ChatGPT

Setting Up an AI-Powered Trip Planner Backend (Gemini 2.5 Pro, Vertex AI & Firebase)

Building a **personalized trip planner** backend involves configuring Google's Vertex AI (with the Gemini 2.5 Pro model), integrating Firebase services, and structuring a FastAPI application. We will use **Astral's `uv` package manager** for dependency management and environment isolation. Below are the steps and best practices to set up the project, including code examples and explanations for each component.

1. Dependencies and Environment Setup

Using `uv` for package management: Begin by installing `uv`, a fast Python package and project manager designed as a drop-in replacement for pip [1]. You can install it via pip or its install script. For example:

```
# Install uv (if not already installed)
pip install uv

# (Optional) Create a virtual environment quickly using uv (faster than venv)
uv venv .venv
```

Activate the virtual environment (if created) and use `uv` to install and pin required packages. The key dependencies for this project include:

- **google-generativeai** – Google's Generative AI SDK (for Gemini model access)
- **google-cloud-aiplatform** – Vertex AI client library (for Vertex AI interactions)
- **vertexai** – (If separate) Vertex AI Python SDK for generative AI
- **google-adk** – Google's Agent Development Kit (for building multi-agent pipelines with Gemini)
- **firebase-admin** – Firebase Admin SDK (for Firebase Auth/Firestore integration)
- **python-dotenv** – For loading configuration from a `.env` file

Using `uv`, add these to your project and lock the versions:

```
# Add dependencies (this will install the packages and allow version resolution)
uv pip install google-generativeai google-cloud-aiplatform vertexai firebase-
admin google-adk python-dotenv

# (Optional) Generate a locked requirements file for reproducibility
uv pip freeze > requirements.txt
```

Alternatively, create a `requirements.in` listing the above packages (with version ranges if desired), then run `uv pip compile -o requirements.txt requirements.in` to produce a locked `requirements.txt` [2]. This ensures deterministic builds (the lockfile should be committed to version control). The `uv` tool will handle dependency resolution and installation in an isolated environment.

Project initialization: Use a structured layout (detailed below) and ensure `uv` is managing the environment. In summary, after installing `uv`, your environment setup is as follows:

- Install all required libraries in an isolated env using `uv`.
- Keep a `requirements.txt` (or `uv.lock` if using uv's native format) in the repo for dependency locking.
- Use `python-dotenv` to manage environment-specific settings without hardcoding secrets in code.

2. Authentication and Configuration

Google Cloud Service Account: Set up a Google Cloud service account with access to Vertex AI. In the Google Cloud Console, create a service account and grant it the **"Vertex AI User"** role [3] (and Firestore/Storage roles if needed for Firebase). Then **create a JSON key** for this service account [4]. **Store this key file securely** – do not commit it to the repo. For local development, place the JSON in a safe path and reference it via an environment variable:

- `GOOGLE_APPLICATION_CREDENTIALS="/path/to/service-account-key.json"` – this tells Google SDKs (like Vertex AI and Firebase Admin) to use this key for authentication [5].

Vertex AI configuration: Enable the Vertex AI API in your GCP project. In the `.env` file, set environment variables for Vertex AI usage:

```
GOOGLE_CLOUD_PROJECT="your-gcp-project-id"
GOOGLE_CLOUD_LOCATION="your-vertex-region"     # e.g., us-central1
GOOGLE_GENAI_USE_VERTEXAI=true                # use Vertex AI endpoints for
generative AI [6]
```

The `GOOGLE_CLOUD_PROJECT` and `GOOGLE_CLOUD_LOCATION` variables ensure the Vertex AI SDK knows which project and region to use [6]. `GOOGLE_GENAI_USE_VERTEXAI=true` tells the Generative AI SDK/ADK to route requests to Vertex AI instead of the public API [7] [8]. With these set (and no `GOOGLE_API_KEY` set, since we'll use the service account), the Google ADK and generative AI libraries will authenticate using Application Default Credentials (your service account) to call the Gemini model on Vertex AI.

Firebase API and credentials: If your trip planner uses Firebase (e.g., for user authentication or storing itineraries in Firestore), set up Firebase in the same GCP project (for simplicity). Obtain your Firebase config values and service account:

- For **Firebase Admin SDK**, you can either reuse the Google Cloud service account (if it has Firebase permissions) or generate a separate Firebase Admin SDK service account key from the Firebase console. Save this JSON key similar to above.
- For **Firebase client API** (if needed on the backend, e.g., to verify ID tokens or call Firebase REST APIs), you'll have an API key, Auth domain, etc. These are usually used on the client side, but you may store the Firebase API key in the backend `.env` if needed (though typically the API key is not secret on its own).

In your `.env` (and `.env.example`), include placeholders for Firebase config:

```
FIREBASE_PROJECT_ID="your-firebase-project-id"
FIREBASE_CLIENT_EMAIL="firebase-service-
account@yourproject.iam.gserviceaccount.com"
FIREBASE_PRIVATE_KEY="-----BEGIN PRIVATE KEY-----\nMIIEvg...YOUR KEY...\n-----
END PRIVATE KEY-----\n"
FIREBASE_API_KEY="your-firebase-web-api-key"  # if needed
```

Alternatively, store the path to the Firebase credentials JSON (similar to the Google Cloud key):

```
FIREBASE_CREDENTIALS="/path/to/firebase-adminsdk.json"
```

Then in code, you can initialize Firebase Admin by loading that file. For example:

```python
import firebase_admin
from firebase_admin import credentials

cred = credentials.Certificate(os.environ["FIREBASE_CREDENTIALS"])
firebase_admin.initialize_app(cred)
```

This will authenticate your server to Firebase services. All sensitive settings (service account paths, API keys, project IDs, etc.) are now sourced from environment variables via `python-dotenv`. **Create a** `.env.example` with the keys and variables (no real secrets, just placeholders) so others know what to set. The real `.env` (with actual secrets) should be kept locally and **added to** `.gitignore` so it's never committed (more on security below).

3. Project Structure and Architecture

Organize the FastAPI project into logical modules for clarity and maintainability. A recommended structure (matching the prompt) is:

```
ai_trip_planner_backend/
├── main.py              # FastAPI app initialization, include routers, startup
config
├── trip_planner/        # Business logic for trip planning
│   ├── routes.py        # API routes (FastAPI APIRouter for trip planning
endpoints)
│   ├── services.py      # Core logic to interact with AI and maps (itinerary
generation)
│   ├── schemas.py       # Pydantic models for request/response schemas
│   └── __init__.py
├── ai_services/         # Integration with AI (Gemini model and ADK agents)
│   ├── gemini_agents.py # ADK agent definitions, model interactions, pipelines
│   ├── __init__.py
│   └── ... (e.g., prompts or helper modules)
├── maps_services/       # Integration with Google Maps or geolocation services
│   ├── maps_client.py   # Functions or classes using Google Maps API (places,
routes)
│   └── __init__.py
├── auth/                # Authentication and user management
│   ├── firebase_auth.py # Utility to verify Firebase JWT, manage users
│   └── __init__.py
├── core/                # Core utilities and configuration
│   ├── config.py        # Reads .env, sets up Config object (project ID, API
keys, etc.)
│   ├── logging.py       # Logging configuration (format, level, handlers)
│   ├── errors.py        # Custom exceptions or error handlers
│   └── __init__.py
├── scripts/             # Example scripts for running agents or setup tasks
│   ├── run_agent_demo.py # Script to demonstrate Gemini agent pipeline usage
locally
│   └── ...
├── tests/               # Test suite
│   ├── test_trip_planner.py   # Tests for trip planning service logic
│   ├── test_api.py           # Tests for FastAPI endpoints (using TestClient)
│   └── __init__.py
├── .env.example         # Example environment configuration
├── requirements.txt     # Locked dependencies (generated by uv)
└── README.md            # Documentation of setup, usage, etc.
```

FastAPI App (main.py): In `main.py`, create the FastAPI instance and include the routers. Also configure any **startup events** for initializing connections (e.g., call `firebase_admin.initialize_app` here if not already called, or load the Vertex AI project config). For example:

```
# main.py
import os
```

```python
from fastapi import FastAPI
from trip_planner import routes as trip_routes
from core import config, logging_config

# Load env variables
from dotenv import load_dotenv
load_dotenv()  # loads variables from .env

# Initialize Firebase Admin (if needed)
if "FIREBASE_CREDENTIALS" in os.environ:
    cred = credentials.Certificate(os.environ["FIREBASE_CREDENTIALS"])
    firebase_admin.initialize_app(cred)

# (Optional) Initialize Vertex AI settings for google.cloud.aiplatform if using
directly
if "VERTEX_PROJECT_ID" in os.environ:
    from google.cloud import aiplatform
    aiplatform.init(project=os.environ["VERTEX_PROJECT_ID"],
location=os.environ["VERTEX_REGION"])

app = FastAPI(title="AI Trip Planner")

# Include API routes
app.include_router(trip_routes.router, prefix="/api/trip", tags=["Trip
Planning"])

# Configure logging
logging_config.setup_logging()
```

(The above is illustrative – the actual structure might differ, e.g. using a factory function to create the app.)

Routes and schemas: In `trip_planner/routes.py`, define FastAPI endpoints for trip planning, using Pydantic models from `schemas.py` for request and response. For instance, a `/plan` POST endpoint might accept a JSON with destination, dates, preferences, etc., and return a structured itinerary. Example snippet:

```python
# trip_planner/schemas.py
from pydantic import BaseModel
from typing import List

class TripRequest(BaseModel):
    destination: str
    start_date: str
    end_date: str
    interests: List[str] = []  # e.g. ["food", "museums"]
```

```
class DayPlan(BaseModel):
    day: int
    activities: List[str]

class TripPlan(BaseModel):
    destination: str
    days: List[DayPlan]
```

```
# trip_planner/routes.py
from fastapi import APIRouter, Depends, HTTPException
from trip_planner.schemas import TripRequest, TripPlan
from trip_planner import services

router = APIRouter()

@router.post("/plan", response_model=TripPlan)
async def plan_trip(req: TripRequest,
user=Depends(firebase_auth.get_current_user)):
    # ^ Example: secure the endpoint using Firebase Auth dependency
    try:
        plan = await services.generate_itinerary(req, user_id=user.uid)
        return plan
    except Exception as e:
        # handle exceptions, e.g., AI call failures
        raise HTTPException(status_code=500, detail=str(e))
```

In this route, we optionally use a dependency (`firebase_auth.get_current_user`) to ensure the user is authenticated via Firebase (the function would verify a Firebase ID token from headers and return a user or raise 401). The route then calls a service function to generate the itinerary and returns the result.

Service Logic: In `trip_planner/services.py`, implement `generate_itinerary` to orchestrate calls to AI and maps services. For example:

```
# trip_planner/services.py
from ai_services import gemini_agents
from maps_services import maps_client
from trip_planner.schemas import TripPlan, DayPlan

async def generate_itinerary(request: TripRequest, user_id: str = None) ->
TripPlan:
    destination = request.destination
    # 1. Use Google Maps service to get key info (geolocation or points of
interest)
    coords = maps_client.geocode_city(destination)
    attractions = maps_client.get_top_attractions(destination,
interests=request.interests)

    # 2. Use AI (Gemini model via ADK) to organize itinerary
    ai_agent = gemini_agents.itinerary_agent  # assume we defined an agent for
trip planning
    plan_text = gemini_agents.run_itinerary_agent(ai_agent, destination,
attractions, request)
    # plan_text might be a structured JSON or text describing the day-by-day
plan

    # 3. Parse or convert the AI output into TripPlan Pydantic model
    plan = parse_plan_text(plan_text)
    return plan
```

This pseudo-code demonstrates how the service might combine **maps data** and **AI reasoning**. The `maps_client` could use Google Maps API (via the `googlemaps` library or HTTP calls) to get geolocation and attractions for the destination. The `gemini_agents.itinerary_agent` would be an ADK agent or function that queries the Gemini 2.5 Pro model to structure the trip plan (more on this in the next section). The result is then parsed into the `TripPlan` schema for output.

Google Maps integration: In `maps_services/maps_client.py`, use the Google Maps API to implement helper functions like `geocode_city` or `get_top_attractions`. For example, using the Google Maps Python client:

```
# maps_services/maps_client.py
import os, googlemaps

gmaps = googlemaps.Client(key=os.environ.get("GOOGLE_MAPS_API_KEY"))

def geocode_city(city_name: str):
    result = gmaps.geocode(city_name)
    if result:
        loc = result[0]['geometry']['location']
        return (loc['lat'], loc['lng'])
    return None

def get_top_attractions(city_name: str, interests=None):
    # This could call Google Places API or use a predefined list for demo.
    # For brevity, return a static list or simplified call.
    return ["Central Park", "Metropolitan Museum of Art"] if city_name.lower()
== "new york" else []
```

You would need to enable the appropriate Maps API (Places API, Geocoding API, etc.) and have an API key (stored in the `.env` as `GOOGLE_MAPS_API_KEY`). The `maps_services` module focuses on **geolocation and POI lookups** – crucial for a trip planner to get real-world data.

Authentication module (auth/): The `auth/firebase_auth.py` might implement a dependency to verify incoming requests. For example:

```python
# auth/firebase_auth.py
import firebase_admin
from firebase_admin import auth as firebase_auth
from fastapi import HTTPException, Security
from fastapi.security import HTTPBearer

security = HTTPBearer()  # HTTP Authorization header bearer token

def get_current_user(token: str = Security(security)):
    try:
        decoded_token = firebase_auth.verify_id_token(token.credentials)
        return decoded_token  # or create a user object
    except Exception as e:
        raise HTTPException(status_code=401, detail="Invalid or expired token")
```

This allows securing endpoints by verifying the Firebase ID token (if using Firebase Auth on the frontend). The `firebase_admin` must be initialized (as shown in main.py) with credentials for this to work.

Core config & logging: The `core/config.py` can centralize reading environment variables and provide a config object. It might also handle environment-specific configuration. For example:

```python
# core/config.py
from pydantic import BaseSettings

class Settings(BaseSettings):
    environment: str = "development"
    vertex_project: str
    vertex_region: str
    firebase_project: str
    # ... other settings

    class Config:
        env_file = ".env"

settings = Settings()  # this will load from .env
```

Using **Pydantic's BaseSettings** allows automatic loading from env and validation. You can then import `settings` and use, e.g., `settings.vertex_project`.

For environment-specific support, you might set an `APP_ENV` in the `.env` (dev/staging/prod) and have logic to adjust settings accordingly (or load a different `.env` file). For instance, use a naming convention like `.env.development`, `.env.production` and load the appropriate one based on `APP_ENV`. This ensures the app can run in different configurations (e.g., using a test Firebase project for dev, and a prod project for production).

Logging and error handling: In `core/logging.py`, configure the Python logging module to output useful information. For example, use `logging.basicConfig` or `dictConfig` to set log level, format (with timestamps, module name, etc.), and possibly file handlers. Ensure to log important events, especially:

- Successful initialization of external services (Vertex AI, Firebase, Maps).
- Warnings or errors from API calls (e.g., if a Vertex AI call fails or returns error, log the event).
- Security-related warnings (like failed auth attempts).

FastAPI allows adding exception handlers – you might add a global handler for generic exceptions to return a standardized error response. For example, catching any `Exception` in `generate_itinerary` and re-raising as HTTP 500 (as shown in the route above) ensures the API doesn't crash silently.

By organizing into these modules, the project remains manageable. The **FastAPI app** ties everything together in `main.py`, including routers and any startup config. The clear separation of concerns (routes vs. logic vs. external integrations) makes the codebase easier to test and extend.

4. Agent and Gemini Integration (Google ADK + Vertex AI)

The heart of the AI trip planner is the integration with **Google's Gemini 2.5 Pro LLM**. We use the **Google Agent Development Kit (ADK)** to create agents that interact with the model and possibly each other. Below we demonstrate how to configure an ADK `LlmAgent` with the Gemini model, and how to compose multiple agents for complex tasks. We also cover function tools (for calling custom code like Maps API) and session management for conversation state.

Initializing a Gemini LLM Agent: With the environment variables set (as above), you can create an ADK `LlmAgent` that uses the Gemini model via Vertex AI. Simply pass the model name string to the agent. For example:

```
# ai_services/gemini_agents.py
from google.adk.agents import LlmAgent

# A basic LLM agent using Gemini 2.5 Pro model
trip_planner_agent = LlmAgent(
    model="gemini-2.5-pro",  # model identifier (ensure Vertex env is
configured)
    name="trip_planner_agent",
    instruction="You are an AI trip planner. Help plan personalized travel
itineraries."
)
```

Under the hood, ADK will resolve `"gemini-2.5-pro"` to the appropriate Vertex AI model endpoint (since we set `GOOGLE_GENAI_USE_VERTEXAI=TRUE`) using the credentials and project info from the environment [7]. You should always use a valid model ID – check Google's documentation for the exact model string; for instance, a preview version might have a suffix [9]. In this example, we assume the generally available `gemini-2.5-pro` is accessible. (If it's a preview model, you might use the specific name like `"gemini-2.5-pro-preview-03-25"` as given in ADK docs [10].)

Multi-agent pipelines: One strength of ADK is the ability to compose agents. For instance, you might want one agent to gather info, another to process it. ADK provides orchestrator agents such as `SequentialAgent` and `ParallelAgent` to run sub-agents in sequence or in parallel.

- **SequentialAgent:** Executes sub-agents one after another (the output of one can feed the next). For example, if planning a trip, one sub-agent could fetch points of interest, and the next could form an itinerary from those points. ADK passes a shared context to each sub-agent so they can share data via session state:

```
from google.adk.agents import SequentialAgent, LlmAgent

step1 = LlmAgent(name="Step1_FetchPOIs", output_key="attractions",
                 instruction="List 5 popular attractions in {city}.")
step2 = LlmAgent(name="Step2_PlanItinerary",
                 instruction="Using the attractions
{attractions}, create a 3-day itinerary for {city}.")

itinerary_pipeline = SequentialAgent(name="ItineraryPipeline",
sub_agents=[step1, step2])
```

In this pipeline, **Step1** might query the model for attractions and save the result to `state['attractions']` (because we set `output_key="attractions"`), then **Step2** can access that value in its prompt by referencing `{attractions}` [11]. The shared `InvocationContext` ensures that `step2` sees the state updated by `step1` [12]. This pattern allows breaking a complex task into simpler chained steps.

- **ParallelAgent:** Executes sub-agents concurrently. This is useful if you have independent tasks to perform with the LLM. For example, you might ask one agent to get today's weather at the destination and another to fetch upcoming events, in parallel, then combine the info. All parallel sub-agents share the same session state (but each gets a branched context for execution):

```
from google.adk.agents import ParallelAgent, LlmAgent

weather_agent = LlmAgent(name="WeatherAgent", output_key="weather",
                         instruction="What is the weather forecast for {city}
during {dates}?")
events_agent = LlmAgent(name="EventsAgent", output_key="events",
                        instruction="List any major events happening in {city}
during {dates}.")
```

```
info_gatherer = ParallelAgent(name="InfoGatherer", sub_agents=[weather_agent,
events_agent])
```

When `info_gatherer` runs, it will launch both `weather_agent` and `events_agent` simultaneously [13]. Once complete, the shared `session.state` will contain both `'weather'` and `'events'` keys filled in. A subsequent agent (or the calling code) can then read `ctx.session.state['weather']` and `['events']` to use that information [14]. This can speed up responses by doing independent LLM calls at the same time.

- **Custom BaseAgent:** While `LlmAgent` covers interactions with LLMs, you can create custom agents by subclassing `BaseAgent` for non-LLM logic or controlling flow. For instance, you might want an agent that checks a condition or formats output. ADK requires implementing an async `_run_async_impl` method that yields `Event` objects. Here's a simple example agent that checks a condition in the shared state and decides whether to continue looping (from ADK docs):

```
from google.adk.agents import BaseAgent
from google.adk.events import Event, EventActions
from google.adk.agents.invocation_context import InvocationContext
from typing import AsyncGenerator

class CheckConditionAgent(BaseAgent):
    async def _run_async_impl(self, ctx: InvocationContext) ->
AsyncGenerator[Event, None]:
        status = ctx.session.state.get("status", "pending")
        done = (status == "completed")
        # Yield an event that signals to escalate (stop loop) if done
        yield Event(author=self.name, actions=EventActions(escalate=done))
```

In this example, the agent reads a `'status'` value from the session state and uses an event with `escalate=True` to indicate a loop should terminate if the status is `"completed"` [15] [16]. This could be used inside a `LoopAgent` to repeatedly run some sub-agents until a condition is met (e.g., keep refining an itinerary until a certain quality threshold is reached, then break). Custom agents let you plug in logic that doesn't involve an LLM call – e.g., checking if the user's budget is exceeded, formatting results, etc.

Shared Session State (`ctx.session.state`): As hinted above, the session state is a dictionary that all agents in the same session can read/write. It's the primary way to pass data between agents. For instance, if one agent gets a list of attractions and saves it in `state['attractions']`, another agent can use that in its prompt or logic [17]. The `LlmAgent.output_key` attribute is a convenient way to automatically store an LLM's output in the state with a given key [18]. Here's a quick conceptual example from the documentation:

```
agent_A = LlmAgent(name="AgentA", instruction="Find the capital of France.",
output_key="capital_city")
agent_B = LlmAgent(name="AgentB", instruction="Tell me about the city
{capital_city}.")
```

```
pipeline = SequentialAgent(name="CityInfo", sub_agents=[agent_A, agent_B])
# When executed, AgentA saves "Paris" to session.state['capital_city'],
# then AgentB sees {capital_city} = "Paris" in its instruction [19] .
```

In our trip planner, we use this mechanism extensively – e.g., the pipeline example above passes the fetched attractions to the planning agent via state. **Structured state** can also be used to accumulate results (like building a list of daily plans across agents).

Function calling with ADK tools: The ADK allows the LLM agents to call **Python functions** through tools. This is analogous to "function calling" in other LLM frameworks, enabling the model to delegate certain tasks to code. We will use this to integrate external APIs (like Google Maps). ADK's `FunctionTool` wraps a Python function so that an agent can invoke it.

For example, suppose we want the AI to get real distance or travel time info via Google Maps during itinerary planning. We could write a Python function `get_travel_time(origin, destination)` that calls the Maps API, and then expose it to the agent:

```
from google.adk.tools import FunctionTool

def get_travel_time(origin: str, destination: str) -> str:
    """Returns travel time by car between origin and destination as a string."""
    # (Call Google Distance Matrix API or similar here; omitted for brevity)
    return "Approximately 30 minutes by car."

travel_time_tool = FunctionTool(func=get_travel_time)
```

Once you have a `FunctionTool`, you include it in an `Agent`'s tools list and instruct the agent on when to use it. For instance:

```
from google.adk.agents import Agent

itinerary_agent = Agent(
    model="gemini-2.5-pro",
    name="itinerary_agent",
    instruction=(

"You are a travel assistant. Plan the itinerary and use tools as needed. "
        "If the user asks for travel times, use the 'get_travel_time' tool."
    ),
    tools=[travel_time_tool]
)
```

Now the agent can decide to call `get_travel_time(origin, destination)` via the tool when crafting its response. The ADK intercepts tool calls: the LLM, if it outputs a special format indicating a tool invocation (e.g., a function call), the ADK will execute the corresponding Python function and return the result to the LLM, all transparently. The example from the ADK documentation demonstrates this with a weather and sentiment analysis tool – the agent's instructions guide it on **when to call each tool** [20] [21]. By following that pattern, our itinerary agent could call a Google Maps tool for distances or a "find attractions" tool for local POIs.

Session management with InMemorySessionService: To maintain conversation or workflow state across multiple turns or function calls, ADK uses a Session service. For a simple backend, the **InMemorySessionService** is sufficient – it keeps session data in memory. We initialize a session at the start of a request (or when a conversation begins) and reuse it for continuity. For example, if we want to preserve the itinerary data between user refinements, we'd use the same session ID. Here's how to set up and use the session service:

```
from google.adk.runners import Runner
from google.adk.sessions import InMemorySessionService
from google.genai import types  # for content types (messages)

# One-time setup of session service (could be at app startup)
session_service = InMemorySessionService()

# Create a session (e.g., per user or per request basis)
app_name = "trip_planner_app"
user_id = "user123"            # could use actual user ID
session_id = "trip-session-1"  # unique session identifier for this planning
sequence

await session_service.create_session(app_name=app_name, user_id=user_id,
session_id=session_id)
runner = Runner(agent=itinerary_agent, app_name=app_name,
session_service=session_service)
```

In the above, we create a session and a `Runner` tied to our main agent [22]. The `Runner` is a helper to execute the agent (and its sub-agents) and stream events. To get a response from the agent, prepare a user message and run the agent:

```
# Prepare user input as a Content message
from google.genai.types import Content, Part
query = "Plan a 2-day trip to Paris focused on food and art."
content = Content(role='user', parts=[Part(text=query)])

# Run the agent and collect events
events = runner.run(user_id=user_id, session_id=session_id, new_message=content)
for event in events:
    if event.is_final_response():
        answer = event.content.parts[0].text
        print("Agent Response:", answer)
```

The ADK Runner will handle the asynchronous execution internally (using `run()` or `run_async()` as needed) and yield a stream of `Event` objects [23]. We iterate through events to find the final response (there might be intermediate steps or tool invocation events as well). The final output from `itinerary_agent` (stored in `answer`) would be the AI-generated trip plan – which we can then format or return via our API.

In a FastAPI context, you might not stream events but rather get the final result synchronously by collecting `events` as above. You can also integrate the ADK call inside an async route by using `await runner.run_async(...)`. The **session ID** can be something like a combination of user and conversation context (or simply a random UUID per request if you don't need continuity beyond one call). InMemorySessionService will keep the `session.state` alive for that session_id as long as the server is running. For more persistent sessions (or if you plan to scale out the app), you'd implement a persistent SessionService (possibly storing state in a DB or cache). For now, in-memory is fine for local development and initial testing.

Using Vertex AI SDK directly (optional): While ADK is our primary interface, note that you can also use the Vertex AI SDK or `google-generativeai` library directly for simpler calls. For example, using `google.generativeai`:

```
import google.generativeai as genai
genai.configure(api_key=None,  # if using Vertex via ADC, ensure api_key is None
                google_project=os.environ["GOOGLE_CLOUD_PROJECT"],
                google_location=os.environ["GOOGLE_CLOUD_LOCATION"])
response = genai.generate_text(model="models/text-bison-001", prompt="Hello
world")
```

But since our use-case benefits from multi-step reasoning and tool use, the ADK approach is more powerful. The ADK essentially wraps around these calls and provides higher-level constructs like Agents, Tools, Sessions, etc.

5. Example Scripts and Usage Documentation

Include a `scripts/` directory with sample scripts to run and test your AI agents in isolation (outside the web API). This helps during development to ensure the AI behaves as expected. For instance, we can have a `scripts/run_agent_demo.py` that uses the ADK Runner to simulate a conversation with the trip planner agent:

```
# scripts/run_agent_demo.py
import asyncio
from google.genai.types import Content, Part
from ai_services.gemini_agents import itinerary_agent
```

```python
from google.adk.sessions import InMemorySessionService
from google.adk.runners import Runner

async def demo():
    session_service = InMemorySessionService()
    await session_service.create_session(app_name="trip_planner_app",
user_id="demo", session_id="demo-session")
    runner = Runner(agent=itinerary_agent, app_name="trip_planner_app",
session_service=session_service)

    user_question = "Can you plan a 3-day trip to Tokyo for food lovers?"
    content = Content(role='user', parts=[Part(text=user_question)])
    events = runner.run(user_id="demo", session_id="demo-session",
new_message=content)
    for event in events:
        if event.is_final_response():
            print(event.content.parts[0].text)

if __name__ == "__main__":
    asyncio.run(demo())
```

Running this script (with `python scripts/run_agent_demo.py` or via `uv`'s run if configured) will output the AI's trip plan to the console. It's a handy way to iterate on your prompt and agent setup without going through the API endpoint each time. You can create multiple scripts for different purposes, e.g., `scripts/test_tools.py` to quickly verify that your FunctionTool integrations work (by sending a query that triggers the tool).

Documentation (README.md): The repository's README should include comprehensive setup and usage instructions, much of which is covered in this answer. Key points to document:

- **Project Overview:** Explain what the project is (AI trip planner backend) and the tech stack (FastAPI, Vertex AI Gemini, Firebase, etc.).
- **Setup Instructions:** How to install dependencies (mention `uv` usage), how to set up Google Cloud (enable Vertex AI API, create service account), how to configure `.env` variables, and how to run the app. For example, document steps like:
- *Prerequisites:* Google Cloud project with Vertex AI enabled; service account JSON; Firebase project setup.
- *Installation:* commands to clone the repo, install `uv`, run `uv pip install -r requirements.txt` (or `uv pip sync` to sync environment with locked deps).
- *Configuration:* where to put the service account JSON, how to fill out `.env` (maybe refer to `.env.example`).
- *Running the app:* e.g., `uvicorn main:app --reload` for development.
- *Running scripts:* e.g., `python scripts/run_agent_demo.py` for AI demo, etc.

- *Usage:* Explain the API endpoints (like a quick docs: e.g. `POST /api/trip/plan` with JSON body, and what it returns). Because we used FastAPI, we automatically get interactive docs at `/docs` – mention that for easy testing.
- *Troubleshooting:* common issues (e.g., missing credentials, incorrect roles causing authentication errors, etc.).

Make sure to include **examples** in the documentation. For instance, show a sample request JSON for a trip plan and a truncated sample response (itinerary) to illustrate the output format.

By providing these scripts and documentation, other developers can easily run and test the AI pipeline locally, and understand how to integrate or adjust the system for their needs.

6. Security Best Practices

Security is paramount when dealing with API keys and user data:

- **Never commit secrets:** Ensure that `.env` is listed in `.gitignore` so that sensitive information (API keys, service account paths) do not enter version control. Likewise, **do not commit service account JSON files**. Treat them as secrets – in production, prefer using a secret manager or environment variables to inject these values at runtime [24]. For example, use Google Secret Manager to store the service account key securely, rather than a file on disk in plaintext [24].

- **Principle of least privilege:** The service account should have only the permissions it needs. For Vertex AI usage, Vertex AI User role is sufficient; if using Firestore, grant Firestore user or editor roles as needed. Avoid using a broad project Editor role. This minimizes damage if the credentials are ever compromised.

- **API Key restrictions:** If using a Google Maps API key, restrict it to required services and domains/IPs. Though the backend uses it server-side, you can still restrict by IP or application. Document in your README how to set these restrictions in Google Cloud Console.

- **Secure Firebase configuration:** If using Firebase Auth, use the Admin SDK on the backend to verify tokens rather than trusting any client input. This we did with `firebase_auth.verify_id_token` which ensures the token's signature is valid and not expired. Also, enforce HTTPS in production for calls to the API so tokens aren't intercepted.

- **Error handling:** Don't leak sensitive info in error messages. In our HTTPException for a caught exception, we currently return `str(e)` which might be too revealing (e.g., stack traces or key names if an error happens). In production, log the detailed error on the server, but return a generic message to the client (or a specific error code). FastAPI allows adding custom exception handlers to format this nicely.

- **CORS and rate limiting:** Since this is an API likely consumed by a frontend (maybe a web app or mobile app), configure CORS appropriately (e.g., allow the frontend origin). Consider implementing rate limiting or authentication on the endpoint so that the expensive AI calls are not abused. If using

Firebase Auth, every request is tied to a user token, which is good for identifying users and possibly gating usage.

By following these practices, you ensure that your development secrets remain safe and the deployed application is robust against common security pitfalls.

7. Testing and Finalizing the Project

Testing: Set up a minimal test suite to validate key parts of the system:

- **Unit tests for service logic:** For example, test that `generate_itinerary` returns a `TripPlan` given certain inputs. Because that function calls external services, in tests you can **mock the AI and Maps calls**. For instance, monkeypatch `gemini_agents.run_itinerary_agent` to return a predetermined output, and monkeypatch `maps_client.get_top_attractions` to return a sample list, then assert that `generate_itinerary` produces the expected `TripPlan` structure.

- **Integration tests for API endpoints:** Use FastAPI's TestClient to simulate HTTP requests. For example, test the `/api/trip/plan` endpoint with a dummy request:

```
from fastapi.testclient import TestClient
from main import app

client = TestClient(app)

def test_plan_trip_endpoint(monkeypatch):
    # Monkeypatch the service to avoid calling real external APIs
    async def fake_generate_itinerary(req, user_id=None):
        return {
            "destination": req.destination,
            "days": [ {"day": 1, "activities": ["Activity A", "Activity B"]} ]
        }
    monkeypatch.setattr("trip_planner.services.generate_itinerary",
fake_generate_itinerary)

    response = client.post("/api/trip/plan", json={
        "destination": "Paris",
        "start_date": "2025-12-01",
        "end_date": "2025-12-03",
        "interests": ["art", "food"]
    })
    assert response.status_code == 200
    data = response.json()
    assert data["destination"] == "Paris"
    assert "days" in data and len(data["days"]) == 1
```

This test overrides `generate_itinerary` with a fake implementation to make the test deterministic and then verifies the API response format. Similar tests can be written for auth (e.g., simulate missing or invalid token to ensure a 401 is returned).

- **AI agent tests:** Testing the LLM itself can be tricky (as it's non-deterministic and depends on external service). However, you can test the integration points: for instance, ensure that when a certain tool function is called, the FunctionTool returns expected output, or that the `itinerary_agent` is set up with the correct model and instructions. If you want, you could use a smaller local model or a dry-run mode for CI, but given this is an integration-heavy component, focus on the above units for automation. The ADK and Vertex calls can be manually tested or in staging environment rather than in unit tests.

After writing tests, you can run them (e.g., `pytest tests/`). They should all pass when the system is correctly set up.

Final Git commit: Once everything is in place – code, config files (except secrets), tests, documentation – it's time to commit and push to your repository. Do a final review to ensure:

- `.env` and any key files are in `.gitignore` (verify they won't be included by running `git status`).
- All necessary files for others to run the project are committed: source code, `requirements.txt` (or equivalent lockfile), the README, and example env file.
- The README clearly explains how to get started, so a new developer or teammate can follow it step-by-step.

Use a meaningful commit message, e.g. *"Initial project setup with FastAPI, Vertex AI (Gemini 2.5 Pro) integration, Firebase auth, and Google ADK agents."* When pushing the code, ensure your repository is private if it contains any remaining sensitive config.

With this, your AI-powered trip planner backend is fully set up for local or self-hosted deployment. You have a robust environment isolation with `uv`, secure configuration via environment variables, a clean project architecture, integrated AI agents using Google's latest LLM via Vertex AI, and supporting services like Firebase and Google Maps. The documentation and tests will help others understand and trust the system. Happy coding and bon voyage for your users' travel plans!

Sources:

- Astral's uv Package – *Next-gen Python Package Manager* [1]
- Google ADK Documentation – *Using Gemini Models with Service Accounts* [5] [10]
- Google ADK Examples – *Sequential and Parallel Agents, Shared State* [11] [14] [19]
- Google ADK Examples – *Custom Agent and Loop control* [15]
- Google ADK Tools Example – *FunctionTool usage with Agents* [20] [21]
- Google ADK Session Example – *InMemorySessionService and Runner* [22] [23]
- GitHub Issue (ADK) – *Required environment variables for Vertex AI authentication* [7] [8]
- Google Cloud Vertex AI Tips – *Use service account credentials and never expose them publicly* [24]

[1] [2] uv: Python packaging in Rust
https://astral.sh/blog/uv

[3] [4] [5] [9] [10] [24] Models & Authentication - Agent Development Kit
https://google.github.io/adk-docs/agents/models/

[6] [7] [8] ADK 0.5.0: LlmAgent with Gemini model fails Vertex AI ADC auth despite GOOGLE_GENAI_USE_VERTEXAI=TRUE · Issue #718 · google/adk-python · GitHub
https://github.com/google/adk-python/issues/718

[11] [12] [13] [14] [15] [16] [17] [18] [19] [20] [21] [22] [23] better-llm.txt
file://file-BAACQ1D7r8ZHvFiGr74dox