Lab: Integrating Multiple AI Models into an Android Application with a Supporting Google Cloud Run Backend Service

Using REST APIs and Google Cloud Run to Access Base and Fine-Tuned Gemini Models

Scope and warning: The code is for guidance and has not been tested end-to-end. Students must verify model availability, endpoint IDs, authorization, billing, and security settings.

1. How this fits Project 2

Component	Project requirement	Possible access
Model A	Student-trained or fine-tuned vision model	On-device inference or separately hosted model API
Model B	Fine-tuned multimodal Gemini	Student-built Cloud Run backend to tuned Google Cloud endpoint
Model C	Additional pretrained vision/foundation model; may be base Gemini	On-device, third-party API, Firebase AI Logic, or Cloud Run

Each model needs a distinct role in the actual mobile system. Base Gemini used only for evaluation does not automatically qualify as Model C.

2. Options for accessing models from Android

Option	Model access method	Use / notes
OPTION 1	On-device model inference (e.g., LiteRT)	For compatible trained or pretrained vision models.
OPTION 2	Direct REST call to an externally hosted model	For non-Gemini models hosted elsewhere, investigate provider-specific APIs, authentication, input/output formats, and costs.
OPTION 3.1	Base Gemini via Firebase AI Logic	Direct Android SDK access to base Gemini; does not support fine-tuned Gemini.
OPTION 3.2	Base **and** fine-tuned Gemini via one Google Cloud Run backend	Build one Node.js + Express service with `/api/base-gemini` and `/api/fine-tuned-gemini`. Both return JSON to Android.

For this lab: Option 3.2 is the unified backend approach for **both** Gemini models. The Android application sends image + text to the same Cloud Run service, selecting the base or tuned endpoint. Option 3.1 is an alternative for base Gemini only; students who use 3.2 for both do not need Firebase AI Logic.

Documentation links for each option

Option 1 - LiteRT for Android

Option 2 - Android networking (OkHttp)

Option 3.1 - Firebase AI Logic Android getting started

Option 3.2 - Cloud Run Node.js deployment

Option 3.2 - Gemini Developer API keys (base model)

Option 3.2 - Vertex AI Gemini API (fine-tuned model)

Before Section 3: Start with the Express tutorial

How to Create a Gemini API based Backend

The linked illustrated CS351 tutorial is the starting point for the Cloud Run backend.
It demonstrates WebStorm, Express Generator, Google AI Studio API keys, Secret Manager,
and deployment to Google Cloud Run. Its Gemini example uses text-only input.

Next, follow Section 3 of this CS663 guide to extend that project for Project 2:

- Accept a camera image plus a text prompt in a JSON POST request.
- Implement /api/base-gemini and /api/fine-tuned-gemini.
- Use the AI Studio API key for base Gemini and Cloud Run ADC/IAM for tuned Gemini.
- Return a JSON object containing the generated text to the Android application.

IMPORTANT: The code in Sections 3.1 onward is guidance and has not been tested.
If your models accept different inputs or return different outputs, adapt the code.

Express Generator file mapping (use this if following the linked tutorial):

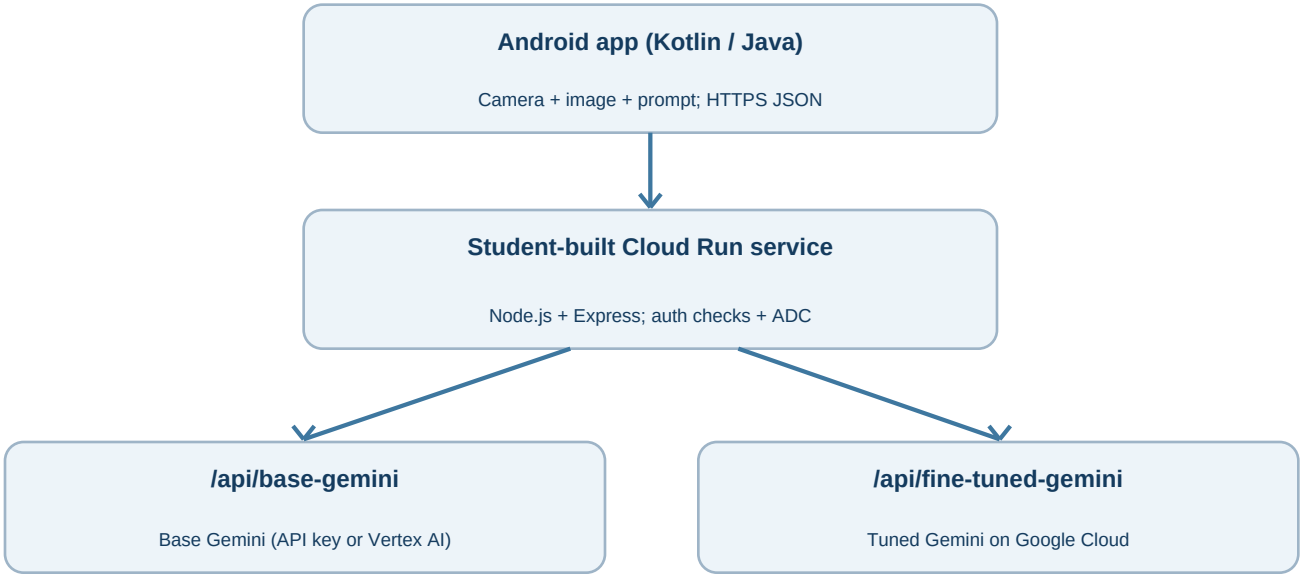
app.js	Configure middleware and mount the API router.
routes/index.js	Or routes/gemini.js: implement both POST endpoints.
bin/www	Starts the Express server; keep its PORT handling.
package.json	Dependencies and the generator startup script.
.env	Local-only secret configuration; never commit.

The standalone index.js sample in Section 3 uses a different project structure.
It is NOT a drop-in replacement for routes/index.js. Transfer the handler logic
into the Express Generator routes, and mount those routes from app.js. Keep
CommonJS or deliberately convert the whole project to ES modules; do not mix styles.

Suggested sequence: finish CS351 tutorial -> adapt Section 3 code -> deploy ->
connect the Android Kotlin/Java client in Section 4 -> test both models.

3. Architecture & Code for Android app + Secure Nodejs+Express backend

Assumption: Both model calls accept a multimodal input consisting of **one image + one text prompt** and return **generated text** inside a JSON response. For other input/output variations (multiple images, video, structured output, bounding boxes, audio, etc.), students must adapt the code and response handling. **All code is illustrative and not tested.**



The Android application sends JSON with `prompt`, `imageBase64`, and `mimeType` (for example, image/jpeg). Both endpoints return JSON of the form `{"text":"..."}`. Models A and C can follow separate on-device or external paths.

Getting Started: Node.js + Express + Gemini on Google Cloud Run

How to Create a Gemini API based Backend

This illustrated tutorial walks through creating a Node.js Express application, obtaining a Gemini API key through Google AI Studio, and deploying the application to Google Cloud Run.

IMPORTANT: The tutorial uses text-only Gemini requests. For CS663 Project 2, you must extend the implementation to handle multimodal image + text requests and access your fine-tuned Gemini model using appropriate Google Cloud authentication (Cloud Run service account / ADC and IAM).

Continue with Section 3.1 on the next page. The code examples are guidance, not tested.

3.1 Choose authentication for each Gemini call

Base Gemini: This example uses a Gemini Developer API key created in Google AI Studio, stored **only in the Cloud Run environment**. **Fine-tuned Gemini:** Uses the Google Cloud Vertex AI / Agent Platform endpoint and the Cloud Run service account through Application Default Credentials (ADC). A Google AI Studio key does **not** authenticate a private tuned Vertex AI endpoint. Alternatively, base Gemini can also use Vertex AI and ADC, eliminating the AI Studio key.

Create/manage a Gemini Developer API key in Google AI Studio

Cloud Run service identity and ADC

3.2 Create the Node.js project

```
mkdir cs663-gemini-service
cd cs663-gemini-service
npm init -y
npm install express @google/genai
npm pkg set type=module
npm pkg set scripts.start="node index.js"
```

Files: `package.json`, `index.js`, `.gitignore`. Never commit API keys or service account JSON files.

3.3 Complete index.js (Node.js + Express)

```
import express from "express";
import { GoogleGenAI } from "@google/genai";

const app = express();
app.use(express.json({ limit: "12mb" }));

const project = process.env.GOOGLE_CLOUD_PROJECT;
const location = process.env.VERTEX_LOCATION || "us-central1";
const baseModel = process.env.BASE_MODEL || "gemini-2.5-flash";
const tunedEndpoint = process.env.TUNED_ENDPOINT;
const apiKey = process.env.GEMINI_API_KEY;

// Developer API: base Gemini, with server-side API key.
const baseAI = apiKey ? new GoogleGenAI({ apiKey }) : null;
// Vertex AI: uses ADC from Cloud Run service account.
const vertexAI = project ? new GoogleGenAI({
  vertexai: true, project, location
}) : null;

function readInput(req) {
  const { prompt, imageBase64, mimeType = "image/jpeg" } = req.body || {};
  if (typeof prompt !== "string" || !prompt.trim())
    throw new Error("prompt must be a nonempty string");
  if (typeof imageBase64 !== "string" || !imageBase64.trim())
    throw new Error("imageBase64 is required");
  if (!["image/jpeg", "image/png", "image/webp"].includes(mimeType))
    throw new Error("unsupported mimeType");
  return { prompt, imageBase64, mimeType };
}

function partsFor({ prompt, imageBase64, mimeType }) {
  return [{ text: prompt }, {
    inlineData: { mimeType, data: imageBase64 }
  }];
}

async function runModel(ai, model, input) {
  const result = await ai.models.generateContent({
    model,
    contents: [{ role: "user", parts: partsFor(input) }]
  });
  return { text: result.text || "" };
}

// IMPORTANT: Add caller authentication, authorization,
// rate limits and abuse prevention before exposing publicly.
app.post("/api/base-gemini", async (req, res) => {
  try {
    if (!baseAI) return res.status(503).json({ error: "Base API not set" });
    const output = await runModel(baseAI, baseModel, readInput(req));
    res.json(output);
  } catch (err) {
    console.error("base error", err);
    res.status(400).json({ error: "Base request failed" });
  }
});

app.post("/api/fine-tuned-gemini", async (req, res) => {
  try {
    if (!vertexAI || !tunedEndpoint)
      return res.status(503).json({ error: "Tuned model not set" });
    const output = await runModel(vertexAI, tunedEndpoint, readInput(req));
    res.json(output);
  } catch (err) {
    console.error("tuned error", err);
    res.status(400).json({ error: "Tuned request failed" });
  }
});

app.get("/health", (_req, res) => res.json({ status: "ok" }));
const port = Number(process.env.PORT || 8080);
app.listen(port, "0.0.0.0", () => console.log(`Listening on ${port}`));
```

Tuned endpoint: Set `TUNED_ENDPOINT` to the actual resource identifier returned by the fine-tuning/deployment workflow (often a full `projects/.../locations/.../endpoints/...` path). Confirm that the SDK accepts that identifier for the selected tuned model. Endpoint locations and model availability vary.

3.4 package.json and .gitignore

```
// package.json (example; npm install writes actual versions)
{
  "name": "cs663-gemini-service",
  "version": "1.0.0",
  "type": "module",
  "scripts": { "start": "node index.js" },
  "dependencies": {
    "@google/genai": "^1.0.0",
    "express": "^4.21.0"
  }
}

// .gitignore (separate file)
node_modules/
.env
*.json.key
```

The version strings above are illustrative; use the versions actually installed and test compatibility. Keep `package-lock.json` in source control for reproducible builds.

3.5 Configure Google Cloud and deploy

In Google Cloud Console, select the student group's project and ensure billing/educational credits are correctly configured. Enable Cloud Run, Cloud Build, Artifact Registry, and Vertex AI APIs. Create a dedicated runtime service account; grant the minimum required Vertex AI permissions (often `roles/aiplatform.user`), and ensure access to the tuned endpoint. Do not download a service-account key.

```
gcloud config set project PROJECT_ID

gcloud services enable run.googleapis.com \
  cloudbuild.googleapis.com artifactregistry.googleapis.com \
  aiplatform.googleapis.com

gcloud iam service-accounts create cs663-gemini-runtime

gcloud projects add-iam-policy-binding PROJECT_ID \
  --member="serviceAccount:cs663-gemini-runtime@PROJECT_ID.iam.gserviceaccount.com" \
  --role="roles/aiplatform.user"
```

Store the Google AI Studio key in Secret Manager rather than source code or plaintext configuration. Give the Cloud Run runtime identity access to that secret. The following deployment is a **template**; configure the secret and tuned endpoint first, and use the current supported Node.js runtime/build settings.

```
gcloud run deploy cs663-gemini-service \
  --source . --region us-central1 \
  --service-account cs663-gemini-runtime@PROJECT_ID.iam.gserviceaccount.com \
  --set-env-vars GOOGLE_CLOUD_PROJECT=PROJECT_ID,VERTEX_LOCATION=us-central1,BASE_MODEL=gemini-2.
5-flash,TUNED_ENDPOINT=ENDPOINT_RESOURCE \
  --set-secrets GEMINI_API_KEY=gemini-api-key:latest
```

Security: The sample endpoints above do not verify Android users. Do not deploy them as an unrestricted public paid-model proxy. A student group must choose an access-control method: for example Firebase Authentication ID tokens validated by Express, plus Firebase App Check verification where appropriate; or a private Cloud Run service with an authorized calling mechanism. Do not assume that making Cloud Run public is secure simply because its upstream Gemini credentials are private.

Cloud Run deploy from source

Secret Manager environment secrets on Cloud Run

Firebase Admin: verify ID tokens (Node.js)

3.6 Testing Your Backend from a Browser

Before developing the Android application, test both deployed Gemini endpoints using a simple browser page served **by the same Node.js + Express application**. This page is only for testing; no separate web service or frontend deployment is needed.

Assumption: Both endpoints accept one image (Base64) and one text prompt and return JSON containing a `text` field. Adapt the examples for different model inputs or outputs. **Code is illustrative guidance and has not been tested end-to-end.**

3.6.1 Files to add to the existing Express application

```
cs663-gemini-service/
  index.js                  # existing standalone Express server
  public/
    test.html               # browser test page
    test.js                 # image encoding and fetch() calls
  package.json
```

For an Express Generator project, use the existing `app.js` instead of standalone `index.js`. Keep the existing Gemini POST handlers unchanged.

3.6.2 Serve the test page from the same Express application

```
// Standalone index.js: add before app.listen(...)
app.use(express.static("public"));
app.get("/test", (_req, res) => {
  res.sendFile("test.html", { root: "public" });
});

// Express Generator: add to app.js after creating app:
// app.use(express.static(path.join(__dirname, "public")));
// app.get("/test", (_req, res) =>
//   res.sendFile(path.join(__dirname, "public", "test.html")));
```

The test page will be available at `https://YOUR-SERVICE.run.app/test`. It calls the existing `POST /api/gemini` and `POST /api/fine-tuned-gemini` endpoints on the same origin.

3.6.3 Create public/test.html

```
<!doctype html>
<html lang="en">
<head>
  <meta charset="utf-8">
  <meta name="viewport" content="width=device-width,initial-scale=1">
  <title>CS663 Gemini Backend Test</title>
  <style>
    body { font: 16px Arial,sans-serif; max-width: 850px;
           margin: 2rem auto; padding: 0 1rem; }
    label { display:block; margin-top: 1rem; font-weight:bold; }
    textarea { width:100%; min-height:80px; }
    button { margin: 1rem 0.5rem 1rem 0; padding:0.7rem; }
    pre { background:#f0f4f8; padding:1rem; white-space:pre-wrap; }
  </style>
</head>
<body>
  <h1>Test Base and Fine-Tuned Gemini</h1>
  <label for="image">Image (JPEG or PNG)</label>
  <input id="image" type="file" accept="image/jpeg,image/png">
  <label for="prompt">Text prompt</label>
```

```
  <textarea id="prompt">Describe this image.</textarea>
  <p><button id="baseBtn">Test Base Gemini</button>
     <button id="tunedBtn">Test Fine-Tuned Gemini</button>
     <button id="bothBtn">Test Both</button></p>
  <h2>Base Gemini JSON response</h2><pre id="base">Not tested</pre>
  <h2>Fine-Tuned Gemini JSON response</h2><pre id="tuned">Not tested</pre>
  <script src="/test.js"></script>
</body>
</html>
```

3.6.4 Create public/test.js - image + text JSON and fetch()

```
const el = (id) => document.getElementById(id);

async function getPayload() {
  const file = el("image").files[0];
  const prompt = el("prompt").value.trim();
  if (!file) throw new Error("Choose an image first.");
  if (!prompt) throw new Error("Enter a text prompt.");
  if (!["image/jpeg", "image/png"].includes(file.type))
    throw new Error("Use a JPEG or PNG image.");
  const dataUrl = await new Promise((resolve, reject) => {
    const reader = new FileReader();
    reader.onload = () => resolve(reader.result);
    reader.onerror = () => reject(new Error("Image read failed"));
    reader.readAsDataURL(file);
  });
  return {
    prompt,
    mimeType: file.type,
    imageBase64: dataUrl.split(",")[1]
  };
}

async function invoke(path, outputId, payload) {
  el(outputId).textContent = "Processing...";
  try {
    const response = await fetch(path, {
      method: "POST",
      headers: { "Content-Type": "application/json" },
      body: JSON.stringify(payload)
    });
    const result = await response.json();
    el(outputId).textContent = JSON.stringify(result, null, 2);
    if (!response.ok) console.error(path, response.status, result);
  } catch (error) {
    el(outputId).textContent = "Request failed: " + error.message;
  }
}
```

3.6.5 Wire the three buttons

```
el("baseBtn").onclick = async () => {
  try { await invoke("/api/gemini", "base", await getPayload()); }
  catch (e) { el("base").textContent = e.message; }
};
el("tunedBtn").onclick = async () => {
  try { await invoke("/api/fine-tuned-gemini", "tuned",
                     await getPayload()); }
  catch (e) { el("tuned").textContent = e.message; }
};
el("bothBtn").onclick = async () => {
  try {
    const payload = await getPayload();
```

```
    await Promise.all([
      invoke("/api/gemini", "base", payload),
      invoke("/api/fine-tuned-gemini", "tuned", payload)
    ]);
  } catch (e) { alert(e.message); }
};
```

3.6.6 Deploy and demonstrate the test page

Deploy the updated Express service to Cloud Run using the deployment instructions already given in this lab. Open `https://YOUR-SERVICE.run.app/test`, select an image, enter a prompt, and test each endpoint. Show that both return JSON containing the generated `text` response. Try the same image and prompt with both models to compare outputs.

Security: The browser page must not contain Gemini API keys or service-account credentials. Protect the test page and API endpoints with suitable caller authentication and authorization; do not leave a public, unrestricted billable model proxy. If authentication is enabled, configure the browser testing flow accordingly. This example does not implement browser sign-in.

What to demonstrate: the deployed `/test` URL, image selection, prompt entry, a successful response from each model, and graceful handling of an error. The same two POST routes will later be called from Android.

Cloud Run authentication overview

3.7 Example HTTP request and JSON response

```
POST https://YOUR-CLOUD-RUN-URL/api/base-gemini
Content-Type: application/json
Authorization: Bearer <app-user-token-if-configured>

{
  "prompt": "Describe the important objects in this image.",
  "mimeType": "image/jpeg",
  "imageBase64": "<BASE64_IMAGE_BYTES>"
}

// Both endpoints return a JSON object like:
{ "text": "The image shows a person near a bicycle." }
```

4. Android Kotlin and Java: send image + text; receive JSON text

The Android application captures or selects an image, converts the image bytes to Base64, POSTs the same JSON payload to either backend endpoint, and parses the returned `text` field. The code is illustrative, **not tested**, and must run off the Android main thread. Replace the placeholder URL and add the application's authentication token according to the security approach chosen in Section 3.

4.1 Android dependencies and permissions

```
// app/build.gradle.kts (Kotlin DSL)
dependencies {
    implementation("com.squareup.okhttp3:okhttp:4.12.0")
}

// AndroidManifest.xml (inside <manifest>)
<uses-permission android:name="android.permission.INTERNET" />
```

OkHttp performs HTTPS requests; `org.json.JSONObject` is included in Android. Android `Base64` encodes image bytes. For camera capture, use CameraX or the Android photo picker; pass the resulting JPEG/PNG bytes to the functions below.

4.2 Kotlin example (imports, request, JSON parsing)

```
import android.util.Base64
import okhttp3.MediaType.Companion.toMediaType
import okhttp3.OkHttpClient
import okhttp3.Request
import okhttp3.RequestBody.Companion.toRequestBody
import org.json.JSONObject

private val client = OkHttpClient()
private const val BASE_URL = "https://YOUR-SERVICE.run.app"

fun callGemini(
    imageBytes: ByteArray,
    prompt: String,
    tuned: Boolean,
    userToken: String? = null
): String {
    val path = if (tuned) "/api/fine-tuned-gemini" else "/api/base-gemini"
    val payload = JSONObject()
        .put("prompt", prompt)
        .put("mimeType", "image/jpeg")
        .put("imageBase64", Base64.encodeToString(imageBytes, Base64.NO_WRAP))
    val body = payload.toString().toRequestBody("application/json".toMediaType())
    val builder = Request.Builder().url(BASE_URL + path).post(body)
    if (userToken != null) builder.header("Authorization", "Bearer $userToken")
    client.newCall(builder.build()).execute().use { response ->
        val json = JSONObject(response.body?.string() ?: "{}")
        if (!response.isSuccessful) error(json.optString("error", "HTTP error"))
        return json.getString("text")
    }
}

// Call from Dispatchers.IO or another background executor, not the UI thread.
// Example: val result = withContext(Dispatchers.IO) {
//     callGemini(jpegBytes, "Describe the scene", tuned = true, userToken)
// }
```

4.3 Java example (imports, request, JSON parsing)

```
import android.util.Base64;
import org.json.JSONObject;
import okhttp3.MediaType;
import okhttp3.OkHttpClient;
import okhttp3.Request;
import okhttp3.RequestBody;
import okhttp3.Response;
import java.io.IOException;

public class GeminiGatewayClient {
  private static final String BASE_URL = "https://YOUR-SERVICE.run.app";
  private final OkHttpClient client = new OkHttpClient();

  // Run on a background thread, not the Android main thread.
  public String callGemini(byte[] imageBytes, String prompt,
                           boolean tuned, String userToken) throws Exception {
    String path = tuned ? "/api/fine-tuned-gemini" : "/api/base-gemini";
    JSONObject payload = new JSONObject();
    payload.put("prompt", prompt);
    payload.put("mimeType", "image/jpeg");
    payload.put("imageBase64", Base64.encodeToString(imageBytes, Base64.NO_WRAP));
    RequestBody body = RequestBody.create(
        payload.toString(), MediaType.parse("application/json"));
    Request.Builder builder = new Request.Builder()
        .url(BASE_URL + path).post(body);
    if (userToken != null) builder.header("Authorization", "Bearer " + userToken);
    try (Response response = client.newCall(builder.build()).execute()) {
      String raw = response.body() == null ? "{}" : response.body().string();
      JSONObject json = new JSONObject(raw);
      if (!response.isSuccessful())
        throw new IOException(json.optString("error", "HTTP error"));
      return json.getString("text");
    }
  }
}
```

Implementation notes: The examples assume JPEG bytes and a text response. Resize/compress large camera images to stay within payload limits, do not log images or sensitive prompts, handle timeouts and retries, and display meaningful failures. If Firebase Authentication is used, retrieve a current Firebase ID token in Android and validate it in the backend; merely attaching an arbitrary token string is not authentication.

5. Google Cloud authentication and security

There are **two separate trust boundaries**: (1) Android user to Cloud Run service; (2) Cloud Run service to Gemini. For (1), authenticate/authorize Android callers using a supported user identity flow and protect the service from abuse. For (2), Cloud Run uses its assigned service account and ADC to access the tuned model. The base model example instead uses a Gemini Developer API key stored as a Cloud Run secret. Firebase AI Logic (Option 3.1) provides Android integration but does not support the private tuned Gemini endpoint.

Students should not put Gemini API keys, service-account JSON files, or Google Cloud service-account tokens into the Android APK. Restrict IAM, set budgets/quotas, monitor usage, and test failure cases. Cloud Run ADC is automatic only after the runtime identity and its permissions are configured correctly.

Vertex AI authentication

Application Default Credentials

Firebase App Check for custom backends

6. Example integrated Project 2 architecture

A possible application flow is: camera image -> Model A detects objects -> Model C (potentially base Gemini) provides complementary scene understanding -> fine-tuned Gemini Model B synthesizes specialized guidance -> Android displays text, audio, or visual output. Students may instead use parallel, conditional, or iterative calls when justified. The diagram in Section 3 illustrates only the Gemini service access path, not the entire three-model pipeline.

Students must document model responsibilities, location of inference, data passed between models, failure handling, timing, and base-versus-tuned evaluation on previously unseen examples.

7. Useful official documentation

Gemini API REST reference

Google Gen AI SDK for JavaScript

Gemini Developer API keys

Google Cloud Vertex AI Gemini quickstart

Vertex AI REST generateContent

Cloud Run Node.js quickstart

Cloud Run service identity

Firebase AI Logic

Firebase Authentication ID token verification

Android CameraX

OkHttp

Final caution: API names, model IDs, supported tuning endpoints, and JavaScript SDK behavior can change. Before distributing this guide as a required lab, test one base request and one tuned request in a group-owned Google Cloud project, then test the Android clients against the deployed service.

8. Deliverables

See Canvas for the due date and time. Submit the URL of your deployed Google Cloud Run service for Item 1 and the URL of a YouTube demonstration video showing both Item 1 and Item 2 working.

1. Deploy a Google Cloud Run backend service

Create and deploy a Node.js + Express backend on Google Cloud Run with these two endpoints:

- POST /api/gemini - invokes a base Gemini model.
- POST /api/fine-tuned-gemini - invokes your fine-tuned Gemini model from a previous exercise/lab.

Endpoint naming: If the earlier sample uses /api/base-gemini, rename that route to /api/gemini for this deliverable.

Each endpoint must accept a multimodal input consisting of an image and text, invoke its corresponding model, and return the generated text in a JSON response. Demonstrate that both endpoints work using a web browser (for example, a browser-based test page that submits the requests and displays the returned results).

2. Create or modify an Android application

Create or modify an Android application that allows the user to take a picture using the device camera and enter a text prompt in a text box.

The application must send both the captured image and the entered text to both backend endpoints, /api/gemini and /api/fine-tuned-gemini; receive and parse the JSON responses; and display the two generated text results to the user with clear, attractive formatting so the outputs can be compared.

YouTube demonstration: Show the deployed backend working from a web browser, then demonstrate the Android application capturing a new camera image, entering a text prompt, invoking both endpoints, and displaying both results.