Healthy Hints: Fix the Image Upload and OCR Tools

Post-codelab extension lab | Node.js? No - Python ADK, Cloud Storage, and Cloud Vision API

Goal. Replace the codelab's hardcoded local ingredient-label image with a working pipeline that accepts a chosen image, uploads it to Google Cloud Storage, passes its unique `gs://` URI to OCR, and asks the ADK agent to discuss the extracted ingredients. This exercise begins after students finish the Google Healthy Hints codelab.

Source: https://codelabs.developers.google.com/multi-tools-ai-agent-adk (Step 7, SDK Integration). The codelab code is the starting point; the corrected implementation and security guidance below are instructional improvements.

1. Identify the defects in the original codelab

• read_ingredients() opens a fixed author-specific path, so it never processes the image selected by the user.

• upload_image() has no argument, uses an unresolved `<image-file-path>` placeholder, uploads to a fixed object name, and returns no URI.

• The two tools are disconnected: the OCR tool ignores the uploaded Cloud Storage object.

• The original OCR code assumes text exists; indexing `texts[0]` can fail when no text is detected.

2. Correct architecture

```
Student selects an existing local ingredient-label file
       |  (path available to the Python ADK process)
       v
ADK tool: upload_image(image_path) -> gs://bucket/uploads/unique-name.jpg
       |  return URI
       v
ADK tool: read_ingredients(gcs_uri) -> OCR text via Vision API
       |
       v
ADK Gemini agent -> ingredient explanation + uncertainty notes
```

Scope: This version works for files that are already accessible on the machine running the ADK Python process. A browser-based upload requires an additional frontend/backend file-transfer endpoint; simply typing a local path into a remote ADK web chat does not upload the file.

3. Prerequisites and setup

Use the same Google Cloud project and Python virtual environment from the completed codelab. Create or reuse a Cloud Storage bucket that your project can access. Bucket names are globally unique. Do not reuse the example author's bucket name.

```
# In your existing ADK project virtual environment:
pip install google-adk google-cloud-storage google-cloud-vision

gcloud config set project YOUR_PROJECT_ID
gcloud services enable storage.googleapis.com vision.googleapis.com
```

```
# Local development credentials for Google Cloud client libraries:
gcloud auth application-default login
```

The identity running Python must have permission to create objects in the chosen bucket (for example, an appropriately scoped Storage Object Creator role) and to call the Vision API. Google Cloud billing, APIs, organization policy, and quotas must permit the operations. Use a dedicated service account with least privilege when deployed. Avoid downloaded service-account JSON keys.

Set environment variables before running the ADK agent:

```
# macOS/Linux
export HEALTHY_HINTS_BUCKET="YOUR_UNIQUE_BUCKET_NAME"
export HEALTHY_HINTS_UPLOAD_DIR="./images"

# Windows PowerShell
$env:HEALTHY_HINTS_BUCKET="YOUR_UNIQUE_BUCKET_NAME"
$env:HEALTHY_HINTS_UPLOAD_DIR=".\images"
```

Place two distinct ingredient-label images in the configured `images` directory, such as `label_a.jpg` and `label_b.jpg`. These must be real files, not merely browser-visible paths.

4. Replace the original tools with the complete Python code

Put the following code in your existing ADK agent module (commonly `agent.py`), replacing the original `upload_image` and `read_ingredients` functions. The example intentionally restricts local file access to an instructor-selected directory.

```
import os
import uuid
from pathlib import Path

from google.cloud import storage, vision

BUCKET_NAME = os.environ["HEALTHY_HINTS_BUCKET"]
UPLOAD_DIR = Path(os.getenv("HEALTHY_HINTS_UPLOAD_DIR", "./images"))
UPLOAD_DIR = UPLOAD_DIR.resolve()
ALLOWED_SUFFIXES = {".jpg", ".jpeg", ".png", ".webp"}


def upload_image(image_path: str) -> str:
    """Upload a local ingredient image; return its unique gs:// URI.

    image_path must refer to a file within HEALTHY_HINTS_UPLOAD_DIR.
    """
    candidate = Path(image_path)
    if not candidate.is_absolute():
        candidate = UPLOAD_DIR / candidate
    candidate = candidate.resolve()

    if not candidate.is_relative_to(UPLOAD_DIR):
        raise ValueError("Image must be inside the allowed images directory")
    if not candidate.is_file():
        raise FileNotFoundError(f"Image not found: {candidate.name}")
    if candidate.suffix.lower() not in ALLOWED_SUFFIXES:
        raise ValueError("Only JPG, PNG, or WebP images are allowed")
    if candidate.stat().st_size > 8 * 1024 * 1024:
        raise ValueError("Image exceeds the 8 MiB classroom limit")

    content_type = {
        ".jpg": "image/jpeg", ".jpeg": "image/jpeg",
        ".png": "image/png", ".webp": "image/webp",
    }[candidate.suffix.lower()]
    object_name = f"uploads/{uuid.uuid4().hex}{candidate.suffix.lower()}"
    bucket = storage.Client().bucket(BUCKET_NAME)
```

```
    blob = bucket.blob(object_name)
    blob.upload_from_filename(str(candidate), content_type=content_type)
    return f"gs://{BUCKET_NAME}/{object_name}"

def read_ingredients(gcs_uri: str) -> str:
    """Run Google Cloud Vision OCR on a specified GCS image URI."""
    prefix = f"gs://{BUCKET_NAME}/uploads/"
    if not gcs_uri.startswith(prefix):
        raise ValueError("Image URI must be in the approved upload prefix")

    client = vision.ImageAnnotatorClient()
    image = vision.Image(source=vision.ImageSource(image_uri=gcs_uri))
    response = client.text_detection(image=image)

    if response.error.message:
        raise RuntimeError(f"Vision OCR failed: {response.error.message}")
    if not response.text_annotations:
        return "No readable text detected in the image."
    return response.text_annotations[0].description
```

Notes: The 8 MiB check is a classroom application limit, not a statement of the Vision API's universal limits. For production, validate actual file contents, restrict the bucket/prefix and IAM, apply lifecycle deletion policies, and handle uploaded data as potentially sensitive. This example assumes the Vision API can read the referenced Cloud Storage object with the configured identities and permissions.

5. Update the ADK agent definition

Keep the existing ADK project structure and model configuration from the codelab. Replace the old `root_agent` definition with:

```
from google.adk.agents import Agent

root_agent = Agent(
    model="gemini-2.5-flash",
    name="healthy_hints_agent",
    description="Analyze ingredient-label images using GCS and Vision OCR",
    instruction=(
        "You help users understand packaged-food ingredient labels. "
        "When the user supplies a local image filename, first call "
        "upload_image(image_path) and capture its returned gs:// URI. "
        "Then call read_ingredients(gcs_uri) with EXACTLY that URI. "
        "Use the returned OCR text to explain the ingredients, "
        "not a remembered or invented ingredient list. "
        "If OCR is incomplete, say so. Avoid categorical claims that "
        "an ingredient is healthy or unhealthy without context. "
        "Do not provide personalized medical advice."
    ),
    tools=[upload_image, read_ingredients],
)
```

The agent uses two registered Python tools. Tool parameters and return values now allow the model to pass the result of the upload to the OCR call. ADK is used here for agent/tool orchestration; Cloud Vision performs OCR and Gemini interprets the text.

6. Run and test the modified agent

From the parent folder of your ADK agent package, start the agent using the same command used in the codelab, such as `adk web` or `adk run your_agent_folder`. Verify that the selected agent is the modified `healthy_hints_agent`.

Example prompt for the local-path version:

```
Analyze the ingredients in label_a.jpg. Upload that file first,
then use OCR on the returned GCS URI. Tell me what text was
recognized and explain the ingredients with appropriate caveats.
```

Repeat with `label_b.jpg`. Inspect the ADK tool trace to verify two distinct URIs and two OCR outputs. The correct sequence is `upload_image` then `read_ingredients`, using the exact returned URI.

7. Optional: deterministic pipeline test before using the agent

To isolate cloud API problems from agent tool-selection problems, run the functions directly in Python. Import from the module containing the functions, adjusting the module path for your project:

```
from your_agent_folder.agent import upload_image, read_ingredients

uri_a = upload_image("label_a.jpg")
uri_b = upload_image("label_b.jpg")
print(uri_a)
print(uri_b)
assert uri_a != uri_b
print("A OCR:\n", read_ingredients(uri_a))
print("B OCR:\n", read_ingredients(uri_b))
```

If this succeeds but the ADK chat fails, inspect the tool call names, arguments, and tool responses. If it fails here, investigate bucket permissions, local paths, API enablement, and the Vision response.

8. Troubleshooting

Symptom	Likely cause / next check
File not found	The path is not inside the configured local image directory, or the file is absent.
403 permission denied	Check ADC/service account, bucket IAM, Vision API permissions, and project selection.
OCR returns no text	Use a clearer image; confirm label text is visible and readable.
Agent reads old image	Verify the OCR tool takes gcs_uri and does not open a fixed local path.
Both uploads share one object	Confirm the object name uses a unique UUID.
Browser upload not working	This lab accepts server-accessible files; implement an upload endpoint for browser bytes.

9. Student deliverables and evaluation

• Screenshot of ADK agent tool trace showing upload -> OCR -> Gemini response for label_a.jpg.

• Equivalent trace for label_b.jpg, with a different GCS URI and different recognized text.

• Source code for corrected tools and agent definition (no credentials or private keys).

• Short explanation: why the original hardcoded file path made the codelab misleading, and how the URI now links the two tools.

• One paragraph explaining what the lab does not yet support: direct browser file uploads, production-grade image validation, and medically personalized nutrition guidance.

10. Extension challenge: actual browser upload

A real web application should accept multipart image uploads in an authenticated backend endpoint, validate file type and size, upload the received bytes to Cloud Storage, and pass the resulting URI to the ADK workflow. Do not accept arbitrary server filesystem paths from public users. This is an optional next exercise, not implemented by the code above.

References: Google ADK codelab: https://codelabs.developers.google.com/multi-tools-ai-agent-adk ; Google Cloud Storage Python client: https://cloud.google.com/python/docs/reference/storage/latest ; Cloud Vision text detection: https://cloud.google.com/vision/docs/ocr ; Google ADK documentation: https://google.github.io/adk-docs/