MOBTEREST STUDIO

AI Full-Stack Dart with Genkit

A Practical Guide by MOBTEREST STUDIO

Flows · Prompts · Tools · Auth · Deployment

> *A practical guide to building production-ready AI-powered backends in Dart. Each chapter covers one core concept from your first typed flow to a live Cloud Run deployment. The code is real, tested on device, and reflects everything that was learned along the way including the Gemini API constraints, the Dart-specific edge cases, and the patterns that hold up in production.*

Before You Start

- Every chapter builds on the previous one. The code at the end of each chapter is the starting point for the next.
- The confirmed working model string throughout this guide is `googleai/gemini-2.0-flash`. The string `gemini-2.5-flash` is deprecated for new API key holders.
- Tool calling and a JSON output schema (`output: schema:` in a Dotprompt file) cannot be combined in the same Gemini call. This guide uses `response.text` for tool-calling flows and `response.output` for structured flows.
- Context in Genkit is a side channel for your data that your Dart code reads but the model never sees. The pattern is explained fully in Chapter 7.
- Run `dart run build_runner build` whenever you change a class annotated with `@Schema()`.

Contents

CHAPTER 01

The Architecture

Keywords: Why Genkit Dart and Pattern 3

The is the most common starting point when adding AI to a Flutter app.
The Flutter app makes a direct call to the AI model, the API key lives inside the app, and everything routes through the phone. This works for a prototype but fails in production.

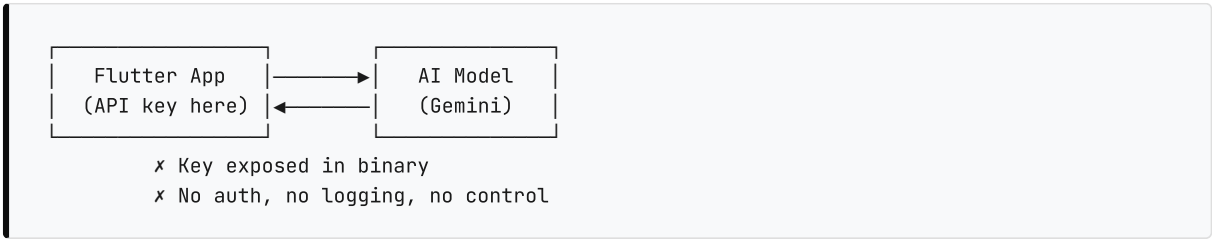
Mobile apps are not secrets. Anyone who downloads your app can decompile it and extract strings from the binary. If your API key is there, someone will find it and when they do, they can use it without restriction, burning through your quota or running up your bill.

Beyond security, the direct-call pattern has no control layer. There is no place to add logging, rate limiting, user identity, or business logic between the Flutter app and the model.

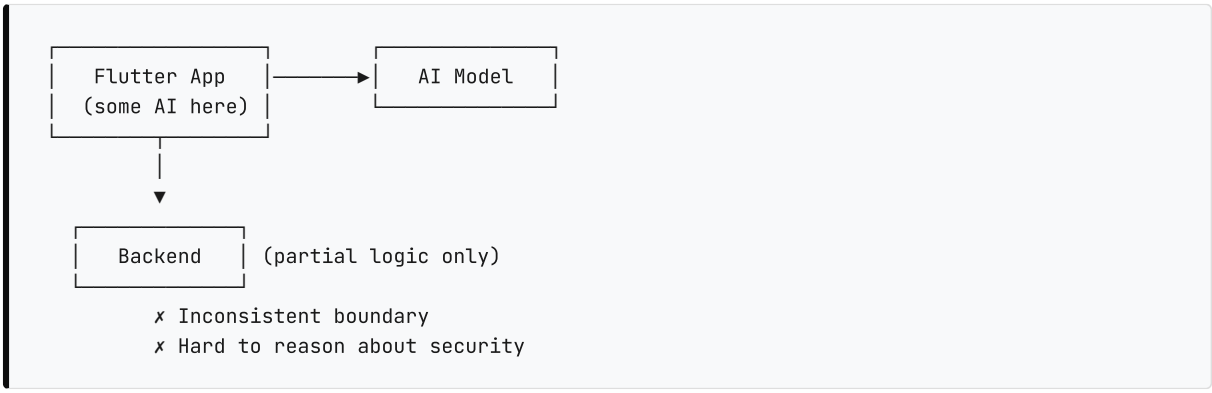
The Three Patterns

There are three common ways to connect Flutter and an AI model:

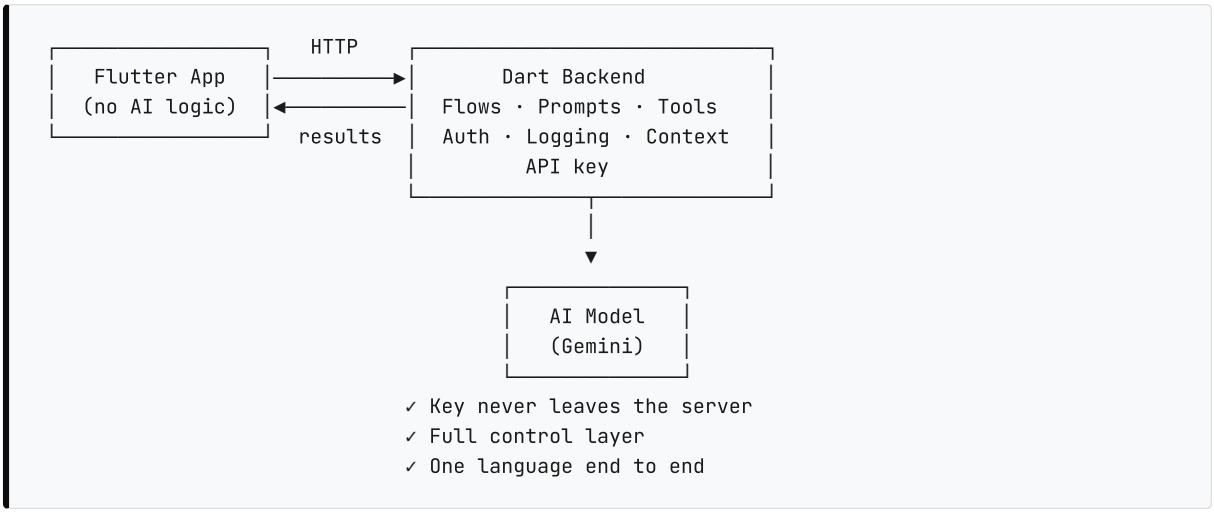
Pattern 1 — Direct client call. Flutter calls the AI model directly. The API key lives in the app binary. Fast to prototype, dangerous to ship. There is no control layer, no server, no security boundary.



Pattern 2 — Hybrid. Some AI logic runs on the client, some on a server. The split is inconsistent and harder to reason about. Keys may still be partially exposed depending on which calls remain on the client.



Pattern 3 — Full server boundary. Flutter is a pure HTTP client. All AI logic (flows, prompts, tools, keys) lives on a Dart backend. The phone never speaks to the model directly. This is the architecture this guide teaches.



Pattern 3: The Right Architecture

Pattern 3 puts a Dart backend between Flutter and the AI model. The Flutter app is a pure HTTP client; it calls endpoints, receives typed results, and knows nothing about which model is running or how the prompts are structured.

The Dart backend, built with Dart Shelf and powered by Genkit, owns everything: the prompts, the flows, the AI model configuration, the API keys, and the logic that decides what the AI does with each request. The AI model, whether Gemini, OpenAI, GPT, Claude, or any other supported provider, never speaks to the phone. The credentials never leave the server.

What the Backend Owns

Flows are the fundamental unit of work in Genkit. Every AI action is wrapped in a flow; it has a name, a typed input, and a typed output. Genkit validates both ends at runtime, traces every execution, and makes every flow callable directly from the Genkit developer UI without touching the Flutter app.

Dotprompt moves prompts out of Dart code and into their own versioned files. Each file carries YAML frontmatter that declares the model, temperature, and input and output schemas. Prompts become source artifacts you can iterate on, commit to git, and change without restarting the server.

Tools give the AI reach into your own data. You define a Dart function; query a database, call an external API, read a file, and give it a name and a description. The model reads the description, decides whether calling the function would help it give a better answer, and Genkit executes it automatically. The model reasons with real data, not just what you put in the prompt.

Auth is the gate. A Shelf middleware checks every incoming request before it reaches any flow. Requests without the correct credentials are rejected immediately with a 401. The AI model is never called. Quota is never burned.

Lumina: The App We Build

Lumina is an AI-powered journaling application built for self-reflection. The premise is simple: **you write about your day, and the app helps you understand it better by responding to you in real time with two distinct AI outputs.**

What the user does

The user opens the app, writes a free-form journal entry in a text field (anything from a sentence to a paragraph) and taps the Analyse button. That single action triggers the entire AI pipeline on the backend.

What the app returns

The app responds with two things:

The first is a **mood chip:** a label like *drained*, *reflective*, *hopeful*, or *anxious*, paired with a confidence score shown as a percentage. This comes from `analyzeMoodFlow`, which reads the journal entry and returns a structured object with the mood label and score. The output is typed: `MoodOutput` has a `mood` field (string) and a `score` field (double from 0.0 to 1.0). The chip appears as soon as the analysis completes.

The second is a **coaching question** — a single open-ended reflection question generated by `journalCoachFlow`, written specifically for that entry. Unlike a generic prompt, the question is informed by the user's past journal history: the flow calls a `getRecentEntries` tool that queries the Drift database, retrieves the last few entries for that user, and gives the AI context before it responds. The question is returned as a complete response personalised to that user's history and tailored to the specific entry they just wrote.

Every piece of this **(the flows, the tool, the database, the context, the auth)**, is built step by step through this guide. By the end, you will have written all of it yourself.

The inputs and outputs at a glance

Step	Input	Output
User writes	Free-form text entry	—
`analyzeMoodFlow`	Entry text	Mood label + confidence score
`journalCoachFlow`	Entry text + past entries (via tool)	Reflection question
Flutter UI	Typed responses	Mood chip + reflection question card

One screen. Two flows. One Dart backend.

CHAPTER 02

Flows & Schemas

Keywords: defineFlow, schemantic, build_runner

A Genkit flow is a function with structure. It takes a typed input, does something (in our case, calls an AI model), and returns a typed output. Genkit validates both ends at runtime, traces the execution, and makes every flow callable directly from the Genkit developer UI.

Before any model calls happen, this chapter is about building that structure.

Project Setup

Create a new Dart project for the backend:

```
dart create lumina_backend
cd lumina_backend
```

Add the Genkit and schemantic packages:

```
dart pub add genkit schemantic
dart pub add --dev schemantic_builder build_runner
```

Three packages are at work here:

- `schemantic` — the schema generation package. You annotate abstract Dart classes with `@Schema()` and it generates concrete classes, JSON serialisers, and Genkit-compatible schema definitions.
- `schemantic_builder` — the code generator for schemantic. It reads the `@Schema()` annotations and produces the `.g.dart` output files.
- `build_runner` — the dev tool that runs the code generator. You invoke it manually whenever you change an annotated class.

Our First Flow: analyzeMoodFlow

Before writing any code, it helps to be clear about what this flow does.

`analyzeMoodFlow` is the first of Lumina's two AI flows. It takes a journal entry as input, sends it to the AI model with a prompt asking for a mood analysis, and returns a structured result: a mood label and a confidence score. That is it. One input, one output, one purpose.

Here is what a call to this flow looks like end to end:

```
Input:
  entryText: "I woke up later than usual. I was too exhausted
              from yesterday's workload."

Output:
  mood:  "drained"
  score:  0.90
```

The mood label comes from a defined set — `reflective`, `anxious`, `hopeful`, `heavy`, `energised`, `drained`, `content`, `unsettled`. The score is a double from 0.0 to 1.0 representing the model's confidence in that label. Together they power the mood chip in the Flutter UI.

Now that we know what the flow does, we can define the schemas that describe its inputs and outputs.

Defining Schemas

Every flow needs an input schema and an output schema. For `analyzeMoodFlow`, the input is a journal entry text and the output is a mood label with a confidence score.

Add these to `bin/lumina_backend.dart`:

```
import 'package:genkit/genkit.dart';
import 'package:schemantic/schemantic.dart';

part 'lumina_backend.g.dart';

@Schema()
abstract class $MoodInput {
  /// The raw text of the journal entry to analyse.
  String get entryText;
}

@Schema()
abstract class $MoodOutput {
  /// The detected mood label.
  String get mood;

  /// Confidence score from 0.0 to 1.0.
  double get score;
}
```

The `$` prefix on `$MoodInput` and `$MoodOutput` is the schemantic convention; it signals to `build_runner` that these abstract classes need concrete implementations generated. The generated file lands in `lumina_backend.g.dart`.

Run the generator:

```
dart run build_runner build
```

After this, `MoodInput` and `MoodOutput` (without the `$`) are valid Dart classes. Genkit also receives `MoodInput.$schema` and `MoodOutput.$schema`; the runtime schema descriptors it uses for validation.

> **WARNING**
> If you change a class annotated with `@Schema()`, re-run `dart run build_runner build` before running the server. Stale generated classes cause type mismatch errors that are confusing to debug.

Defining the Flow

```
void main() async {
  // The plugins list is where you declare your AI provider.
  // An empty list means no model is wired yet — we will add
  // the Gemini plugin in Chapter 3. This is also where you
  // would switch to OpenAI, Claude, or any other supported
  // provider by swapping the plugin.
  final ai = Genkit(plugins: []);

  final sc-camel-analyze-mood-flow = ai.defineFlow(
    name: 'analyzeMoodFlow',
    inputSchema: MoodInput.$schema,
    outputSchema: MoodOutput.$schema,
    fn: (MoodInput input, _) async {
      // Gemini is wired in Chapter 3.
      // For now, return a hardcoded result to verify the structure.
      return MoodOutput(
        mood: 'reflective',
        score: 0.8,
      );
    },
  );

  final result = await analyzeMoodFlow(
    MoodInput(entryText: 'Today felt heavy but productive.'),
  );
  print('Mood: ${result.mood}, Score: ${result.score}');
}
```

Run it:

```
dart run bin/lumina_backend.dart
```

> **NOTE**
> The `fn` callback receives two parameters: the typed input you defined (e.g. `MoodInput`) and a second parameter that carries Genkit's execution context. The context is how user identity and other request-level data travel through the flow without being exposed to the AI model. We cover it fully in Chapter 7. For now, name it `_` to tell Dart you are intentionally ignoring it; the code will not compile if you leave the second parameter out entirely.

The Second Flow: journalCoachFlow

Lumina needs a second flow (`journalCoachFlow`) that generates the personalised reflection question. Where `analyzeMoodFlow` returns structured data, `journalCoachFlow` returns a single open-ended question written specifically for the entry, for example:

```
Input:
  entryText: "I woke up later than usual. I was too exhausted
              from yesterday's workload."

Output:
  question: "What boundary could you set tomorrow to
             protect your energy?"
```

Add its schemas alongside the mood schemas:

```
@Schema()
abstract class $CoachInput {
  String get entryText;
}

@Schema()
abstract class $CoachOutput {
  String get question;
}
```

Run `build_runner` again after adding schemas:

```
dart run build_runner build
```

Define `journalCoachFlow` with the same `ai.defineFlow` pattern alongside `analyzeMoodFlow` in `bin/lumina_backend.dart`:

```
final sc-camel-journal-coach-flow = ai.defineFlow(
  name: 'journalCoachFlow',
  inputSchema: CoachInput.$schema,
  outputSchema: CoachOutput.$schema,
  fn: (CoachInput input, _) async {
    // The AI model and prompt are wired in Chapter 5 (Dotprompt).
    // For now, return a hardcoded result to verify the structure.
    return CoachOutput(
      question: 'What would make tomorrow feel more manageable?',
    );
  },
);
```

CHAPTER 03

Generating Content

Keywords: ai.generate(), Gemini plugin, Dev UI

`analyzeMoodFlow` is defined and structured. But the `fn` body returns hardcoded values. This chapter wires Gemini in and introduces the Genkit Developer UI, the local tool you will use to test and debug every flow in this guide.

Plugin Setup and API Key

Add the Google AI plugin:

```
dart pub add genkit_google_genai
```

Get your Gemini API key from **aistudio.google.com**. Set it as an environment variable and never in source code:

```
# macOS / Linux
export GEMINI_API_KEY=your_key_here

# Windows (PowerShell)
$env:GEMINI_API_KEY="your_key_here"
```

The `googleAI()` plugin reads `GEMINI_API_KEY` from the environment automatically. Pass no key in code; it never appears in source and works identically locally and on Cloud Run.

Genkit Dart supports multiple AI providers through a plugin system. Each provider has its own package and plugin function:

Provider	Package	Plugin
Google Generative AI	`genkit_google_genai`	`googleAI()`
Google Vertex AI	`genkit_vertexai`	`vertexAI()`
OpenAI	`genkit_openai`	`openAI()`
Anthropic	`genkit_anthropic`	`anthropic()`

Install only the provider you need:

```
# Google Generative AI (used in this guide)
dart pub add genkit_google_genai

# Google Vertex AI
dart pub add genkit_vertexai

# OpenAI
dart pub add genkit_openai

# Anthropic
dart pub add genkit_anthropic
```

Swapping providers is a one-line change; replace the plugin in `Genkit(plugins: [...])` and update the model string. The flow code itself does not change.

Update `main()` to initialise with the plugin:

```
import 'package:genkit_google_genai/genkit_google_genai.dart';

final ai = Genkit(plugins: [googleAI()]);
```

ai.generate()

`ai.generate()` is the core generation method. Its three key parameters:

- **model** — which model to call. Use `googleAI.gemini('gemini-2.0-flash')`.
- **prompt** — the text instruction sent to the model.
- **outputSchema** — when present, Genkit instructs the model to return structured JSON matching the schema, and `response.output` returns a typed Dart object. Without it, `response.output` is null — use `response.text` instead.

Update the `analyzeMoodFlow` `fn` — the function body that runs every time the flow is called. This is where the actual AI call lives. Replace the hardcoded return from Chapter 2 with a real `ai.generate()` call:

```
fn: (MoodInput input, _) async {
  final response = await ai.generate(
    model: googleAI.gemini('gemini-2.0-flash'),
    prompt: 'Analyse the mood of this journal entry. '
            'Return a mood label (reflective, anxious, hopeful, '
            'heavy, energised, drained, content, unsettled) and '
            'a confidence score between 0.0 and 1.0. '
            'Entry: "${input.entryText}"',
    outputSchema: MoodOutput.$schema,
  );
  // response.output is a typed MoodOutput validated by Genkit.
  return response.output!;
},
```

The Genkit Developer UI

Install the Genkit CLI:

```
# macOS / Linux — recommended
curl -sL cli.genkit.dev | bash

# or via npm if you have Node.js installed
npm install -g genkit-cli

# Windows — download the binary from:
# https://cli.genkit.dev
```

Now, instead of running your app with `dart run`, run it through the Genkit CLI:

```
genkit start -- dart run bin/lumina_backend.dart
```

You will see output like this:

```
Genkit Developer UI started
Open http://localhost:4000 in your browser
Starting Dart app...
Flow registered: analyzeMoodFlow
```

The Genkit developer UI opens at `http://localhost:4000`. Every flow you have defined appears in the sidebar. Click any flow, type an input, and run it — you get the typed output and a full execution trace showing every model call, every token count, and the exact response. No app restart required.

This is the primary development loop for AI features: define a flow, run it in the Dev UI, inspect the trace, iterate on the prompt, repeat.

> **NOTE**
> The `--` separator in the `genkit start` command tells the CLI that everything after it is the command to wrap. Run `dart run` directly and you lose the Dev UI.

CHAPTER 04

The Shelf Server

Keywords: shelfHandler, Router, Pipeline, CORS

The flows work. Now they need to be reachable over HTTP. This chapter wraps the Dart backend in a Shelf server and exposes `analyzeMoodFlow` and `journalCoachFlow` as HTTP endpoints.

Dependencies

Add the Shelf packages:

```
dart pub add shelf shelf_router shelf_cors_headers genkit_shelf
```

The HTTP Contract

Every Genkit Shelf endpoint follows the same request and response shape:

```
// Request body
{"data": {"fieldName": value}}

// Response body
{"result": {"field": value}}
```

`shelfHandler` enforces this contract automatically. On the Flutter side, wrap every input in `{"data": ...}` and extract from `{"result": ...}` in the response.

The Complete Server

```
import 'dart:io';
import 'package:shelf/shelf.dart';
import 'package:shelf/shelf_io.dart' as io;
import 'package:shelf_router/shelf_router.dart';
import 'package:shelf_cors_headers/shelf_cors_headers.dart';
import 'package:genkit_shelf/genkit_shelf.dart';

void main() async {
  final ai = Genkit(plugins: [googleAI()]);

  // Flows defined here (see Chapters 2 and 3)

  final router = Router()
    ..post('/analyzeMood', shelfHandler(analyzeMoodFlow))
    ..post('/journalCoach', shelfHandler(journalCoachFlow));

  final handler = Pipeline()
      .addMiddleware(corsHeaders())
      .addHandler(router.call);

  await io.serve(handler, '0.0.0.0', 3400);
  print('Lumina backend running on http://localhost:3400');
}
```

Test it:

```
curl -X POST http://localhost:3400/analyzeMood \
  -H "Content-Type: application/json" \
  -d '{"data": {"entryText": "Today felt heavy but productive."}}'
```

Expected response:

```
{"result": {"mood": "reflective", "score": 0.74}}
```

> **NOTE**
> Bind to `0.0.0.0`, not `localhost`. Cloud Run requires the server to accept external traffic. The binding is correct from the start — it does not need to change at deployment.

CHAPTER 05

Dotprompt

Keywords: Prompt files, frontmatter, Handlebars

The prompt for `analyzeMoodFlow` is currently a hardcoded string inside a Dart function. Every change means editing code, restarting the server, and re-testing. The AI instruction is buried inside application logic with no separation and no history.

Dotprompt fixes this. Prompts become files. Model configuration moves into frontmatter. The Dart flow body shrinks. And because prompts are files, they version with git like any other source artifact.

Anatomy of a .prompt File

A Dotprompt file has two sections separated by `---` delimiters.

YAML frontmatter declares the model, temperature, max tokens, and the input and output schemas using Picoschema — a compact format where each field is `fieldName: type, description`.

The template is the prompt text. Input values are inserted with Handlebars `{{variableName}}` syntax.

Prompt Versioning

Because prompts are files, they version exactly like source code. Add both `.prompt` files to git:

```
git add prompts/
git commit -m "add analyzeMood.prompt and journalCoach.prompt — Lesson 05"
```

From this point forward, every prompt iteration becomes a commit:

```
# Example — refining mood labels
git log --oneline prompts/
# a3f9c12  refine mood labels — add 'heavy' and 'drained'
# b8e21a4  lower temperature to 0.2 for more consistent scoring
# c1d4087  add journalCoach.prompt — reflection question flow
# d2f1993  initial analyzeMood.prompt
```

You can diff prompt versions exactly like code:

```
git diff HEAD~1 prompts/analyzeMood.prompt
# - temperature: 0.3
# + temperature: 0.2
# - Return a mood label and confidence score...
# + Return one of: reflective, anxious, hopeful, heavy, energised...
```

This is what "prompts as code" means in practice. Every change is intentional, traceable, and reversible.

analyzeMood.prompt

Create a `prompts/` folder at the project root. Add `analyzeMood.prompt`:

```
---
model: googleai/gemini-2.0-flash
config:
  temperature: 0.3
  maxOutputTokens: 256
input:
  schema:
    entryText: string, The raw text of the journal entry
output:
  schema:
    mood: string, The mood label
    score: number, Confidence score from 0.0 to 1.0
---
Analyse the mood of this journal entry.

Return one label from: reflective, anxious, hopeful, heavy,
energised, drained, content, unsettled.

Return a confidence score between 0.0 and 1.0.

Entry: {{entryText}}
```

The output schema in the frontmatter tells Genkit to instruct the model to return structured JSON. With it declared, `response.output` is a `Map<String, dynamic>` you parse with `fromJson`.

> **WARNING**
> `output: schema:` and tool calling cannot be combined in the same Gemini call. If a flow uses tools, omit the output schema from the prompt file and use `response.text` instead.

Loading a Prompt in Dart

Replace the inline `ai.generate()` call with a Dotprompt call:

```
fn: (MoodInput input, _) async {
  final sc-camel-analyze-mood = await ai.prompt('analyzeMood');
  final response = await analyzeMood({'entryText': input.entryText});
  return MoodOutput.fromJson(response.output! as Map<String, dynamic>);
},
```

`ai.prompt('analyzeMood')` is async — it reads `prompts/analyzeMood.prompt` from disk. The filename (without extension) must match exactly, case-sensitive.

> **NOTE**
> `response.output` is a typed Dart object only when the prompt file declares `output: schema:`. Without it, `response.output` is null. Use `response.text` to read the raw string the model returned.

journalCoach.prompt

The coaching flow does not use `output: schema:` because it uses tool calling (Chapter 6). Tool calling and structured JSON output cannot be combined in a single Gemini call.

```
---
model: googleai/gemini-2.0-flash
config:
  temperature: 0.7
  maxOutputTokens: 256
input:
  schema:
    entryText: string, The current journal entry text
---
You are a thoughtful journaling coach.

Read this journal entry and generate one open-ended reflection
question that helps the writer explore their thoughts more deeply.
Be specific, warm, and curious.

Entry: {{entryText}}
```

> **WARNING**
> Prompts are read from disk at runtime, including in production. The `prompts/` directory must be copied into the Docker image (Chapter 10). This is the most common deployment mistake with Dotprompt.

CHAPTER 06

Tool Calling & Drift

Keywords: defineTool, getRecentEntries, Drift DAO

A language model only knows what you put in the prompt. Without additional context, the coaching question is generic; the model generates based on one entry with no history. For example, given the entry "I woke up later than usual. I was too exhausted from yesterday's workload.", a model with no history might return:

```
"What made today feel challenging?"
```

With access to the user's past entries, knowing they have written about exhaustion three days in a row, it can return something far more specific:

```
"What boundary could you set tomorrow to protect your energy?"
```

That difference is what tools make possible.

This chapter adds a Drift SQLite database to store journal entries, and a `getRecentEntries` tool that lets the model query that database before generating a reflection question.

Drift Setup

```
dart pub add drift sqlite3 path
dart pub add --dev drift_dev
```

Create `lib/database.dart`:

```
import 'package:drift/drift.dart';
import 'package:drift/native.dart';
import 'package:path/path.dart' as p;
import 'dart:io';

part 'database.g.dart';

class JournalEntries extends Table {
  IntColumn get id ⇒ integer().autoIncrement()();
  TextColumn get sc-camel-user-id ⇒ text()();
  TextColumn get sc-camel-entry-text ⇒ text()();
  DateTimeColumn get sc-camel-created-at ⇒ dateTime()();
}

@DriftDatabase(tables: [JournalEntries])
class LuminaDatabase extends _$LuminaDatabase {
  LuminaDatabase() : super(_openConnection());

  @override int get sc-camel-schema-version ⇒ 1;

  Future<List<JournalEntry>> getRecent(String userId, {int limit = 5}) ⇒
    (select(journalEntries)
      ..where((t) ⇒ t.userId.equals(userId))
      ..orderBy([(t) ⇒ OrderingTerm.desc(t.createdAt)])
      ..limit(limit))
      .get();

  Future<void> insertEntry(String userId, String text) ⇒
    into(journalEntries).insert(JournalEntriesCompanion.insert(
      userId: userId,
      entryText: text,
      createdAt: DateTime.now(),
    ));
}

LazyDatabase _openConnection() ⇒ LazyDatabase(() async {
  final sc-camel-db-file = File(p.join(Directory.current.path, 'lumina.db'));
  return NativeDatabase.createInBackground(dbFile);
});
```

Run the generator:

```
dart run build_runner build
```

Defining the Tool

Before defining the tool itself, add the input schema for it in `bin/lumina_backend.dart` alongside the other schemas, then run `build_runner`:

```
@Schema()
abstract class $RecentEntriesInput {
  /// The user ID to fetch entries for.
  String get userId;

  /// Maximum number of entries to fetch. Default is 5.
  int get limit;
}
```

```
dart run build_runner build
```

`ai.defineTool` takes a name, a description, an input schema, and a function. The description is what the model reads to decide whether to call the tool; write it clearly and specifically.

```
final sc-camel-get-recent-entries = ai.defineTool(
  name: 'getRecentEntries',
  description: 'Fetches the most recent journal entries for the '
               'current user to provide historical information for '
               'coaching. Call this before generating a reflection '
               'question.',
  inputSchema: RecentEntriesInput.$schema,
  fn: (RecentEntriesInput input, _) async {
    final entries = await db.entriesDao.getRecent(
      input.userId,
      limit: input.limit,
    );

    if (entries.isEmpty) return 'No previous entries found.';

    return entries.map((e) => '- ${e.entryText}').join('\n');
  },
);
```

The tool reads `userId` from its input schema; passed in directly by the flow. Context propagation, which allows `userId` to travel invisibly through the execution without being passed explicitly, is covered in Chapter 7.

Passing the Tool to journalCoachFlow

```
fn: (CoachInput input, context) async {
  final sc-camel-journal-coach = await ai.prompt('journalCoach');
  final response = await journalCoach(
    {
      'entryText': input.entryText,
      'userId': input.userId, // ← pass userId to the prompt template
    },
    PromptGenerateOptions(tools: [getRecentEntries]),
  );

  // response.text, not response.output —
  // tool calling and output: schema: cannot be combined.
  return CoachOutput(question: response.text.trim());
},
```

NOTE

The tool description quality determines whether the model calls it. A vague description like "gets entries" often results in the tool never being called. A specific description that tells the model when and why to call it works reliably.

CHAPTER 07

Middleware & Context

Keywords: Logging, contextProvider, userId

The backend has flows, tools, and a database. Before connecting Flutter, two production concerns need to be addressed: visibility into what is happening on the server, and user identity that travels safely through the execution without touching the model.

Three Categories of Information in Genkit

Genkit distinguishes three categories of data flowing through an AI operation:

- **Input** — directly guides the model. Goes in the flow input schema. The model sees it.
- **Generation context** — relevant to the model but not request-specific. Injected into the prompt template.
- **Execution context** — important to your code but invisible to the model. This is Genkit context.

`userId` belongs in execution context. If it were in the flow input schema, Genkit would include it in what it sends to Gemini. The model could see it, reference it in a response, or behave unexpectedly with it. Context keeps it on the Dart side, completely invisible to the model.

Logging Middleware

A Shelf middleware is a function that takes a `Handler` and returns a `Handler`. It wraps the inner handler — running code before passing the request through, and more code after getting the response back.

```
Middleware loggingMiddleware() {
  return (Handler innerHandler) {
    return (Request request) async {
      final start = DateTime.now();
      final response = await innerHandler(request);
      final duration = DateTime.now().difference(start);
      print(
        '[${DateTime.now()}] '
        '${request.method} ${request.url} '
        '→ ${response.statusCode} (${duration.inMilliseconds}ms)',
      );
      return response;
    };
  };
}
```

contextProvider

With logging in place, the next step is making user identity available to flows without exposing it to the model.

The `contextProvider` function bridges the HTTP request and Genkit context. It receives the Shelf `Request` object and returns a `Map<String, dynamic>` that Genkit makes available throughout the flow execution — including inside every tool the flow calls.

```
Map<String, dynamic> contextProvider(Request request) {
  final sc-camel-user-id = request.headers['x-user-id'] ?? 'anonymous';
  return {'userId': userId};
}
```

Pass it to each `shelfHandler` call:

```
final router = Router()
  ..post('/analyzeMood',
      shelfHandler(analyzeMoodFlow, contextProvider: contextProvider))
  ..post('/journalCoach',
      shelfHandler(journalCoachFlow, contextProvider: contextProvider));
```

Reading Context in a Flow or Tool

The `fn` parameter's second argument is the context object. Access the map via `context.context?['userId']` — the double-level access is because the context map sits one level inside the Genkit side-channel object:

```
fn: (CoachInput input, context) async {
  // The context parameter is untyped — do not annotate it.
  // Access the map via context.context?, not context[] directly.
  final sc-camel-user-id = context.context?['userId'] as String? ?? 'anonymous';
  // ...
}
```

> **WARNING**
>
> `ActionContext` does not exist in Genkit Dart. Do not annotate the context parameter with a type. Let Dart infer it and access the map via `context.context?['key']`.

Updated Pipeline

```
final handler = Pipeline()
    .addMiddleware(loggingMiddleware())  // runs first
    .addMiddleware(corsHeaders())
    .addHandler(router.call);
```

CHAPTER 08

Authorization

Keywords: authMiddleware, API key, 401 rejection

The endpoints are open. Anyone who discovers the URL can call the flows; burning Gemini quota or probing the API with no restriction. When the backend deploys to Cloud Run, that URL is publicly reachable. Auth goes in now.

authMiddleware

The auth middleware follows the same Shelf pattern as `loggingMiddleware`. It reads the `x-api-key` header and returns a 401 if the key is missing or wrong. Genkit never runs on rejected requests.

```
import 'dart:io';

// expectedKey is the value passed in when adding the middleware
// to the pipeline: .addMiddleware(authMiddleware(apiKey))
// apiKey (read from Platform.environment) becomes expectedKey here.
Middleware authMiddleware(String expectedKey) {
  return (Handler innerHandler) {
    return (Request request) async {
      final key = request.headers['x-api-key'] ?? '';

      if (key ≠ expectedKey) {
        return Response(
          401,
          body: '{"error": "Unauthorized"}',
          headers: {'Content-Type': 'application/json'},
        );
      }

      return innerHandler(request);
    };
  };
}
```

The `?? ''` fallback is deliberate. An empty string expected key means all requests are rejected; the safe failure mode when the environment variable is not set.

API Key as Environment Variable

The API key must never live in source code. Read it from the environment the same way the backend reads the Gemini key:

```
# Set alongside GEMINI_API_KEY
export LUMINA_API_KEY=lumina-secret-2026
```

```
// Read in main() before defining flows
final sc-camel-api-key = Platform.environment['LUMINA_API_KEY'] ?? '';
```

Updated Pipeline — Auth First

Auth middleware must go first. The pipeline runs top to bottom, and a rejected request stops before reaching logging, CORS, or Genkit.

```
final handler = Pipeline()
    .addMiddleware(authMiddleware(apiKey))  // ← first
    .addMiddleware(loggingMiddleware())
    .addMiddleware(corsHeaders())
    .addHandler(router.call);
```

Testing

```
# With key — expect 200
curl -X POST http://localhost:3400/analyzeMood \
  -H "Content-Type: application/json" \
  -H "x-api-key: lumina-secret-2026" \
  -d '{"data": {"entryText": "Today felt heavy."}}'

# Without key — expect 401
curl -X POST http://localhost:3400/analyzeMood \
  -H "Content-Type: application/json" \
  -d '{"data": {"entryText": "Today felt heavy."}}'
```

> **NOTE**
> The 401 request should not appear in the server log; auth rejects it before the logging middleware runs. This confirms the pipeline order is correct.

CHAPTER 09

Flutter Client

Keywords: LuminaService, Future.wait, JournalScreen

The backend is complete. This chapter connects Flutter. By the end, the full Lumina stack runs end to end for the first time.

Flutter Project Setup

```
flutter create lumina_app
cd lumina_app
flutter pub add http
```

The `http` package is the only dependency needed to call the backend flows. Flutter knows nothing about Genkit, schemantic, or build_runner.

Response Models

Create plain Dart model classes in the Flutter project that mirror the backend output schemas:

```
// lib/models/mood_result.dart
class MoodResult {
  final String mood;
  final double score;
  const MoodResult({required this.mood, required this.score});

  factory MoodResult.fromJson(Map<String, dynamic> json) ⇒ MoodResult(
    mood: json['mood'] as String,
    score: (json['score'] as num).toDouble(),
  );
}

class CoachResult {
  final String question;
  const CoachResult({required this.question});

  factory CoachResult.fromJson(Map<String, dynamic> json) ⇒
    CoachResult(question: json['question'] as String);
}
```

LuminaService

```
// lib/services/lumina_service.dart
import 'dart:convert';
```

```
import 'package:http/http.dart' as http;
import '../models/mood_result.dart';

class LuminaService {
  // Update to Cloud Run URL in Chapter 10
  static const String _base = 'http://localhost:3400';
  // Move _apiKey to a --dart-define or env approach before shipping.
  // For local development, hardcoding is acceptable.
  static const String _apiKey = 'lumina-secret-2026';

  final String _userId;
  LuminaService({required String userId}) : _userId = userId;

  Map<String, String> get _headers => {
    'Content-Type': 'application/json',
    'x-api-key': _apiKey,
    'x-user-id': _userId,
  };

  Future<MoodResult> analyzeMood(String entryText) async {
    final response = await http.post(
      Uri.parse('$_base/analyzeMood'),
      headers: _headers,
      body: jsonEncode({'data': {'entryText': entryText}}),
    );
    if (response.statusCode != 200) {
      throw Exception('analyzeMood failed: ${response.statusCode}');
    }
    final result =
        jsonDecode(response.body)['result'] as Map<String, dynamic>;
    return MoodResult.fromJson(result);
  }

  Future<CoachResult> journalCoach(String entryText) async {
    final response = await http.post(
      Uri.parse('$_base/journalCoach'),
      headers: _headers,
      body: jsonEncode({'data': {'entryText': entryText}}),
    );
    if (response.statusCode != 200) {
      throw Exception('journalCoach failed: ${response.statusCode}');
    }
    final result =
        jsonDecode(response.body)['result'] as Map<String, dynamic>;
    return CoachResult.fromJson(result);
  }
}
```

Platform Network Configuration

Android — add to `android/app/src/main/AndroidManifest.xml`:

```
<uses-permission android:name="android.permission.INTERNET"/>
```

iOS — for local development over HTTP, add to `ios/Runner/Info.plist`:

```
<key>NSAppTransportSecurity</key>
<dict>
  <key>NSAllowsLocalNetworking</key>
  <true/>
</dict>
```

> **NOTE**
> Remove `NSAllowsLocalNetworking` after deploying to Cloud Run in Chapter 10. The production endpoint uses HTTPS, which iOS allows by default.

Using Future.wait

Fire both flow calls in parallel — not sequentially. If one takes 300ms and the other 800ms, `Future.wait` takes 800ms total rather than 1100ms:

```
final results = await Future.wait([
  _service.analyzeMood(text),
  _service.journalCoach(text),
]);
setState(() {
  _mood = results[0] as MoodResult;
  _coach = results[1] as CoachResult;
});
```

CHAPTER 10

Deploying to Cloud Run

Keywords: Dockerfile, Secret Manager, gcloud

Lumina is complete as a local application. The backend runs on your machine, Flutter connects to localhost, and the full AI stack works. The final step is to ship it.

Two Backend Changes Before Deploying

Cloud Run injects a `PORT` environment variable at runtime. The server must read it and bind to `InternetAddress.anyIPv4` to accept external traffic.

```
// Updated io.serve call in main()
final port = int.parse(Platform.environment['PORT'] ?? '8080');
await io.serve(handler, InternetAddress.anyIPv4, port);
print('Lumina backend running on port $port');
```

The Dockerfile

A `Dockerfile` is a text file that describes how to package your application into a container image; a self-contained unit that includes your code, its dependencies, and everything it needs to run. Cloud Run takes that image, runs it, and manages the infrastructure around it.

Create a file named `Dockerfile` at the root of your backend project (alongside `pubspec.yaml`). No file extension.

The Lumina Dockerfile uses a **two-stage build**. The first stage compiles the Dart code into a native binary using the full Dart SDK. The second stage copies only that binary into a much smaller runtime image — the Dart SDK itself is not needed to run a compiled binary, so the final image stays lean.

```
# Stage 1: Build
FROM dart:stable AS build
WORKDIR /app
COPY pubspec.* ./
RUN dart pub get
COPY . .
RUN dart pub get --offline
RUN dart compile exe bin/lumina_backend.dart -o bin/lumina_backend

# Stage 2: Runtime
FROM debian:stable-slim
RUN apt-get update && apt-get install -y ca-certificates \
    && rm -rf /var/lib/apt/lists/*
WORKDIR /app
COPY --from=build /runtime/ /
COPY --from=build /app/bin/lumina_backend /app/bin/lumina_backend
COPY --from=build /app/prompts /app/prompts
EXPOSE 8080
CMD ["/app/bin/lumina_backend"]
```

WARNING

The `prompts/` directory must be explicitly copied into the image. This is done with the line `COPY --from=build /app/prompts /app/prompts` in Stage 2 of the Dockerfile — you can see it in the code above. Without it, Genkit reads `.prompt` files from disk at runtime but finds nothing there, and every Dotprompt call fails with a file-not-found error in production. This is the most common deployment mistake.

NOTE

Use `debian:stable-slim`, not `scratch`. The compiled Dart binary has dynamic library dependencies that `scratch` cannot satisfy. Add `ca-certificates` so outbound HTTPS calls to the Gemini API work.

.dockerignore

Create a `.dockerignore` file at the project root to prevent unnecessary files from being included in the build context:

```
.dart_tool/
build/
.git/
*.db
.env
```

Setting Up Google Cloud

Before using Secret Manager or deploying, you need the Google Cloud CLI installed and configured. If you have not done this before, follow these steps:

1. Install the gcloud CLI

```
# macOS (via Homebrew)
brew install --cask google-cloud-sdk

# Or download the installer for your OS from:
# https://cloud.google.com/sdk/docs/install
```

2. Log in and set your project

```
# Log in with your Google account
gcloud auth login

# Create a new project (or use an existing one)
gcloud projects create YOUR_PROJECT_ID --name="Lumina"

# Set it as the active project
gcloud config set project YOUR_PROJECT_ID
```

3. Enable the required APIs

```
gcloud services enable run.googleapis.com \
  secretmanager.googleapis.com \
  cloudbuild.googleapis.com
```

Cloud Run, Secret Manager, and Cloud Build all need to be enabled before you can deploy. This is a one-time setup per project.

Secret Manager

Secret Manager is Google Cloud's service for storing sensitive values (API keys, passwords, credentials) outside your codebase. Cloud Run injects them as environment variables at startup, so your Dart code reads them exactly the same way it reads local environment variables with `Platform.environment`.

```
# Create secrets
echo -n "your-gemini-api-key" | \
  gcloud secrets create GEMINI_API_KEY --data-file=- \
  --replication-policy=automatic

echo -n "lumina-secret-2026" | \
  gcloud secrets create LUMINA_API_KEY --data-file=- \
  --replication-policy=automatic

# Grant Cloud Run access to read secrets
gcloud projects add-iam-policy-binding YOUR_PROJECT_ID \
  --member="serviceAccount:YOUR_PROJECT_NUMBER-compute@developer.gserviceaccount.com" \
  --role="roles/secretmanager.secretAccessor"
```

Deploy

With the secrets created and the gcloud CLI configured, run the deploy command from the root of your backend project:

```
gcloud run deploy lumina-backend \
  --source . \
  --region us-central1 \
  --allow-unauthenticated \
  --set-secrets GEMINI_API_KEY=GEMINI_API_KEY:latest \
  --set-secrets LUMINA_API_KEY=LUMINA_API_KEY:latest
```

After deployment, `gcloud` prints the service URL. Copy it.

Update Flutter and Smoke Test

```
// lib/services/lumina_service.dart
// Replace:
static const String _base = 'http://localhost:3400';
// With:
static const String _base = 'https://lumina-backend-xxxx-uc.a.run.app';
```

Remove the iOS `NSAllowsLocalNetworking` exception from `Info.plist` — the production URL is HTTPS and does not need it.

Smoke test the live endpoint:

```
curl -X POST https://lumina-backend-xxxx-uc.a.run.app/analyzeMood \
  -H "Content-Type: application/json" \
  -H "x-api-key: lumina-secret-2026" \
  -d '{"data": {"entryText": "Today felt alive and focused."}}'
```

A 200 response confirms the deployment is successful, the secrets are injected, and the Dotprompt files are being read from the container filesystem.

What You Built

Two typed Genkit flows. A Drift SQLite database. Tool calling that gives the AI access to real journal history. Dotprompt-managed prompts versioned in git. A Shelf server with authentication, logging, and context propagation middleware. A Flutter client calling both flows over HTTP. A live Cloud Run deployment with secrets managed by Secret Manager.

All in Dart. One language from the Flutter UI to the AI infrastructure.

What to Build Next

Lumina works. Now make it yours. Here are four directions to take it; each one a natural next step that uses everything already built.

---

weeklyInsightsFlow

`LEVEL: BEGINNER — same pattern as analyzeMoodFlow`

Read the past 7 days of entries from Drift, pass them to Gemini, and ask for a weekly theme summary. The flow pattern is identical to what you have already built: `defineFlow` → Drift query tool → Gemini → typed output. The only new thing is the prompt.

---

getMoodTrend tool

`LEVEL: BEGINNER — same pattern as getRecentEntries`

Query 30 days of mood scores from Drift and give `journalCoachFlow` a second tool. The coaching question becomes aware of the user's emotional pattern over time; not just the current entry. Two tools, one flow, significantly smarter output.

Firebase Auth JWT

`LEVEL: INTERMEDIATE — replaces the static API key`

Replace the hardcoded `x-api-key` check in `authMiddleware` with Firebase ID token verification. The middleware structure does not change; only what you check inside it. Every user gets their own identity, and the `userId` in context becomes a verified claim rather than a trusted header.

Cloud SQL or Supabase

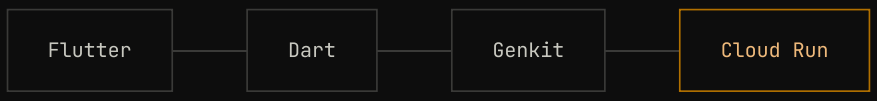
`LEVEL: INTERMEDIATE — swap the database layer`

Replace local SQLite with a managed production database. Only `_openConnection()` in `database.dart` changes. The DAO, the tool, and the flows see no difference. Lumina becomes a multi-user application that persists data beyond a single container instance.

MOBTEREST STUDIO

Follow Mobterest Studio for more.

- GETTING STARTED WITH GENKIT VIDEO
- GITHUB REPO SOURCE CODE
- YOUTUBE CHANNEL
- OFFICIAL WEBSITE



Mobterest Studio

https://www.mobterest.studio/