * Lesson 2 — Combine API data + Drive folder content in a Gemini request
 *
 * What this file includes:
 *  - doGet() / chatWithFolderAndApi() → Web app entrypoints
 *  - fetchDataFromApi_()              → API helper (from Lesson 1)
 *  - getFolderText_()                 → Read text from a Drive folder
 *  - callGeminiWithContext_()         → Send question + folder + API data to Gemini
 *  - answerWithFolderAndApi()         → Main function used by the web app
 */

/**************************************
 * CONFIG
 **************************************/

// Replace with your own folder ID before use
const DEFAULT_FOLDER_ID = 'PASTE_YOUR_FOLDER_ID_HERE';

// Gemini model + output config
const GEMINI_MODEL = 'gemini-2.5-flash';
const MAX_OUTPUT_TOKENS = 768;

/**************************************
 * WEB APP ENTRYPOINTS
 **************************************/

/**
 * Serves the HTML UI for the web app.
 */
function doGet(e) {
  return HtmlService.createHtmlOutputFromFile('Index')
    .setTitle('API + Drive + Gemini Assistant');
}
```

```
/**
 * Called from the web UI.
 * Wraps answerWithFolderAndApi() and returns an object.
 */
function chatWithFolderAndApi(question) {
  const answer = answerWithFolderAndApi(question);
  return { answer: answer || '' };
}

/**************************************
 * MAIN ENTRY – LESSON 2 FUNCTION
 **************************************/

/**
 * Main function for Lesson 2.
 * Call this with a question. It will:
 *  - Get data from an API
 *  - Read text from a Drive folder
 *  - Send both, plus your question, to Gemini
 *  - Return Gemini's answer.
 *
 * Example:
 *   answerWithFolderAndApi(
 *     "Summarize what the docs say and relate it to the API data."
 *   );
 */
function answerWithFolderAndApi(question) {
  const q = (question || '').trim();
  if (!q) {
    throw new Error('Question is empty.');
  }

  // 1) Fetch data from the configured API (Lesson 1 helper)
  const apiData = fetchDataFromApi_();

  // 2) Read text content from the Drive folder
  const folderText = getFolderText_(DEFAULT_FOLDER_ID);
```

```
  // 3) Call Gemini with all context
  const answer = callGeminiWithContext_(q, apiData, folderText);

  return answer;
}

/**
 * Quick test function you can run from the editor.
 */
function testAnswerWithFolderAndApi() {
  const question =
    'Give me a short summary that connects the folder docs with the
API data.';
  const result = answerWithFolderAndApi(question);
  Logger.log('AI answer:\n' + result);
}

/**************************************
 * PART 1 — API HELPER (FROM LESSON 1)
 **************************************/

/**
 * Fetch data from the API specified in
 * Script Properties → DATA_API_URL.
 *
 * You can set DATA_API_URL to, for example:
 *  - https://catfact.ninja/fact
 *  - https://www.boredapi.com/api/activity
 *  - An Open-Meteo weather URL
 *
 * If you use an API that requires headers (for example,
 * https://icanhazdadjoke.com/ with Accept: application/json),
 * you can adapt this helper accordingly.
 */
function fetchDataFromApi_() {
  const props = PropertiesService.getScriptProperties();
  const dataApiUrl = props.getProperty('DATA_API_URL');
```

```
  if (!dataApiUrl) {
    throw new Error('DATA_API_URL is not set in Script properties.');
  }

  const res = UrlFetchApp.fetch(dataApiUrl, { muteHttpExceptions: true
});
  const code = res.getResponseCode();
  const body = res.getContentText();

  if (code < 200 || code >= 300) {
    return {
      source: 'error',
      status: code,
      body
    };
  }

  try {
    return JSON.parse(body);
  } catch (e) {
    return {
      source: 'error',
      status: code,
      body,
      parseError: String(e)
    };
  }
}

/**************************************
 * PART 2 — READ DRIVE FOLDER CONTENT
 **************************************/

/**
 * Read text from all supported files in a Drive folder.
 * Supports:
 *  - Google Docs
```

```
 *  - Plain text files
 *
 * For Lesson 2, we keep it simple:
 *  - Concatenate all text into a single long string.
 *  - If it's very long, we truncate to avoid hitting token limits.
 */
function getFolderText_(folderId) {
  if (!folderId) {
    throw new Error('Folder ID is missing.');
  }

  const folder = DriveApp.getFolderById(folderId);
  const files = folder.getFiles();

  let combined = '';

  while (files.hasNext()) {
    const file = files.next();
    const mime = file.getMimeType();
    let text = '';

    if (mime === MimeType.GOOGLE_DOCS) {
      text = DocumentApp.openById(file.getId()).getBody().getText();
    } else if (mime === MimeType.PLAIN_TEXT) {
      text = file.getBlob().getDataAsString();
    } else {
      // Skip unsupported formats for now (PDF, Sheets, etc.)
      continue;
    }

    text = (text || '').trim();
    if (!text) continue;

    combined += '\n\n=== FILE: ' + file.getName() + ' ===\n' + text;
  }

  // Optional: truncate very long content to avoid huge prompts
  const MAX_CHARS = 6000;
```

```
  if (combined.length > MAX_CHARS) {
    combined =
      combined.slice(0, MAX_CHARS) +
      '\n\n[... truncated folder content for prompt length ...]';
  }

  if (!combined) {
    combined = '[No readable text files found in the folder.]';
  }

  return combined;
}

/**************************************
 * PART 3 — GEMINI REQUEST WITH CONTEXT
 **************************************/

/**
 * Build a prompt that includes:
 *  - The user's question
 *  - Raw JSON data from the API
 *  - Text from the Drive folder
 *
 * Then send it to Gemini and return the answer text.
 */
function callGeminiWithContext_(question, apiData, folderText) {
  const prompt =
    'You are an AI assistant. You will receive three things:\n' +
    '1) A user question\n' +
    '2) Raw JSON data from an API\n' +
    '3) Text content from a Google Drive folder\n\n' +
    'Your job is to answer the user\'s question in a clear, helpful
way, ' +
    'using both the API data and the folder content whenever they are
relevant.\n' +
    'If some information is missing or unclear, mention that
politely.\n\n' +
    '--- USER QUESTION ---\n' +
```

```
    question +
    '\n\n' +
    '--- API DATA (JSON) ---\n' +
    JSON.stringify(apiData, null, 2) +
    '\n\n' +
    '--- FOLDER CONTENT ---\n' +
    folderText +
    '\n\n' +
    'Now write your answer below:';

  return callGeminiText_(prompt);
}

/**
 * Core Gemini text call using the REST API.
 */
function callGeminiText_(prompt) {
  const apiKey = getGeminiApiKey_();
  const url =
    'https://generativelanguage.googleapis.com/v1beta/models/' +
    GEMINI_MODEL +
    ':generateContent?key=' +
    apiKey;

  const payload = {
    contents: [{ role: 'user', parts: [{ text: prompt }] }],
    generationConfig: {
      temperature: 0.3,
      maxOutputTokens: MAX_OUTPUT_TOKENS
    }
  };

  const options = {
    method: 'post',
    contentType: 'application/json',
    muteHttpExceptions: true,
    payload: JSON.stringify(payload)
  };
```

```
  const res = UrlFetchApp.fetch(url, options);
  const data = JSON.parse(res.getContentText());
  const text =
    data &&
    data.candidates &&
    data.candidates[0] &&
    data.candidates[0].content &&
    data.candidates[0].content.parts &&
    data.candidates[0].content.parts[0].text;

  return text || 'No response from Gemini. Raw: ' +
res.getContentText();
}

/**************************************
 * UTIL – GEMINI API KEY
 **************************************/
function getGeminiApiKey_() {
  const key = PropertiesService.getScriptProperties().getProperty(
    'GEMINI_API_KEY'
  );
  if (!key) {
    throw new Error('Missing GEMINI_API_KEY in Script Properties.');
  }
  return key;
}

/**************************************
 * OPTIONAL – LESSON 1 TEST HELPERS
 **************************************/

/**
 * Uses DATA_API_URL from Script Properties and logs the full JSON.
 */
function testDataApiProperty() {
  const data = fetchDataFromApi_();
  Logger.log(JSON.stringify(data, null, 2));
```

```
}

/**
 * Weather example (if DATA_API_URL is an Open-Meteo URL).
 */
function testWeather() {
  const data = fetchDataFromApi_();
  Logger.log(JSON.stringify(data, null, 2));

  const cw = data.current_weather;
  if (cw) {
    Logger.log('Temperature: ' + cw.temperature + '°C');
    Logger.log('Wind speed: ' + cw.windspeed + ' km/h');
  }
}

/**
 * Cat fact example (if DATA_API_URL is https://catfact.ninja/fact).
 */
function testCatFact() {
  const data = fetchDataFromApi_();
  Logger.log('Cat fact: ' + data.fact);
}

/**
 * Joke example – if you want to use https://icanhazdadjoke.com/,
 * update fetchDataFromApi_() to send the proper Accept header.
 */
function testJoke() {
  const data = fetchDataFromApi_();
  Logger.log('Joke (or raw data): ' + JSON.stringify(data));
}
```

💬 Step 3 – Chat UI (Index.html)

Now, let's add the web UI.

In your Apps Script project:

- **File → New → HTML file**
- Name it: `Index`
- Paste this:

```
<!DOCTYPE html>
<html>
  <head>
    <base target="_top" />
    <meta charset="UTF-8" />
    <style>
      body {
        font-family: system-ui, -apple-system, BlinkMacSystemFont,
'Segoe UI',
          sans-serif;
        margin: 0;
        padding: 16px;
        background: #020617;
        color: #e5e7eb;
      }
      .app {
        max-width: 900px;
        margin: 0 auto;
        background: #020617;
        border-radius: 16px;
        padding: 16px;
        box-shadow: 0 10px 40px rgba(0, 0, 0, 0.6);
        border: 1px solid #1f2937;
        display: flex;
        flex-direction: column;
        height: calc(100vh - 32px);
        box-sizing: border-box;
      }
      h1 {
        font-size: 1.5rem;
        margin-top: 0;
```

```
  margin-bottom: 0.4rem;
}
.subtitle {
  font-size: 0.9rem;
  color: #9ca3af;
  margin-bottom: 0.75rem;
}
.messages {
  flex: 1;
  overflow-y: auto;
  padding: 8px;
  border-radius: 8px;
  background: #020617;
  font-size: 0.9rem;
  border: 1px solid #1f2937;
  margin-bottom: 8px;
}
.message {
  max-width: 80%;
  margin-bottom: 8px;
  padding: 8px 10px;
  border-radius: 12px;
  line-height: 1.4;
  white-space: pre-wrap;
}
.message.user {
  margin-left: auto;
  background: #0b1120;
  border: 1px solid #1d4ed8;
  text-align: right;
}
.message.bot {
  margin-right: auto;
  background: #020617;
  border: 1px solid #334155;
}
.message.system {
  margin: 4px auto 10px auto;
```

```
  background: #020617;
  border: 1px dashed #374151;
  font-size: 0.8rem;
  color: #9ca3af;
  text-align: center;
}
.bubble-label {
  display: block;
  font-size: 0.7rem;
  color: #9ca3af;
  margin-bottom: 2px;
}
.input-area {
  border-top: 1px solid #1f2937;
  padding-top: 8px;
}
.input-row {
  display: flex;
  gap: 8px;
}
textarea {
  flex: 1;
  resize: none;
  min-height: 46px;
  max-height: 110px;
  padding: 8px 10px;
  border-radius: 8px;
  border: 1px solid #374151;
  background: #020617;
  color: #e5e7eb;
  font-family: inherit;
  font-size: 0.9rem;
}
button {
  padding: 8px 12px;
  border-radius: 8px;
  border: none;
  background: #3b82f6;
```

```
  color: white;
  font-weight: 600;
  cursor: pointer;
  font-size: 0.9rem;
  white-space: nowrap;
}
button:disabled {
  opacity: 0.6;
  cursor: default;
}
.status {
  font-size: 0.8rem;
  color: #9ca3af;
  margin-top: 4px;
  min-height: 1.2em;
}
.status.error {
  color: #f97373;
}
.typing {
  display: inline-flex;
  align-items: center;
  gap: 4px;
}
.typing-dot {
  width: 4px;
  height: 4px;
  border-radius: 999px;
  background: #9ca3af;
  opacity: 0.7;
  animation: pulse 1s infinite ease-in-out;
}
.typing-dot:nth-child(2) {
  animation-delay: 0.15s;
}
.typing-dot:nth-child(3) {
  animation-delay: 0.3s;
}
```

```
      @keyframes pulse {
        0%, 100% { transform: translateY(0); opacity: 0.3; }
        50% { transform: translateY(-2px); opacity: 1; }
      }
    </style>
  </head>
  <body>
    <div class="app">
      <header>
        <h1>API + Drive + Gemini Assistant</h1>
        <div class="subtitle">
          Ask a question and the assistant will use both your
configured API
          (DATA_API_URL) and the text in your Drive folder to answer.
        </div>
      </header>

      <main>
        <div id="messages" class="messages"></div>

        <div class="input-area">
          <div class="input-row">
            <textarea
              id="userInput"
              placeholder="Ask a question… (Enter to send, Shift+Enter
for a new line)"
            ></textarea>
            <button id="sendBtn" onclick="onSend()">Send</button>
          </div>
          <div id="status" class="status"></div>
        </div>
      </main>
    </div>

    <script>
      const messagesEl = document.getElementById('messages');
      const userInputEl = document.getElementById('userInput');
      const sendBtn = document.getElementById('sendBtn');
```

```
const statusEl = document.getElementById('status');
let typingMessageEl = null;

function addMessage(text, sender) {
  const wrapper = document.createElement('div');
  wrapper.className = 'message ' + sender;

  if (sender === 'user' || sender === 'bot') {
    const label = document.createElement('span');
    label.className = 'bubble-label';
    label.textContent = sender === 'user' ? 'You' : 'Assistant';
    wrapper.appendChild(label);
  }

  const content = document.createElement('div');
  content.textContent = text;
  wrapper.appendChild(content);

  messagesEl.appendChild(wrapper);
  messagesEl.scrollTop = messagesEl.scrollHeight;
}

function addSystemMessage(text) {
  const wrapper = document.createElement('div');
  wrapper.className = 'message system';
  wrapper.textContent = text;
  messagesEl.appendChild(wrapper);
  messagesEl.scrollTop = messagesEl.scrollHeight;
}

function showTyping() {
  if (typingMessageEl) return;
  typingMessageEl = document.createElement('div');
  typingMessageEl.className = 'message bot';

  const label = document.createElement('span');
  label.className = 'bubble-label';
  label.textContent = 'Assistant';
```

```
  typingMessageEl.appendChild(label);

  const typing = document.createElement('div');
  typing.className = 'typing';
  typing.innerHTML =
    '<span class="typing-dot"></span>' +
    '<span class="typing-dot"></span>' +
    '<span class="typing-dot"></span>';
  typingMessageEl.appendChild(typing);

  messagesEl.appendChild(typingMessageEl);
  messagesEl.scrollTop = messagesEl.scrollHeight;
}

function hideTyping() {
  if (typingMessageEl && typingMessageEl.parentNode) {
    typingMessageEl.parentNode.removeChild(typingMessageEl);
  }
  typingMessageEl = null;
}

function setLoading(isLoading) {
  sendBtn.disabled = isLoading;
  sendBtn.textContent = isLoading ? 'Thinking…' : 'Send';
}

function showStatus(msg, isError) {
  statusEl.textContent = msg || '';
  statusEl.classList.toggle('error', !!isError);
}

function autoGrowTextarea() {
  userInputEl.style.height = 'auto';
  userInputEl.style.height = userInputEl.scrollHeight + 'px';
}

function onSend() {
  const text = userInputEl.value.trim();
```

```
        if (!text) {
          showStatus('Please enter a question.', true);
          return;
        }
        showStatus('', false);

        addMessage(text, 'user');
        userInputEl.value = '';
        autoGrowTextarea();

        setLoading(true);
        showTyping();

        google.script.run
          .withSuccessHandler(function (res) {
            setLoading(false);
            hideTyping();
            if (res && res.answer) {
              addMessage(res.answer, 'bot');
            } else {
              addMessage(
                'I could not generate an answer from the API and
folder content.',
                'bot'
              );
            }
          })
          .withFailureHandler(function (err) {
            setLoading(false);
            hideTyping();
            showStatus(err.message || String(err), true);
            addMessage('Sorry, something went wrong on the server.',
'bot');
          })
          .chatWithFolderAndApi(text);
      }

      document.addEventListener('DOMContentLoaded', function () {
```

```
        addSystemMessage(
          'This assistant uses both your configured API (via
DATA_API_URL) and ' +
            'the text from your Drive folder (DEFAULT_FOLDER_ID in
Code.gs). ' +
            'Ask a question to see how it combines both sources.'
        );

        userInputEl.addEventListener('input', autoGrowTextarea);
        userInputEl.addEventListener('keydown', function (e) {
          if (e.key === 'Enter' && !e.shiftKey) {
            e.preventDefault();
            onSend();
          }
        });

        autoGrowTextarea();
      });
    </script>
  </body>
</html>
```

🚀 Step 4 – Deploy the Web App

1. In Apps Script, click **Deploy** → **New deployment**
2. Choose **Web app**
3. Set:
    - **Execute as:** Me
    - **Who has access:** Anyone with the link (or your choice)
4. Click **Deploy** and copy the URL

Open the URL in your browser:

- Type a question
- The app will:
    - Fetch API JSON from `DATA_API_URL`
    - Read Docs/text from `DEFAULT_FOLDER_ID`
    - Ask Gemini to combine both and answer

🧪 Practice Ideas for Learners

You can add these as exercises at the end of Lesson 2:

1. **Change the API**
    - Switch `DATA_API_URL` between:
        - Cat facts (`https://catfact.ninja/fact`)
        - Bored API (`https://www.boredapi.com/api/activity`)
        - Weather (Open-Meteo)
    - Ask: “Summarize the folder and weave in today’s weather/cat fact/activity.”
2. **Filter Folder Files**
    - Update `getFolderText_()` to only include files whose name contains `"Guide"` or `"Notes"`.
3. **Show Debug Information**
    - Log `apiData` and `folderText` before the Gemini call.
    - Add a dev-only toggle to see the raw JSON/folder text.

📘 Lesson 3 — Adding Embeddings + Retrieval (RAG) to Your Gemini + API Project

In Lessons 1 and 2, you learned how to:

- Call **public APIs** using Apps Script
- Read text from a **Google Drive folder**
- Combine both sources into a **Gemini prompt**
- Build a full UI web app

Now it's time to level up.

In **Lesson 3**, we introduce:

- **Chunking** (splitting documents into manageable pieces)
- **Embeddings** using Gemini
- **Cosine similarity**
- **Retrieval-Augmented Generation (RAG)**
- **Using cached embeddings** for speed

This allows your assistant to **search your Drive folder intelligently**, returning only the most relevant parts to Gemini — resulting in better, faster answers.

🎯 What You Will Build

By the end of Lesson 3, your assistant will:

1. Read all text from a Drive folder

2. Break each file into chunks
3. Convert each chunk into a **vector embedding**
4. Compute similarity between your question and each chunk
5. Retrieve only the top-K most relevant chunks
6. Send *only those chunks* to Gemini
7. Produce a precise, grounded answer

This is the same technique used in professional RAG systems such as:

- ChatGPT Retrieval
- Gemini File Bench
- LLM-powered search engines

You'll now have your own miniature version — in pure Google Apps Script.

🧠 Part 1 — Why Use Embeddings?

Before today, your assistant worked like this:

> "Dump the entire folder text into one giant prompt."

This works — but:

- Large folders exceed prompt limits
- Gemini must analyze irrelevant text
- Answers may be vague or slow

Embeddings solve this.

When you create an embedding, Gemini converts your text into a **vector** (a list of numbers) that represents its meaning.

Then you can compare embeddings using **cosine similarity**, which tells you how related two pieces of text are.

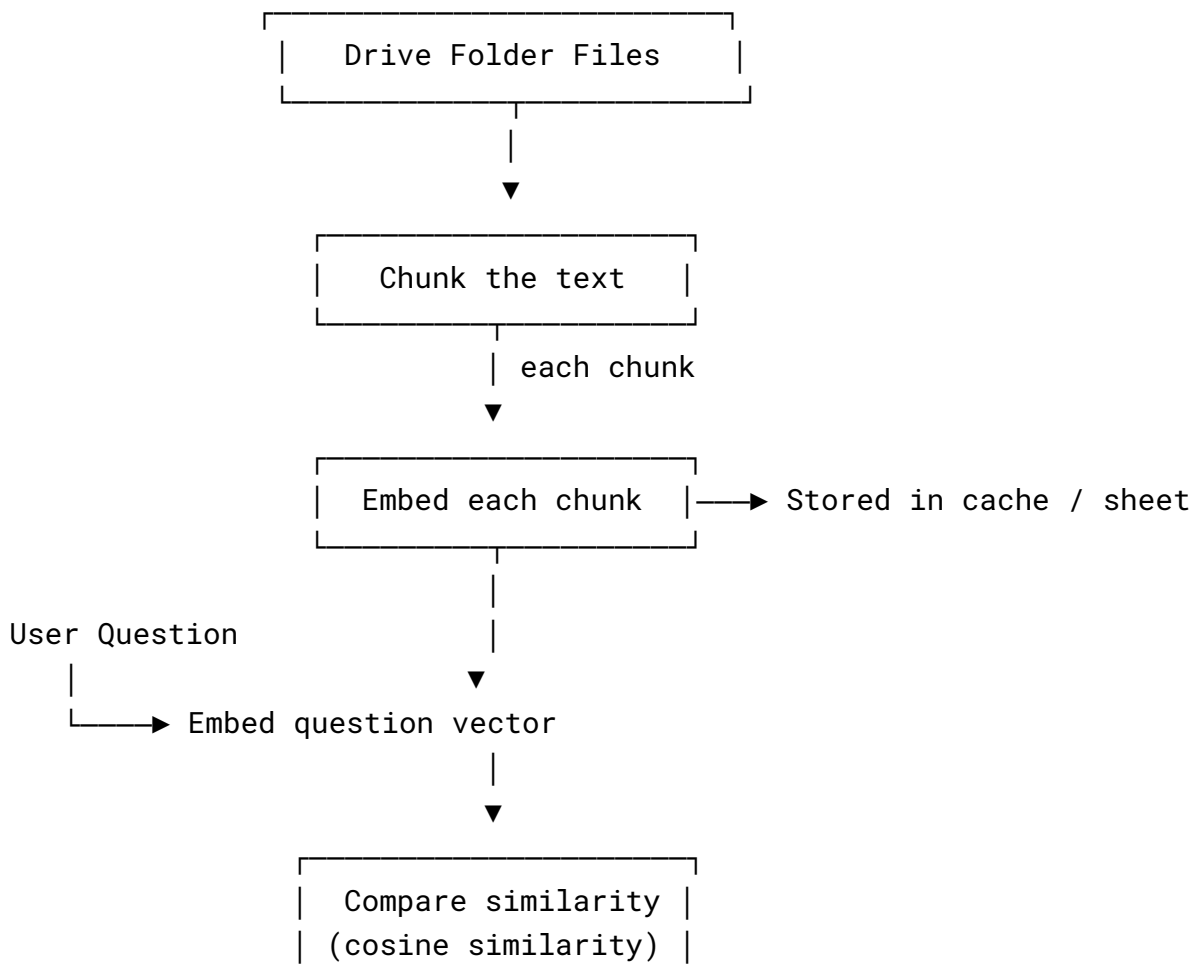
Think of it as:

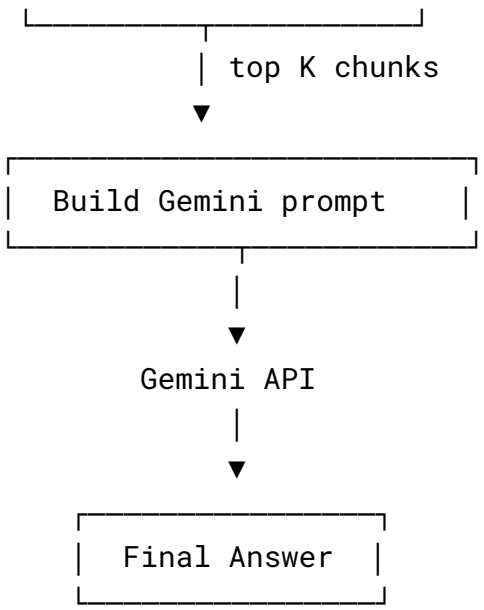
> "Find the most relevant paragraphs before asking Gemini."

This is dramatically faster and more accurate.

🧩 Part 2 — Overall Flow (Diagram)

Here's how the pipeline works:





This is real RAG — Retrieval-Augmented Generation — done entirely with Apps Script.

🔧 Part 3 — Full Lesson 3 Code (Apps Script)

Create a new file:

`Lesson3_RAG.gs`

Paste this **clean and optimized** version:

```
/**
 * Lesson 3 – Add Embeddings + Retrieval (RAG)
 *
 * What this file includes:
 *  - Chunking text
 *  - Creating embeddings for each chunk
 *  - Cosine similarity
 *  - Retrieving top-K relevant chunks
 *  - Sending ONLY relevant chunks to Gemini
```

```
 *
 * NOTE: Insert your folder ID in DEFAULT_FOLDER_ID.
 */

/**************************************
 * CONFIG
 **************************************/
const DEFAULT_FOLDER_ID = 'PASTE_YOUR_FOLDER_ID_HERE';
const EMBEDDING_MODEL = 'models/gemini-embedding-001';
const GENERATION_MODEL = 'gemini-2.5-flash';

const CHUNK_SIZE = 1200;   // characters per chunk
const TOP_K = 5;           // how many chunks to return
const MAX_OUTPUT_TOKENS = 1024;

/**************************************
 * MAIN ENTRY
 **************************************/
function answerWithRag(question) {
  question = question.trim();
  if (!question) {
    throw new Error('Question cannot be empty.');
  }

  // 1. Read & chunk the folder contents
  const chunks = getFolderChunks_(DEFAULT_FOLDER_ID);

  if (chunks.length === 0) {
    return 'No readable files found in the folder.';
  }

  // 2. Embed all chunks (with caching)
  const chunkEmbeddings = chunks.map(c => ({
    chunk: c,
    embedding: embedTextCached_(c.text)
  }));

  // 3. Embed the user's question
```

```
  const queryEmbedding = embedText_(question);

  // 4. Compute cosine similarity
  const ranked = chunkEmbeddings
    .map(item => ({
      chunk: item.chunk,
      score: cosineSimilarity_(queryEmbedding, item.embedding)
    }))
    .sort((a, b) => b.score - a.score);

  // 5. Take top-K chunks
  const selected = ranked.slice(0, TOP_K);

  // 6. Build the context payload
  const context = selected
    .map((c, i) =>
      `[#${i + 1} – ${c.chunk.fileName} –
score=${c.score.toFixed(3)}]\n${c.chunk.text}`
    )
    .join('\n\n');

  const prompt =
    'You are an AI assistant using retrieval. Use ONLY the information
below:\n\n' +
    '--- Retrieved Chunks ---\n' +
    context +
    '\n--- End Chunks ---\n\n' +
    'Question: ' + question + '\n' +
    'If the answer is not contained in the retrieved text, say you do
not know.';

  // 7. Generate final answer using Gemini
  return callGeminiText_(prompt);
}

/**************************************
 * STEP 1 – Read + Chunk Drive Files
 **************************************/
```

```
function getFolderChunks_(folderId) {
  const folder = DriveApp.getFolderById(folderId);
  const files = folder.getFiles();
  const chunks = [];

  while (files.hasNext()) {
    const file = files.next();
    let text = '';

    if (file.getMimeType() === MimeType.GOOGLE_DOCS) {
      text = DocumentApp.openById(file.getId()).getBody().getText();
    } else if (file.getMimeType() === MimeType.PLAIN_TEXT) {
      text = file.getBlob().getDataAsString();
    } else {
      continue;
    }

    text = text.trim();
    if (!text) continue;

    // split into fixed-size chunks
    let start = 0;
    let index = 0;

    while (start < text.length) {
      const end = Math.min(start + CHUNK_SIZE, text.length);
      chunks.push({
        id: file.getId() + ':' + index,
        fileName: file.getName(),
        text: text.substring(start, end)
      });
      start = end;
      index++;
    }
  }

  return chunks;
}
```

```
/**************************************
 * STEP 2 — Embeddings
 **************************************/
function embedText_(text) {
  const apiKey = getGeminiKey_();
  const url =
`https://generativelanguage.googleapis.com/v1beta/${EMBEDDING_MODEL}:e
mbedContent?key=${apiKey}`;

  const payload = {
    model: EMBEDDING_MODEL,
    content: { parts: [{ text }] },
    taskType: 'RETRIEVAL_DOCUMENT'
  };

  const res = UrlFetchApp.fetch(url, {
    method: 'post',
    contentType: 'application/json',
    payload: JSON.stringify(payload)
  });

  const data = JSON.parse(res.getContentText());
  return data.embedding?.values || data.embeddings?.[0]?.values;
}

/**
 * Cached version — stores embeddings in script cache.
 * Speeds up repeated queries by avoiding recomputation.
 */
function embedTextCached_(text) {
  const cache = CacheService.getScriptCache();
  const key = 'embed_' + Utilities.base64Encode(text).slice(0, 200);

  const cached = cache.get(key);
  if (cached) return JSON.parse(cached);

  const embedding = embedText_(text);
```

```
  cache.put(key, JSON.stringify(embedding), 21600); // cache 6 hours

  return embedding;
}

/**************************************
 * STEP 3 – Cosine Similarity
 **************************************/
function cosineSimilarity_(a, b) {
  let dot = 0,
    magA = 0,
    magB = 0;

  for (let i = 0; i < a.length; i++) {
    dot += a[i] * b[i];
    magA += a[i] * a[i];
    magB += b[i] * b[i];
  }
  return dot / (Math.sqrt(magA) * Math.sqrt(magB));
}

/**************************************
 * STEP 4 – Gemini Text Call
 **************************************/
function callGeminiText_(prompt) {
  const apiKey = getGeminiKey_();
  const url =
`https://generativelanguage.googleapis.com/v1beta/models/${GENERATION_
MODEL}:generateContent?key=${apiKey}`;

  const payload = {
    contents: [{ role: 'user', parts: [{ text: prompt }] }],
    generationConfig: { maxOutputTokens: MAX_OUTPUT_TOKENS }
  };

  const res = UrlFetchApp.fetch(url, {
    method: 'post',
    contentType: 'application/json',
```

```
    payload: JSON.stringify(payload)
  });

  const data = JSON.parse(res.getContentText());
  return data.candidates?.[0]?.content?.parts?.[0]?.text || '';
}

/**************************************
 * UTIL – API KEY
 **************************************/
function getGeminiKey_() {
  return PropertiesService.getScriptProperties().getProperty(
    'GEMINI_API_KEY'
  );
}
```

💬 How to Use Lesson 3

Run this in your Apps Script console:

```
Logger.log(answerWithRag("What does the folder say about
onboarding?"));
```

The script will:

- Read ALL Drive files
- Break them into chunks
- Embed each chunk
- Compare embeddings to your question
- Retrieve only the top-K
- Ask Gemini to answer using ONLY those chunks

This is a complete retrieval pipeline.

🧪 Practice Exercises for The Learner

1. **Change CHUNK_SIZE**
    - Try 500, 1200, 2000
    - What happens to accuracy?
2. **Make TOP_K adjustable**
    - Add query parameters like:
      `answerWithRag(question, topK)`
3. **Add PDF extraction**
    - Convert PDF → Google Doc → text
    - Chunk it the same way
4. **Cache the entire chunk list**
    - Store chunk/embedding arrays in `CacheService`
    - Dramatically reduces Drive reading time

📘 Lesson 4A — Combine RAG + API Data in a Single Gemini Answer

So far you've built:

- **Lesson 1** – Call public APIs from Apps Script (`DATA_API_URL`)
- **Lesson 2** – Combine **API data + folder text** in a Gemini prompt
- **Lesson 3** – Add **embeddings + cosine similarity** to create a basic RAG system over a Drive folder

Now in **Lesson 4A**, you'll:

> Use **RAG over the folder** *and* include **live API data** in the same Gemini request.

This lets your assistant answer questions like:

> "Using the onboarding notes in the folder and today's weather from the API, suggest a welcoming message for new students."

The answer will be:

- Grounded in the **most relevant chunks** of your Drive documents
- Enriched with **up-to-date information** from your chosen API

🧠 High-Level Flow

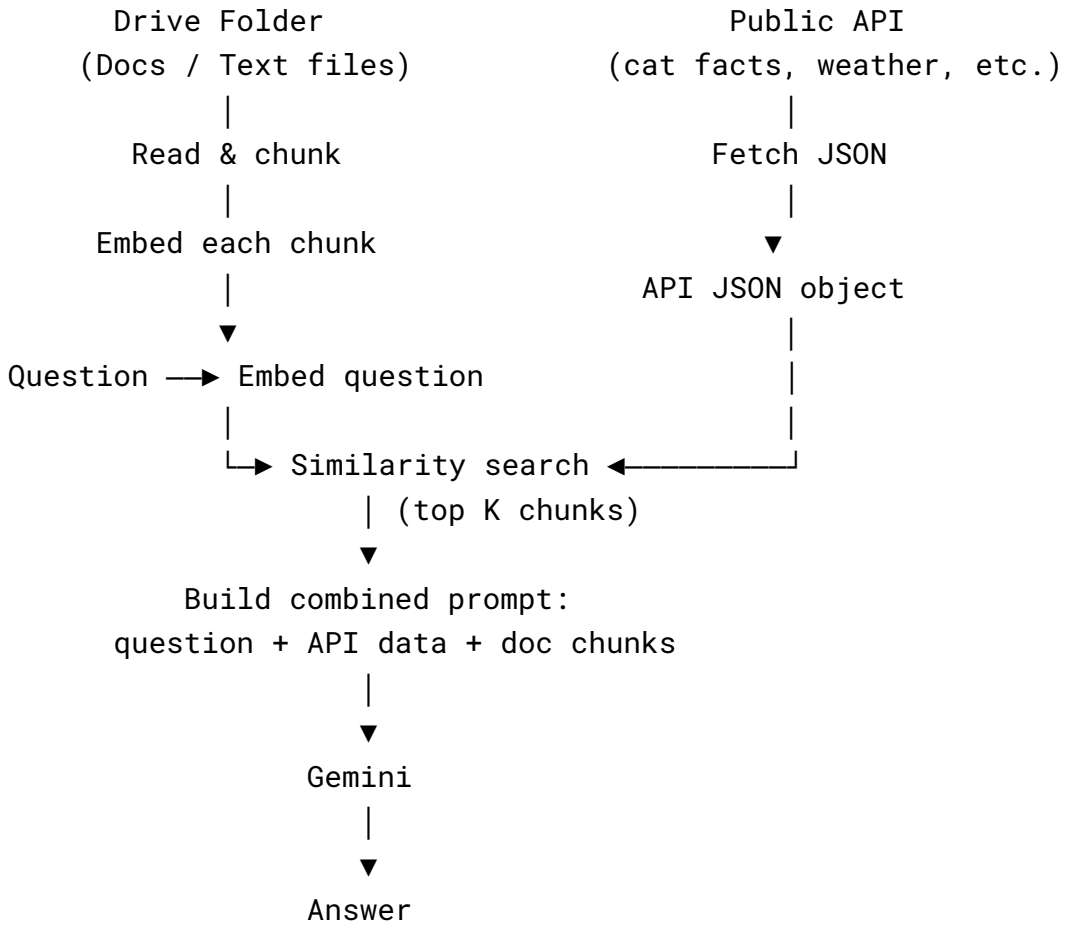
Here's what changes compared to Lesson 3:

1. **RAG part (same as Lesson 3):**
    - Read folder → chunk → embed chunks → embed question → cosine similarity → top-K chunks
2. **API part (from Lesson 1 & 2):**

- Call `DATA_API_URL`
- Parse the JSON

3. **Gemini prompt:**
   - Include:
     - The user's question
     - API JSON
     - Retrieved top-K document chunks
   - Ask Gemini to "use both the API data and the retrieved text" to answer.

Diagram:



🔧 Full Lesson 4A Code (Apps Script)

Create a new file, for example:

`Lesson4A_RagAndApi.gs`

Paste this full script.
(You can keep Lessons 2 & 3 in separate files if you like; this version is self-contained.)

> ✏️ **Important:** Replace `PASTE_YOUR_FOLDER_ID_HERE` with your real folder ID.

```
/**
 * Lesson 4A — Combine RAG (embeddings over Drive) + API data in one Gemini answer
 *
 * What this file does:
 *  - Reads & chunks text from a Drive folder
 *  - Creates embeddings for each chunk (with caching)
 *  - Computes cosine similarity to the user question
 *  - Retrieves the top-K relevant chunks (RAG)
 *  - Fetches JSON from an external API (DATA_API_URL)
 *  - Sends question + API JSON + retrieved chunks to Gemini
 *  - Returns a single, combined answer
 */

/**************************************
 * CONFIG
 **************************************/

// 1) Replace with your own Drive folder ID
const DEFAULT_FOLDER_ID = 'PASTE_YOUR_FOLDER_ID_HERE';

// 2) Gemini models
const EMBEDDING_MODEL = 'models/gemini-embedding-001';
const GENERATION_MODEL = 'gemini-2.5-flash';

// 3) RAG parameters
const CHUNK_SIZE = 1200;   // characters per chunk
```

```
const TOP_K = 5;           // how many chunks to keep
const MAX_OUTPUT_TOKENS = 1024;

/**************************************
 * MAIN ENTRY – LESSON 4A
 **************************************/

/**
 * Main function for Lesson 4A.
 *
 * You can call this directly from the Apps Script editor:
 *
 *   Logger.log(
 *     answerWithRagAndApi("Explain how the docs relate to the API
data.")
 *   );
 */
function answerWithRagAndApi(question) {
  const q = (question || '').trim();
  if (!q) {
    throw new Error('Question cannot be empty.');
  }

  // 1) RAG over Drive folder
  const chunks = getFolderChunks_(DEFAULT_FOLDER_ID);
  if (chunks.length === 0) {
    return 'No readable files found in the folder.';
  }

  // Embed all chunks (with caching)
  const chunkEmbeddings = chunks.map(c => ({
    chunk: c,
    embedding: embedTextCached_(c.text)
  }));

  // Embed the question
  const queryEmbedding = embedText_(q);
```

```
  // Rank chunks by cosine similarity
  const ranked = chunkEmbeddings
    .map(item => ({
      chunk: item.chunk,
      score: cosineSimilarity_(queryEmbedding, item.embedding)
    }))
    .sort((a, b) => b.score - a.score);

  // Take top-K
  const selected = ranked.slice(0, TOP_K);

  // Build RAG context
  const ragContext = selected
    .map((c, i) =>
      `[#${i + 1} – ${c.chunk.fileName} –
score=${c.score.toFixed(3)}]\n` +
      c.chunk.text
    )
    .join('\n\n');

  // 2) Fetch external API data
  const apiData = fetchDataFromApi_();
  const apiJsonPretty = JSON.stringify(apiData, null, 2);

  // 3) Build combined prompt
  const prompt =
    'You are an AI assistant that must answer using BOTH of the
following sources:\n' +
    '1) JSON data from an external API\n' +
    '2) Retrieved text chunks from a Google Drive folder (RAG)\n\n' +
    'Guidelines:\n' +
    '- Use the API data for up-to-date or numeric information.\n' +
    '- Use the retrieved chunks for policies, descriptions, and longer
explanations.\n' +
    '- If information is missing or unclear, say so politely.\n\n' +
    '--- USER QUESTION ---\n' +
    q + '\n\n' +
    '--- API DATA (JSON) ---\n' +
```

```
    apiJsonPretty + '\n\n' +
    '--- RETRIEVED DOCUMENT CHUNKS ---\n' +
    ragContext + '\n\n' +
    'Now write a single, clear answer that blends the API information
and the document content where relevant.';

  // 4) Call Gemini for the final answer
  return callGeminiText_(prompt);
}

/**
 * Helper for quick tests from the editor.
 */
function testAnswerWithRagAndApi() {
  const question =
    'Using both the document content and the API data, ' +
    'summarize the key ideas and give one practical suggestion.';
  const ans = answerWithRagAndApi(question);
  Logger.log('AI Answer:\n' + ans);
}

/**************************************
 * PART 1 — READ + CHUNK DRIVE FILES
 **************************************/

/**
 * Read the folder and split files into fixed-size chunks.
 * Supports:
 *  - Google Docs
 *  - Plain text files
 */
function getFolderChunks_(folderId) {
  if (!folderId) {
    throw new Error('Folder ID is missing.');
  }

  const folder = DriveApp.getFolderById(folderId);
  const files = folder.getFiles();
```

```
const chunks = [];

while (files.hasNext()) {
  const file = files.next();
  const mime = file.getMimeType();
  let text = '';

  if (mime === MimeType.GOOGLE_DOCS) {
    text = DocumentApp.openById(file.getId()).getBody().getText();
  } else if (mime === MimeType.PLAIN_TEXT) {
    text = file.getBlob().getDataAsString();
  } else {
    // Skip PDFs, Sheets, Slides, etc. for now
    continue;
  }

  text = (text || '').trim();
  if (!text) continue;

  // Split into fixed-size chunks
  let start = 0;
  let index = 0;

  while (start < text.length) {
    const end = Math.min(start + CHUNK_SIZE, text.length);
    const chunkText = text.substring(start, end);

    chunks.push({
      id: file.getId() + ':' + index,
      fileName: file.getName(),
      text: chunkText
    });

    start = end;
    index++;
  }
}
```

```
  return chunks;
}

/**************************************
 * PART 2 — EMBEDDINGS (WITH CACHE)
 **************************************/

/**
 * Low-level call to the Gemini embedding API.
 */
function embedText_(text) {
  const apiKey = getGeminiKey_();
  const url =

`https://generativelanguage.googleapis.com/v1beta/${EMBEDDING_MODEL}:e
mbedContent?key=${apiKey}`;

  const payload = {
    model: EMBEDDING_MODEL,
    content: { parts: [{ text }] },
    taskType: 'RETRIEVAL_DOCUMENT'
  };

  const res = UrlFetchApp.fetch(url, {
    method: 'post',
    contentType: 'application/json',
    payload: JSON.stringify(payload)
  });

  const data = JSON.parse(res.getContentText());
  return data.embedding?.values || data.embeddings?.[0]?.values;
}

/**
 * Cached version — stores embeddings in Script Cache
 * to speed up repeated runs on the same text.
 */
function embedTextCached_(text) {
```

```
  const cache = CacheService.getScriptCache();
  const key = 'embed_' + Utilities.base64Encode(text).slice(0, 200);

  const cached = cache.get(key);
  if (cached) {
    return JSON.parse(cached);
  }

  const embedding = embedText_(text);
  // Cache for 6 hours (in seconds)
  cache.put(key, JSON.stringify(embedding), 21600);

  return embedding;
}

/**************************************
 * PART 3 — COSINE SIMILARITY
 **************************************/

/**
 * Measures how similar two embedding vectors are.
 * 1.0  = very similar
 * 0.0  = not similar
 */
function cosineSimilarity_(a, b) {
  if (!a || !b || a.length !== b.length) return 0;

  let dot = 0;
  let magA = 0;
  let magB = 0;

  for (let i = 0; i < a.length; i++) {
    dot += a[i] * b[i];
    magA += a[i] * a[i];
    magB += b[i] * b[i];
  }

  if (magA === 0 || magB === 0) return 0;
```

```
  return dot / (Math.sqrt(magA) * Math.sqrt(magB));
}

/**************************************
 * PART 4 — GEMINI TEXT CALL
 **************************************/
function callGeminiText_(prompt) {
  const apiKey = getGeminiKey_();
  const url =

`https://generativelanguage.googleapis.com/v1beta/models/${GENERATION_
MODEL}:generateContent?key=${apiKey}`;

  const payload = {
    contents: [{ role: 'user', parts: [{ text: prompt }] }],
    generationConfig: {
      maxOutputTokens: MAX_OUTPUT_TOKENS,
      temperature: 0.3
    }
  };

  const res = UrlFetchApp.fetch(url, {
    method: 'post',
    contentType: 'application/json',
    payload: JSON.stringify(payload)
  });

  const data = JSON.parse(res.getContentText());
  return data.candidates?.[0]?.content?.parts?.[0]?.text || '';
}

/**************************************
 * PART 5 — API HELPER (FROM LESSON 1/2)
 **************************************/

/**
```

```
 * Fetch data from the API specified in Script Properties →
DATA_API_URL.
 *
 * Examples you can use as DATA_API_URL:
 *  - https://catfact.ninja/fact
 *  - https://www.boredapi.com/api/activity
 *  - An Open-Meteo weather URL
 */
function fetchDataFromApi_() {
  const props = PropertiesService.getScriptProperties();
  const dataApiUrl = props.getProperty('DATA_API_URL');

  if (!dataApiUrl) {
    throw new Error('DATA_API_URL is not set in Script Properties.');
  }

  const res = UrlFetchApp.fetch(dataApiUrl, { muteHttpExceptions: true
});
  const code = res.getResponseCode();
  const body = res.getContentText();

  if (code < 200 || code >= 300) {
    return {
      source: 'error',
      status: code,
      body
    };
  }

  try {
    return JSON.parse(body);
  } catch (e) {
    return {
      source: 'error',
      status: code,
      body,
      parseError: String(e)
    };
```

```
  }
}

/**************************************
 * UTIL – GEMINI API KEY
 **************************************/
function getGeminiKey_() {
  const key = PropertiesService.getScriptProperties().getProperty(
    'GEMINI_API_KEY'
  );
  if (!key) {
    throw new Error('Missing GEMINI_API_KEY in Script Properties.');
  }
  return key;
}
```

🔌 How to Hook This into Your Web App (Optional)

If you want to plug this into the **Lesson 2 chat UI**, just add this wrapper function in the same file:

```
function chatWithRagAndApi(question) {
  const answer = answerWithRagAndApi(question);
  return { answer: answer || '' };
}
```

Then, in your `Index.html` (web app UI), change:

```
google.script.run.chatWithFolderAndApi(text);
```

to:

```
google.script.run.chatWithRagAndApi(text);
```

Now your existing chatbot UI is powered by:

- **RAG over your Drive folder**
- **Live API data**
- **Gemini**

all working together.

📘 Lesson 4B — Store Embeddings in Google Sheets (DIY Vector Database)

In **Lesson 3**, you learned how to:

- Read & chunk text from a Drive folder
- Embed each chunk with Gemini
- Compare a question embedding to chunk embeddings using cosine similarity
- Retrieve the **top-K most relevant chunks** and send them to Gemini

But there was one limitation:

> Every time you ran it, you had to **rebuild embeddings in memory**.

For small folders that's fine, but as soon as you have more content, this becomes slow and wasteful.

In **Lesson 4B**, you'll fix that.

> You'll store your chunk embeddings in a **Google Sheet**, turning it into a simple, persistent **vector store**.

That means:

- You generate embeddings **once** (or only when content changes)
- You can query them as many times as you like
- You can inspect and debug your index visually in Sheets

🧠 Concept: Sheet as a Vector Database

We'll use a Sheet like this:

chunkId	fileId	fileName	chunkIndex	text	embeddingJson
1Abc...:0	1Abc…	Onboarding.md	0	“Welcome to …”	"[0.0123, -0.09, 0.45…]"
1Abc...:1	1Abc…	Onboarding.md	1	“In this course…”	"[0.083, 0.11, -0.02…]"
2Xyz...:0	2Xyz…	Policy.docx	0	“All students…”	"[0.004, -0.22, 0.19…]"

Each row = **one chunk** of text from your folder.

We’ll:

1. **Build (or rebuild) the index**
    - Read folder → chunk → embed each chunk
    - Write a row per chunk into a Sheet
2. **Query the index**
    - Load existing rows
    - Embed the user’s question
    - Compute cosine similarity to each row’s embedding
    - Take top-K rows
    - Build a prompt for Gemini

🔧 Setup Overview

We’ll introduce one new Script Property:

- `RAG_SHEET_ID` → the ID of the Spreadsheet that stores your index

If it doesn't exist yet, the code will **create the spreadsheet automatically** and save the ID into Script Properties for you.

You still need:

- `GEMINI_API_KEY` in Script Properties
- A Drive folder with Docs / text files
- A `DEFAULT_FOLDER_ID` in the script (like previous lessons)

🧩 Full Lesson 4B Code (Apps Script)

Create a new file in your project, e.g.:

`Lesson4B_SheetIndex.gs`

Paste this entire script:

```
/**
 * Lesson 4B — Store RAG Embeddings in Google Sheets
 *
 * What this file does:
 *  - Builds a persistent RAG index in a Spreadsheet
 *  - Each row = one chunk of text + its embedding vector
 *  - Allows you to run RAG queries without recomputing chunk
embeddings
 *
 * New:
 *  - getRagSheet_()         → Create/find index spreadsheet & sheet
 *  - buildRagIndex()        → Build or rebuild the full index
 *  - loadRagIndex_()        → Read chunks + embeddings from the Sheet
 *  - answerWithRagFromSheet(question)
 */

/**************************************
 * CONFIG
 **************************************/
```

```
// Folder whose contents will be indexed
const DEFAULT_FOLDER_ID = 'PASTE_YOUR_FOLDER_ID_HERE';

// Gemini models
const EMBEDDING_MODEL = 'models/gemini-embedding-001';
const GENERATION_MODEL = 'gemini-2.5-flash';

// RAG behavior
const CHUNK_SIZE = 1200;   // characters per chunk
const TOP_K = 5;           // how many chunks to retrieve per query
const MAX_OUTPUT_TOKENS = 1024;

// Sheet/index behavior
const RAG_SHEET_NAME = 'RAG_Index';       // tab name
const RAG_SHEET_PROP_KEY = 'RAG_SHEET_ID'; // Script property key

/**************************************
 * PUBLIC FUNCTIONS (YOU CALL THESE)
 **************************************/

/**
 * 1) Build (or rebuild) the RAG index in a Spreadsheet.
 *
 * Run this when:
 *  - You add/update documents in the folder
 *  - You want a fresh index
 */
function buildRagIndex() {
  const folderId = DEFAULT_FOLDER_ID;
  if (!folderId) {
    throw new Error('DEFAULT_FOLDER_ID is not set.');
  }

  const sheet = getRagSheet_();

  // Clear old data (keep header row)
  sheet.clearContents();
```

```
sheet.clearFormats();

// Set up headers
sheet.getRange(1, 1, 1, 6).setValues([[
  'chunkId',
  'fileId',
  'fileName',
  'chunkIndex',
  'text',
  'embeddingJson'
]]);

// 1) Read folder and chunk
const chunks = getFolderChunksForIndex_(folderId);
if (chunks.length === 0) {
  Logger.log('No readable content found in folder.');
  return;
}

// 2) Embed each chunk and prepare rows
const rows = [];
chunks.forEach(chunk => {
  const embedding = embedText_(chunk.text);
  rows.push([
    chunk.chunkId,
    chunk.fileId,
    chunk.fileName,
    chunk.chunkIndex,
    chunk.text,
    JSON.stringify(embedding)
  ]);
});

// 3) Write all rows at once
if (rows.length > 0) {
  sheet.getRange(2, 1, rows.length, 6).setValues(rows);
}
```

```
  Logger.log('RAG index built. Total chunks: ' + rows.length);
}

/**
 * 2) Answer questions using the Sheet index.
 *
 * This:
 *  - Loads existing chunks + embeddings from the Sheet
 *  - Embeds the question
 *  - Computes cosine similarity
 *  - Fetches top-K chunks
 *  - Calls Gemini with a RAG prompt
 */
function answerWithRagFromSheet(question) {
  const q = (question || '').trim();
  if (!q) throw new Error('Question cannot be empty.');

  // Load the index from Sheet
  const index = loadRagIndex_();
  if (index.length === 0) {
    throw new Error(
      'RAG index is empty. Run buildRagIndex() first to create it.'
    );
  }

  // Embed the question
  const queryEmbedding = embedText_(q);

  // Rank chunks by cosine similarity
  const ranked = index
    .map(entry => ({
      entry,
      score: cosineSimilarity_(queryEmbedding, entry.embedding)
    }))
    .sort((a, b) => b.score - a.score);

  const top = ranked.slice(0, TOP_K);
```

```
  const context = top.map((r, i) =>
    `[#${i + 1} – ${r.entry.fileName} –
score=${r.score.toFixed(3)}]\n` +
    r.entry.text
  ).join('\n\n');

  const prompt =
    'You are an AI assistant using a document index stored in
Sheets.\n' +
    'Use ONLY the text below to answer the user's question.\n\n' +
    '--- Retrieved Chunks ---\n' +
    context +
    '\n--- End Chunks ---\n\n' +
    'Question: ' + q + '\n' +
    'If the answer is not contained in the text, say that you do not
know.';

  return callGeminiText_(prompt);
}

/**
 * Quick test function: run this from the editor.
 */
function testAnswerWithRagFromSheet() {
  const question = 'What do these documents say about onboarding?';
  const answer = answerWithRagFromSheet(question);
  Logger.log('AI Answer:\n' + answer);
}

/**************************************
 * PART 1 – SHEET INDEX HELPERS
 **************************************/

/**
 * Get or create the Spreadsheet + Sheet used for the RAG index.
 *
 * - If RAG_SHEET_ID property exists → open that Spreadsheet.
 * - Otherwise → create one, save its ID in Script Properties.
```

```
 * - Ensures the RAG_Index sheet exists.
 */
function getRagSheet_() {
  const props = PropertiesService.getScriptProperties();
  let sheetId = props.getProperty(RAG_SHEET_PROP_KEY);
  let ss;

  if (sheetId) {
    // Reuse existing spreadsheet
    ss = SpreadsheetApp.openById(sheetId);
  } else {
    // Create a new spreadsheet just for the RAG index
    ss = SpreadsheetApp.create('Gemini RAG Index');
    sheetId = ss.getId();
    props.setProperty(RAG_SHEET_PROP_KEY, sheetId);
    Logger.log('Created new RAG index spreadsheet with ID: ' +
sheetId);
  }

  // Get or create the sheet
  let sheet = ss.getSheetByName(RAG_SHEET_NAME);
  if (!sheet) {
    sheet = ss.insertSheet(RAG_SHEET_NAME);
  }

  return sheet;
}

/**
 * Load chunk + embedding data from the RAG sheet.
 *
 * Returns an array of:
 *  {
 *    chunkId,
 *    fileId,
 *    fileName,
 *    chunkIndex,
 *    text,
```

```
 *    embedding: number[]
 *  }
 */
function loadRagIndex_() {
  const sheet = getRagSheet_();
  const range = sheet.getDataRange();
  const values = range.getValues();

  if (values.length <= 1) {
    return [];
  }

  const rows = values.slice(1); // skip header row

  const index = rows.map(row => {
    const [chunkId, fileId, fileName, chunkIndex, text, embeddingJson]
= row;
    let embedding = [];

    try {
      embedding = JSON.parse(embeddingJson || '[]');
    } catch (e) {
      embedding = [];
    }

    return {
      chunkId,
      fileId,
      fileName,
      chunkIndex,
      text,
      embedding
    };
  }).filter(entry => entry.embedding.length > 0);

  return index;
}
```

```
/**************************************
 * PART 2 — READ + CHUNK DRIVE FILES
 **************************************/

/**
 * Similar to Lesson 3, but returns richer metadata for each chunk
 * so we can store it in the Sheet.
 */
function getFolderChunksForIndex_(folderId) {
  const folder = DriveApp.getFolderById(folderId);
  const files = folder.getFiles();
  const chunks = [];

  while (files.hasNext()) {
    const file = files.next();
    const mime = file.getMimeType();
    let text = '';

    if (mime === MimeType.GOOGLE_DOCS) {
      text = DocumentApp.openById(file.getId()).getBody().getText();
    } else if (mime === MimeType.PLAIN_TEXT) {
      text = file.getBlob().getDataAsString();
    } else {
      // Skip unsupported file types for now
      continue;
    }

    text = (text || '').trim();
    if (!text) continue;

    let start = 0;
    let chunkIndex = 0;

    while (start < text.length) {
      const end = Math.min(start + CHUNK_SIZE, text.length);
      const chunkText = text.substring(start, end);

      chunks.push({
```

```
        chunkId: file.getId() + ':' + chunkIndex,
        fileId: file.getId(),
        fileName: file.getName(),
        chunkIndex,
        text: chunkText
      });

      start = end;
      chunkIndex++;
    }
  }

  return chunks;
}

/**************************************
 * PART 3 — EMBEDDINGS
 **************************************/

/**
 * Low-level Gemini embedding call.
 */
function embedText_(text) {
  const apiKey = getGeminiKey_();
  const url =

`https://generativelanguage.googleapis.com/v1beta/${EMBEDDING_MODEL}:e
mbedContent?key=${apiKey}`;

  const payload = {
    model: EMBEDDING_MODEL,
    content: { parts: [{ text }] },
    taskType: 'RETRIEVAL_DOCUMENT'
  };

  const res = UrlFetchApp.fetch(url, {
    method: 'post',
    contentType: 'application/json',
```

```
    payload: JSON.stringify(payload)
  });

  const data = JSON.parse(res.getContentText());
  return data.embedding?.values || data.embeddings?.[0]?.values || [];
}

/**************************************
 * PART 4 — COSINE SIMILARITY
 **************************************/
function cosineSimilarity_(a, b) {
  if (!a || !b || a.length === 0 || a.length !== b.length) return 0;

  let dot = 0;
  let magA = 0;
  let magB = 0;

  for (let i = 0; i < a.length; i++) {
    dot += a[i] * b[i];
    magA += a[i] * a[i];
    magB += b[i] * b[i];
  }

  if (magA === 0 || magB === 0) return 0;

  return dot / (Math.sqrt(magA) * Math.sqrt(magB));
}

/**************************************
 * PART 5 — GEMINI TEXT CALL
 **************************************/
function callGeminiText_(prompt) {
  const apiKey = getGeminiKey_();
  const url =

`https://generativelanguage.googleapis.com/v1beta/models/${GENERATION_
MODEL}:generateContent?key=${apiKey}`;
```

```
  const payload = {
    contents: [{ role: 'user', parts: [{ text: prompt }] }],
    generationConfig: {
      maxOutputTokens: MAX_OUTPUT_TOKENS,
      temperature: 0.3
    }
  };

  const res = UrlFetchApp.fetch(url, {
    method: 'post',
    contentType: 'application/json',
    payload: JSON.stringify(payload)
  });

  const data = JSON.parse(res.getContentText());
  return data.candidates?.[0]?.content?.parts?.[0]?.text || '';
}

/**************************************
 * UTIL — GEMINI API KEY
 **************************************/
function getGeminiKey_() {
  const key = PropertiesService.getScriptProperties().getProperty(
    'GEMINI_API_KEY'
  );
  if (!key) {
    throw new Error('Missing GEMINI_API_KEY in Script Properties.');
  }
  return key;
}
```

🧪 How to Use Lesson 4B

1. **Set your folder ID:**

```
const DEFAULT_FOLDER_ID = 'YOUR_REAL_FOLDER_ID_HERE';
```

2. Make sure **Script Properties** has:

- `GEMINI_API_KEY` set.

3. In the Apps Script editor, run:

```
buildRagIndex();
```

The script will:

- Create a new Spreadsheet called **"Gemini RAG Index"** (if it doesn't already exist)
- Add a `RAG_Index` sheet
- Fill it with chunks + embeddings

4. Then test a query:

```
testAnswerWithRagFromSheet();
```

Or directly:

```
Logger.log(
  answerWithRagFromSheet("Summarize the key onboarding information.")
);
```

You can now open the created spreadsheet from Drive and inspect your **vector index** visually.

🔁 Connecting to the Web App (Optional)

If you want to plug this into your existing chatbot UI:

Add this function:

```
function chatWithRagFromSheet(question) {
```

```
  const answer = answerWithRagFromSheet(question);
  return { answer: answer || '' };
}
```

Then in your `Index.html`, update the line that calls the server:

```
google.script.run.chatWithRagFromSheet(text);
```

Now your web app uses the **Sheet-backed index** instead of rebuilding embeddings each time.

📘 Lesson 4C — Add Chat History on Top of RAG (Conversation + Documents)

So far you've built:

- **Lesson 3** — RAG over a Drive folder (embeddings + similarity search)
- **Lesson 4B** — Stored those embeddings in a **Google Sheet** for a persistent index

But your assistant still behaves like this:

> "Each question is independent. It ignores what we said before."

In **Lesson 4C**, we'll change that:

> You'll give your assistant a **chat memory**, so it can:
>
> - Remember the last few questions and answers
> - Use them as context, *alongside* your RAG documents
> - Produce more natural, conversational replies

We'll still keep things simple and safe:

- Memory is short-term (last N turns)
- Stored in Apps Script **User Cache**
- No long-term database needed

🧠 What "Chat History" Means Here

We're not building a full chat server or multi-user system. Instead, for each user:

- We keep a **small list of recent turns**:
  `[{role: 'user', text: '...'}, {role: 'assistant', text: '...'},`

`...]`

- Each time the user asks a new question:
    1. We **load** the recent history
    2. We **run RAG** using the *current* question (so retrieval stays focused)
    3. We build a prompt that includes:
        - Short previous conversation
        - Retrieved chunks from the RAG index
        - The latest question
    4. We **append** the new Q&A to history

This gives you a nice balance:

- RAG keeps answers **grounded in documents**
- Chat history adds **context + continuity**

🔧 How We'll Store History

Apps Script gives us `CacheService`, which has:

- `getUserCache()` — cache tied to the active user
- Stores small strings (up to a few KB) for a time (e.g. 6 hours)

We'll:

- Serialize the history as JSON:
  `[{role, text}, ...]`

- Save it under a key like `"CHAT_HISTORY"`
- Limit history to the **last 8 turns** or so, to avoid huge prompts

---

🔩 Full Lesson 4C Code (Chat + RAG + Sheets)

This lesson assumes you **already have Lesson 4B** in your project (the Sheet index code).

We'll:

- Reuse:
  - `loadRagIndex_()`
  - `embedText_()`
  - `cosineSimilarity_()`
  - `callGeminiText_()`
  - `getGeminiKey_()`
- Add new functions for:
  - Chat history
  - Conversation-aware answer function

You can put this into a new file, e.g.:

```
Lesson4C_ChatRag.gs
```

```
/**
 * Lesson 4C – Chat History + RAG (Sheet Index)
 *
 * Builds on Lesson 4B:
 *  - Uses RAG index stored in a Google Sheet (RAG_Index tab)
 *  - Adds short-term chat memory using CacheService.getUserCache()
```

```
 *
 * New:
 *  - getChatHistory_(), saveChatHistory_()
 *  - addTurnToHistory_()
 *  - answerWithChatRag(question)
 *  - chatWithChatRag(question) → optional web app wrapper
 */

/**************************************
 * CONFIG (reuse from 4B if you like)
 **************************************/
const GENERATION_MODEL = 'gemini-2.5-flash';
const MAX_OUTPUT_TOKENS = 1024;
const TOP_K = 5;  // how many chunks to retrieve per query

// Chat history
const CHAT_HISTORY_KEY = 'CHAT_HISTORY';
const CHAT_HISTORY_MAX_TURNS = 8; // total messages (user+assistant)

/**************************************
 * PUBLIC ENTRY – CONVERSATION + RAG
 **************************************/

/**
 * Main function for Lesson 4C:
 * - Loads recent chat history (short-term memory)
 * - Loads RAG index from Sheet
 * - Does similarity search on current question
 * - Builds a prompt that includes:
 *    - chat history
 *    - retrieved document chunks
 *    - latest user question
 * - Calls Gemini and returns the final answer
 * - Updates chat history with the new exchange
 */
function answerWithChatRag(question) {
  const q = (question || '').trim();
  if (!q) throw new Error('Question cannot be empty.');
```

```
  // 1) Load existing RAG index from Sheet (from Lesson 4B)
  const index = loadRagIndex_();
  if (index.length === 0) {
    throw new Error('RAG index is empty. Run buildRagIndex() first.');
  }

  // 2) Load chat history from user cache
  const history = getChatHistory_(); // [{role, text}, ...]

  // 3) Embed the current question and retrieve top-K chunks
  const queryEmbedding = embedText_(q);

  const ranked = index
    .map(entry => ({
      entry,
      score: cosineSimilarity_(queryEmbedding, entry.embedding)
    }))
    .sort((a, b) => b.score - a.score);

  const top = ranked.slice(0, TOP_K);

  const docContext = top.map((r, i) =>
    `[#${i + 1} – ${r.entry.fileName} –
score=${r.score.toFixed(3)}]\n` +
    r.entry.text
  ).join('\n\n');

  // 4) Convert chat history to a readable string
  const conversationText = history.map(turn =>
    `${turn.role.toUpperCase()}: ${turn.text}`
  ).join('\n');

  // 5) Build the full prompt
  const prompt =
    'You are a helpful assistant that uses BOTH:\n' +
    '- The previous conversation with the user (chat history)\n' +
    '- Retrieved document chunks from a RAG index in Sheets\n\n' +
```

```
    'Rules:\n' +
    '- Use the chat history for context and follow-up
understanding.\n' +
    '- Use the retrieved document chunks for factual or policy
information.\n' +
    '- If the documents do not contain the answer, say you do not
know.\n' +
    '- Do NOT invent details that are not present in the
documents.\n\n' +
    '--- CHAT HISTORY ---\n' +
    (conversationText || '[No previous messages; this is the first
turn.]') +
    '\n\n' +
    '--- RETRIEVED DOCUMENT CHUNKS ---\n' +
    docContext +
    '\n--- END DOCUMENT CHUNKS ---\n\n' +
    'Now the user says:\n' +
    'USER: ' + q + '\n\n' +
    'Please answer as the ASSISTANT in a clear, concise way.';

  // 6) Call Gemini
  const answer = callGeminiText_(prompt);

  // 7) Update chat history with the new turn
  const newHistory = addTurnToHistory_(history, { role: 'user', text:
q });
  const newHistoryWithAssistant = addTurnToHistory_(newHistory, {
    role: 'assistant',
    text: answer
  });
  saveChatHistory_(newHistoryWithAssistant);

  return answer;
}

/**
 * Optional: wrapper for web app UI.
 * You can call this from your HTML via google.script.run.
```

```
 */
function chatWithChatRag(question) {
  const answer = answerWithChatRag(question);
  return { answer: answer || '' };
}

/**************************************
 * CHAT HISTORY HELPERS
 **************************************/

/**
 * Load chat history from the user cache.
 * Returns an array of {role: 'user'|'assistant', text: string}
 */
function getChatHistory_() {
  const cache = CacheService.getUserCache();
  const json = cache.get(CHAT_HISTORY_KEY);
  if (!json) return [];
  try {
    const history = JSON.parse(json);
    if (Array.isArray(history)) return history;
    return [];
  } catch (e) {
    return [];
  }
}

/**
 * Save chat history back into the user cache.
 */
function saveChatHistory_(history) {
  const cache = CacheService.getUserCache();
  cache.put(CHAT_HISTORY_KEY, JSON.stringify(history), 21600); // 6
hours
}

/**
 * Add a new turn to the history, keeping only the last N messages.
```

```
 */
function addTurnToHistory_(history, newTurn) {
  const updated = history.concat([newTurn]);
  const extra = Math.max(0, updated.length - CHAT_HISTORY_MAX_TURNS);
  if (extra > 0) {
    return updated.slice(extra);
  }
  return updated;
}

/**************************************
 * RAG HELPERS — REUSE FROM LESSON 4B
 **************************************/

/**
 * loadRagIndex_()
 * Reads existing chunks & embeddings from the RAG_Index sheet.
 * (Same as Lesson 4B — you can reuse your existing implementation.)
 *
 * Expected output:
 * [
 *   {
 *     chunkId,
 *     fileId,
 *     fileName,
 *     chunkIndex,
 *     text,
 *     embedding: number[]
 *   },
 *   ...
 * ]
 */
function loadRagIndex_() {
  const props = PropertiesService.getScriptProperties();
  const sheetId = props.getProperty('RAG_SHEET_ID');
  if (!sheetId) {
    throw new Error('RAG_SHEET_ID is not set. Run buildRagIndex()
first.');
```

```
  }

  const ss = SpreadsheetApp.openById(sheetId);
  const sheet = ss.getSheetByName('RAG_Index');
  if (!sheet) {
    throw new Error('RAG_Index sheet not found in the RAG
spreadsheet.');
  }

  const values = sheet.getDataRange().getValues();
  if (values.length <= 1) return [];

  const rows = values.slice(1); // skip header
  const index = rows.map(row => {
    const [chunkId, fileId, fileName, chunkIndex, text, embeddingJson]
= row;
    let embedding = [];
    try {
      embedding = JSON.parse(embeddingJson || '[]');
    } catch (e) {
      embedding = [];
    }
    return { chunkId, fileId, fileName, chunkIndex, text, embedding };
  }).filter(entry => entry.embedding.length > 0);

  return index;
}

/**************************************
 * EMBEDDINGS + COSINE SIMILARITY
 * (can be reused from previous lessons)
 **************************************/
const EMBEDDING_MODEL = 'models/gemini-embedding-001';

function embedText_(text) {
  const apiKey = getGeminiKey_();
  const url =
```

```
`https://generativelanguage.googleapis.com/v1beta/${EMBEDDING_MODEL}:embedContent?key=${apiKey}`;

  const payload = {
    model: EMBEDDING_MODEL,
    content: { parts: [{ text }] },
    taskType: 'RETRIEVAL_DOCUMENT'
  };

  const res = UrlFetchApp.fetch(url, {
    method: 'post',
    contentType: 'application/json',
    payload: JSON.stringify(payload)
  });

  const data = JSON.parse(res.getContentText());
  return data.embedding?.values || data.embeddings?.[0]?.values || [];
}

function cosineSimilarity_(a, b) {
  if (!a || !b || a.length === 0 || a.length !== b.length) return 0;

  let dot = 0;
  let magA = 0;
  let magB = 0;

  for (let i = 0; i < a.length; i++) {
    dot += a[i] * b[i];
    magA += a[i] * a[i];
    magB += b[i] * b[i];
  }

  if (magA === 0 || magB === 0) return 0;

  return dot / (Math.sqrt(magA) * Math.sqrt(magB));
}
```

```
/**************************************
 * GEMINI TEXT CALL + API KEY
 **************************************/
function callGeminiText_(prompt) {
  const apiKey = getGeminiKey_();
  const url =

`https://generativelanguage.googleapis.com/v1beta/models/${GENERATION_
MODEL}:generateContent?key=${apiKey}`;

  const payload = {
    contents: [{ role: 'user', parts: [{ text: prompt }] }],
    generationConfig: {
      maxOutputTokens: MAX_OUTPUT_TOKENS,
      temperature: 0.3
    }
  };

  const res = UrlFetchApp.fetch(url, {
    method: 'post',
    contentType: 'application/json',
    payload: JSON.stringify(payload)
  });

  const data = JSON.parse(res.getContentText());
  return data.candidates?.[0]?.content?.parts?.[0]?.text || '';
}

function getGeminiKey_() {
  const key = PropertiesService.getScriptProperties().getProperty(
    'GEMINI_API_KEY'
  );
  if (!key) {
    throw new Error('Missing GEMINI_API_KEY in Script Properties.');
  }
  return key;
}
```

🧪 How to Use Lesson 4C

Make sure you've already run **Lesson 4B**:

```
buildRagIndex(); // creates the sheet + embeddings
```

1.

Then test the new chat-aware function:

```
Logger.log(answerWithChatRag("Give me a quick summary of the documents."));
Logger.log(answerWithChatRag("Can you suggest one practical action from that?"));
Logger.log(answerWithChatRag("Rewrite that suggestion in a friendlier tone."));
```

2.

Because chat history is stored in `CacheService.getUserCache()`:

- Those three calls share the same short-term memory
- The third question is interpreted as a follow-up to the earlier answers

---

🔌 Hooking Into Your Web App

If your `Index.html` already calls something like:

```
google.script.run.chatWithRagFromSheet(text);
```

You can switch it to the chat-aware version:

```
google.script.run.chatWithChatRag(text);
```

No UI changes are required — the same chat window now has:

- RAG

- Sheet-based vector store
- AND short-term chat memory

📘 Lesson 5 — Reset Chat + Show Sources (Docs & Chunks) in the UI

By now you have:

- A **RAG index in Sheets** (Lesson 4B)
- **Chat history + RAG** working together (Lesson 4C)
- A **web app UI** (from Lesson 2) calling `google.script.run`

But two things are missing:

1. Users can't **reset** the conversation easily
2. You can't see **which documents** Gemini actually used

Lesson 5 fixes both:

- Add a **"Reset chat"** function on the server + button in the UI
- Extend the API to return **debug info (sources)** for each answer
- Show that info in the UI under the assistant messages

🧠 What We'll Add

1. On the server (Apps Script)

We'll:

Upgrade `answerWithChatRag()` so it returns:

```
{
  answer: "…",
  sources: [
    { fileName, chunkIndex, score, preview }
```

```
  ]
}
```

-
- Add `chatWithChatRagDebug(question)` that returns this object to the UI
- Add `resetChatHistoryPublic()` to clear chat history from the UI

2. On the client (Index.html)

We'll:

- Add a **"Reset conversation"** button at the top
- Update `onSend()` to use `chatWithChatRagDebug`
- When the assistant responds, show a small **"Sources"** block under the message with file names + scores

---

🧩 Part 1 — Update the Server: Chat + Sources + Reset

In your `Lesson4C_ChatRag.gs` file (or similar), we'll:

1. Replace `answerWithChatRag()` with a **version that returns `{answer, sources}`**
2. Adjust the `chatWithChatRag()` wrapper (or add a new one)
3. Add a public reset function

1.1 Updated `answerWithChatRag` (return answer + sources)

```
/**
 * Main function for Lesson 5:
 * - Same idea as Lesson 4C, but now returns:
 *    {
 *      answer: string,
 *      sources: [
```

```
 *        { fileName, chunkIndex, score, preview }
 *      ]
 *    }
 */
function answerWithChatRag(question) {
  const q = (question || '').trim();
  if (!q) throw new Error('Question cannot be empty.');

  // 1) Load RAG index (from Sheet)
  const index = loadRagIndex_();
  if (index.length === 0) {
    throw new Error('RAG index is empty. Run buildRagIndex() first.');
  }

  // 2) Load chat history
  const history = getChatHistory_(); // [{role, text}, ...]

  // 3) Embed the question and compute similarity
  const queryEmbedding = embedText_(q);

  const ranked = index
    .map(entry => ({
      entry,
      score: cosineSimilarity_(queryEmbedding, entry.embedding)
    }))
    .sort((a, b) => b.score - a.score);

  const top = ranked.slice(0, TOP_K);

  // 4) Build document context + source metadata
  const sources = top.map(r => {
    const preview = (r.entry.text || '').slice(0, 180).replace(/\s+/g,
' ') +
      (r.entry.text.length > 180 ? '…' : '');
    return {
      fileName: r.entry.fileName,
      chunkIndex: r.entry.chunkIndex,
      score: r.score,
```

```
      preview
    };
  });

  const docContext = top.map((r, i) =>
    `[#${i + 1} — ${r.entry.fileName} — chunk=${r.entry.chunkIndex} —
score=${r.score.toFixed(3)}]\n` +
    r.entry.text
  ).join('\n\n');

  // 5) Format chat history
  const conversationText = history.map(turn =>
    `${turn.role.toUpperCase()}: ${turn.text}`
  ).join('\n');

  // 6) Build prompt
  const prompt =
    'You are a helpful assistant that uses BOTH:\n' +
    '- The previous conversation with the user (chat history)\n' +
    '- Retrieved document chunks from a RAG index in Sheets\n\n' +
    'Rules:\n' +
    '- Use the chat history for follow-up understanding.\n' +
    '- Use the retrieved chunks for factual or policy information.\n'
+
    '- If the documents do not contain the answer, say you do not
know.\n' +
    '- Do NOT invent details that are not present in the
documents.\n\n' +
    '--- CHAT HISTORY ---\n' +
    (conversationText || '[No previous messages; this is the first
turn.]') +
    '\n\n' +
    '--- RETRIEVED DOCUMENT CHUNKS ---\n' +
    docContext +
    '\n--- END DOCUMENT CHUNKS ---\n\n' +
    'Now the user says:\n' +
    'USER: ' + q + '\n\n' +
    'Please answer as the ASSISTANT in a clear, concise way.';
```

```
  // 7) Call Gemini
  const answer = callGeminiText_(prompt);

  // 8) Update history (user + assistant)
  const newHistory = addTurnToHistory_(history, { role: 'user', text:
q });
  const newHistoryWithAssistant = addTurnToHistory_(newHistory, {
    role: 'assistant',
    text: answer
  });
  saveChatHistory_(newHistoryWithAssistant);

  // 9) Return both answer + sources
  return {
    answer: answer || '',
    sources
  };
}
```

1.2 New wrapper for UI: `chatWithChatRagDebug`

If you already have `chatWithChatRag`, you can either replace it or add a new one. For clarity, let's use a debug-friendly name:

```
/**
 * Web-app wrapper for Lesson 5.
 * The UI will call this via google.script.run.
 */
function chatWithChatRagDebug(question) {
  return answerWithChatRag(question); // already returns {answer,
sources}
}
```

1.3 Public function to reset chat history

We already have `getChatHistory_`, `saveChatHistory_`, etc.
Now we expose a **public** reset:

```
/**
 * Clears chat history for the current user.
 * Can be called from the web UI (Reset Conversation button).
 */
function resetChatHistoryPublic() {
  const cache = CacheService.getUserCache();
  cache.remove(CHAT_HISTORY_KEY);
  return { ok: true };
}
```

That's all you need server-side for Lesson 5.

🧩 Part 2 — Update the Web App UI

Now we'll adjust your `Index.html` to:

1. Add a **Reset** button
2. Use `chatWithChatRagDebug` instead of the old function
3. Render **Sources** under each assistant answer

I'll show a minimal version assuming you're using the chat UI style we built earlier (Lesson 2 / API + Drive assistant).

2.1 Add a "Reset" button in the header

Inside `<body>`, your main container probably looks like this:

```
<div class="app">
  <header>
    <h1>API + Drive + Gemini Assistant</h1>
    <div class="subtitle">
      Ask a question and the assistant will use your documents.
    </div>
  </header>
  ...
```

```
</div>
```

Update the header to include a small button on the right:

```
<header style="display:flex; justify-content:space-between;
align-items:center; gap:8px;">
  <div>
    <h1>Docs + RAG Chat Assistant</h1>
    <div class="subtitle">
      Ask questions grounded in your Drive documents with short-term
chat memory.
    </div>
  </div>
  <button id="resetBtn" onclick="onResetChat()"
style="font-size:0.8rem;">
    Reset conversation
  </button>
</header>
```

2.2 Update the message rendering to support “Sources”

In your `<script>` block, you probably had something like:

```
function addMessage(text, sender) {
  const wrapper = document.createElement('div');
  wrapper.className = 'message ' + sender;
  ...
}
```

Let’s extend it to optionally attach a “Sources” panel.

Replace your `addMessage` function with:

```
<script>
  const messagesEl = document.getElementById('messages');
  const userInputEl = document.getElementById('userInput');
  const sendBtn = document.getElementById('sendBtn');
  const statusEl = document.getElementById('status');
```

```
  let typingMessageEl = null;

  function addMessage(text, sender, sources) {
    const wrapper = document.createElement('div');
    wrapper.className = 'message ' + sender;

    // Label (You / Assistant)
    if (sender === 'user' || sender === 'bot') {
      const label = document.createElement('span');
      label.className = 'bubble-label';
      label.textContent = sender === 'user' ? 'You' : 'Assistant';
      wrapper.appendChild(label);
    }

    // Main text
    const content = document.createElement('div');
    content.textContent = text;
    wrapper.appendChild(content);

    // Optional sources block (only for bot messages with debug data)
    if (sender === 'bot' && Array.isArray(sources) && sources.length >
0) {
      const src = document.createElement('div');
      src.style.marginTop = '6px';
      src.style.paddingTop = '4px';
      src.style.borderTop = '1px solid #1f2937';
      src.style.fontSize = '0.75rem';
      src.style.color = '#9ca3af';

      const title = document.createElement('div');
      title.textContent = 'Sources used:';
      title.style.marginBottom = '2px';
      src.appendChild(title);

      const list = document.createElement('ul');
      list.style.margin = 0;
      list.style.paddingLeft = '16px';
```

```
      sources.forEach((s) => {
        const li = document.createElement('li');
        li.textContent =
          `${s.fileName} (chunk ${s.chunkIndex}, score
${s.score.toFixed(3)})` +
          (s.preview ? ` — "${s.preview}"` : '');
        list.appendChild(li);
      });

      src.appendChild(list);
      wrapper.appendChild(src);
    }

    messagesEl.appendChild(wrapper);
    messagesEl.scrollTop = messagesEl.scrollHeight;
  }

  // ... keep your typing, status, setLoading, etc.
</script>
```

2.3 Update onSend() to use the new server function

Change your previous onSend() (which probably called `chatWithFolderAndApi`, `chatWithRagFromSheet`, or `chatWithChatRag`) to now call `chatWithChatRagDebug`.

For example:

```
function onSend() {
  const text = userInputEl.value.trim();
  if (!text) {
    showStatus('Please enter a question.', true);
    return;
  }
  showStatus('', false);

  addMessage(text, 'user');
  userInputEl.value = '';
  autoGrowTextarea();
```

```
  setLoading(true);
  showTyping();

  google.script.run
    .withSuccessHandler(function (res) {
      setLoading(false);
      hideTyping();

      if (res && res.answer) {
        // res.sources may be undefined or empty
        addMessage(res.answer, 'bot', res.sources || []);
      } else {
        addMessage(
          'I could not generate an answer.',
          'bot'
        );
      }
    })
    .withFailureHandler(function (err) {
      setLoading(false);
      hideTyping();
      showStatus(err.message || String(err), true);
      addMessage('Sorry, something went wrong on the server.', 'bot');
    })
    .chatWithChatRagDebug(text);
}
```

2.4 Implement the Reset button behavior

Add a `onResetChat()` function in the same script:

```
function onResetChat() {
  setLoading(true);
  showStatus('Resetting conversation…', false);

  google.script.run
    .withSuccessHandler(function (res) {
      setLoading(false);
```

```
      showStatus('Conversation reset.', false);

      // Clear messages in the UI
      messagesEl.innerHTML = '';

      // Optionally add a system message
      addSystemMessage(
        'Conversation reset. Ask a new question to start a fresh
chat.'
      );
    })
    .withFailureHandler(function (err) {
      setLoading(false);
      showStatus(err.message || String(err), true);
    })
    .resetChatHistoryPublic();
}
```

Make sure you still have this helper (we used it earlier):

```
function addSystemMessage(text) {
  const wrapper = document.createElement('div');
  wrapper.className = 'message system';
  wrapper.textContent = text;
  messagesEl.appendChild(wrapper);
  messagesEl.scrollTop = messagesEl.scrollHeight;
}
```

🧪 Things to Try in Lesson 5

You can give learners a few practice tasks:

1. **Check the sources**
    - Ask a question about one specific document.
    - Confirm that the “Sources used” list only shows that file.

2. **See how scores change**
    - Ask a broad question → multiple files, similar scores
    - Ask a very specific question → one chunk with a much higher score
3. **Test reset**
    - Have a multi-step conversation
    - Click **Reset conversation**
    - Ask: “What did I just ask you before this?”
    - The assistant should say it doesn’t know (history cleared).
4. **Optional:** add a **“Show/Hide Sources”** toggle per message
    - Wrap the sources block in a `<details>` element
    - Let users expand/collapse to keep the UI clean.

📘 Lesson 6 — Multi-Folder Workspaces (Switch Between Knowledge Bases)

How to let your Apps Script + Gemini assistant dynamically switch between multiple folders (each with its own RAG index).

🔥 What Lesson 6 Adds

Until now, your assistant worked with exactly **one folder**, and one RAG index sheet.

Lesson 6 adds the ability to manage **multiple "workspaces"**, each with:

- A **Drive Folder**
- Its own **RAG Index** (a tab in the same spreadsheet OR multiple spreadsheets)
- A **name** (e.g. "Policies", "Project X", "Onboarding Docs")
- A **selectable workspace dropdown** in the web UI

This means your AI assistant can answer questions like:

> "Switch to the *Onboarding Docs* workspace."
> "Now answer using the *Policies* workspace."
> "Use the *Research Notes* workspace."

Each workspace becomes a *separate knowledge brain*.

🎯 Learning Objectives

By the end of Lesson 6, you will be able to:

✔ Allow your AI to switch between multiple Drive folders

✔ Maintain a RAG index per workspace

✔ **Add a workspace selector UI**

✔ **Load the correct embeddings for each workspace**

✔ **Keep chat history separate per workspace**

✔ **Enable “Reset per workspace”**

🧱 Lesson 6 Architecture Overview

```
+---------------------------+
|    Workspace Registry     |  <-- A JSON list saved in Script
Properties
+---------------------------+
        |           |
        |           |
        v           v
 Workspace A    Workspace B     Workspace C
 ------------   ------------    ------------
 - Folder ID    - Folder ID     - Folder ID
 - Index Sheet  - Index Sheet   - Index Sheet
 - Chat History - Chat History  - Chat History
```

Each workspace has:

```
{
  name: "Policies",
  folderId: "1xxxx",
  sheetId: "1xxxx (Spreadsheet for RAG index)"
}
```

Your assistant receives:

```
- Selected workspace
- Retrieves RAG index for that workspace
- Uses its Drive folder, embeddings, and chat history
- Answers
```

🧩 Part 1 — Store Workspaces in Script Properties

Add this new property:

`WORKSPACES`

Containing JSON:

```
[
  {
    "name": "Policies",
    "folderId": "xxxxx",
    "sheetId": "xxxxx"
  },
  {
    "name": "Onboarding",
    "folderId": "yyyyy",
    "sheetId": "yyyyy"
  }
]
```

Add helper functions into a new file named:

Lesson6_Workspaces.gs

✅ 1. Workspace Registry Helpers

```
const WORKSPACE_KEY = 'WORKSPACES';

/**
 * Returns array of workspaces:
 * [{name, folderId, sheetId}, ...]
```

```
 */
function getWorkspaces_() {
  const s =
PropertiesService.getScriptProperties().getProperty(WORKSPACE_KEY);
  if (!s) return [];
  try { return JSON.parse(s); }
  catch(e) { return []; }
}

/**
 * Saves workspace list.
 */
function saveWorkspaces_(arr) {
  PropertiesService.getScriptProperties()
    .setProperty(WORKSPACE_KEY, JSON.stringify(arr));
}

/**
 * Add a new workspace.
 */
function addWorkspace(name, folderId, sheetId) {
  const ws = getWorkspaces_();
  ws.push({ name, folderId, sheetId });
  saveWorkspaces_(ws);
  return ws;
}
```

🧩 Part 2 — Selecting a Workspace

We store the user's active workspace in the **User Cache**:

```
const ACTIVE_WORKSPACE_KEY = 'ACTIVE_WORKSPACE';

function setActiveWorkspace(name) {
  const ws = getWorkspaces_();
  const found = ws.find(w => w.name === name);
```

```
  if (!found) throw new Error("Workspace not found: " + name);

  CacheService.getUserCache().put(
    ACTIVE_WORKSPACE_KEY,
    JSON.stringify(found),
    3600
  );

  return found;
}

function getActiveWorkspace_() {
  const cache = CacheService.getUserCache();
  const raw = cache.get(ACTIVE_WORKSPACE_KEY);
  if (!raw) return null;
  try { return JSON.parse(raw); }
  catch(e) { return null; }
}
```

🧩 Part 3 — Updating Lesson 4C to Use Workspaces

Modify `answerWithChatRag` to load:

- workspace RAG index
- workspace chat history

```
function answerWithChatRagMulti(question) {
  const q = question.trim();
  if (!q) throw new Error("Question empty");

  const ws = getActiveWorkspace_();
  if (!ws) throw new Error("No active workspace selected.");
```

```
  // load index for THIS workspace
  const index = loadRagIndexForSheet_(ws.sheetId);

  // history is unique per workspace
  const historyKey = CHAT_HISTORY_KEY + "_" + ws.name;
  const history = getChatHistoryForKey_(historyKey);

  // everything else is the same as Lesson 4C
  // ... similarity search, Gemini call ...

  // save history under workspace-specific key
  saveChatHistoryForKey_(historyKey, updatedHistory);

  return {
    answer,
    sources,
    workspace: ws.name
  };
}
```

Add helper functions:

```
function getChatHistoryForKey_(key) {
  const cache = CacheService.getUserCache();
  const json = cache.get(key);
  if (!json) return [];
  try { return JSON.parse(json); }
  catch(e) { return []; }
}

function saveChatHistoryForKey_(key, history) {
  CacheService.getUserCache().put(key, JSON.stringify(history),
21600);
}
```

🧩 Part 4 — Updating the Web App (UI)

Add a **workspace dropdown**:

```
<select id="workspaceSelect" onchange="onWorkspaceChange()">
</select>
```

Populate it when the page loads:

```
function init() {
  google.script.run
    .withSuccessHandler(function (ws) {
      const sel = document.getElementById("workspaceSelect");
      sel.innerHTML = "";

      ws.forEach(w => {
        const opt = document.createElement("option");
        opt.value = w.name;
        opt.textContent = w.name;
        sel.appendChild(opt);
      });
    })
    .getWorkspacesPublic(); // wrapper for getWorkspaces_()
}
```

Implement workspace switching:

```
function onWorkspaceChange() {
  const name = document.getElementById("workspaceSelect").value;

  google.script.run
    .withSuccessHandler(function () {
      addSystemMessage("Workspace switched to: " + name);
      document.getElementById("messages").innerHTML = "";
    })
    .setActiveWorkspacePublic(name);
}
```

Add these two public functions in Apps Script:

```
function getWorkspacesPublic() {
  return getWorkspaces_();
}

function setActiveWorkspacePublic(name) {
  return setActiveWorkspace(name);
}
```

🧩 Part 5 — RAG in the New Workspace

Your original `buildRagIndex()` now becomes:

```
function buildRagIndexForWorkspace(name) {
  const ws = getWorkspaces_().find(w => w.name === name);
  if (!ws) throw new Error("Workspace not found: " + name);

  return buildRagIndex_(ws.folderId, ws.sheetId);
}
```

This allows:

```
buildRagIndexForWorkspace("Policies");
buildRagIndexForWorkspace("Onboarding");
buildRagIndexForWorkspace("Project A");
```

🧪 What Users Can Do Now

Try:

✔ “Switch workspace to *Project A*.”

✔ “Where is the onboarding checklist stored?”

✔ “Reset workspace chat.”

✔ “Build RAG index for *Policies*.”

✔ “Now answer using *Policies*.”

You now have a **multi-brain AI system**, each workspace isolated with:

- Its own documents
- Its own embeddings
- Its own chat memory

This is extremely close to a lightweight **Notion AI**, **ChatGPT Files**, or **Mini RAG Assistant**.