FLUTTERFEVER

ENGINEERING GUIDE

DART x AI

DART FOR AI

Build AI-powered apps with Dart, clearly and safely.

HTTP + JSON | async / await | Streaming | Structured Output | Function Calling | RAG

Practical patterns for real Flutter AI apps - not just prompt demos.

A focused guide for developers moving from Dart fundamentals to AI engineering.

flutterfever.com

START HERE

Dart for AI: What You Are Actually Learning

Dart is not the language used to train frontier AI models - and that is completely fine. For Flutter developers, Dart is the language that turns AI capabilities into real products: it collects user input, sends requests, validates structured data, streams partial answers, calls tools, updates state, and connects the model to the rest of your application.

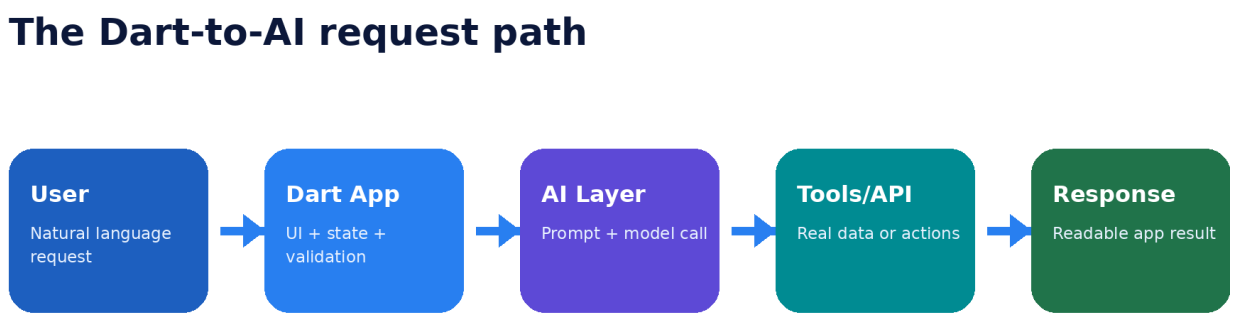

Figure 1. Dart is the orchestration layer between your UI, models, and real application capabilities.

What this guide focuses on

Building AI-powered Flutter and Dart applications: prompts, HTTP, JSON, streaming, structured results, tools, RAG, safety, testing, and product architecture.

What it does not pretend

Dart is not a replacement for Python-based model training stacks. The goal here is production AI integration and application engineering.

Learning path

1. Master the Dart primitives that AI apps use constantly.
2. Build a safe request/response layer.
3. Stream responses without freezing the UI.
4. Convert model output into typed data.
5. Let models request controlled functions and app actions.
6. Add retrieval, memory, reliability, and tests.

> **The mental model**
>
> AI is another asynchronous service. Treat it like a powerful, probabilistic API - not like a magical replacement for your app architecture.

CORE LANGUAGE

1. The Dart Skills AI Apps Use Most

You do not need every corner of the Dart language before building AI features. A small set of concepts appears repeatedly in real AI integrations.

High-frequency Dart

String and interpolation
Map<String, dynamic>
List<T>
nullable types
classes and constructors
Future<T>
Stream<T>

High-frequency AI work

Build JSON payloads
parse responses
track conversation history
stream tokens
represent tool calls
model API errors
update UI state

A message is just typed data

DART MODEL

```
class ChatMessage {
  final String role;
  final String content;

  const ChatMessage({
    required this.role,
    required this.content,
  });

  Map<String, dynamic> toJson() => {
    'role': role,
    'content': content,
  };
}
```

This tiny class already gives you a safer boundary than passing loose maps throughout the entire app. As your product grows, types become one of the strongest defenses against malformed AI data.

Collections are the language of model APIs

PAYLOAD

```
final messages = <ChatMessage>[
  ChatMessage(role: 'system', content: 'Be concise.'),
  ChatMessage(role: 'user', content: 'Explain Futures.'),
];

final payload = {
  'messages': messages.map((m) => m.toJson()).toList(),
};
```

> **Beginner rule**
> When you can confidently work with String, Map, List, classes, null safety, Future, and Stream, you already have the core Dart vocabulary required for most AI app integrations.

NETWORKING

2. HTTP + JSON: The First Real AI Building Block

Most AI features start as an HTTP request. Your Dart code builds a JSON payload, sends it to a server, waits for a response, and decodes the JSON. The model provider may sit behind your own backend, which is usually the safer production design.

A provider-independent client

HTTP REQUEST

```
import 'dart:convert';
import 'package:http/http.dart' as http;

Future<String> askAssistant(String prompt) async {
  final response = await http.post(
    Uri.parse('https://api.example.com/ai/chat'),
    headers: {'Content-Type': 'application/json'},
    body: jsonEncode({'prompt': prompt}),
  );

  if (response.statusCode != 200) {
    throw Exception('AI request failed: ${response.statusCode}');
  }

  final data = jsonDecode(response.body) as Map<String, dynamic>;
  return data['text'] as String;
}
```

The Flutter app does not need to know which model the backend uses. It only needs a stable contract: send prompt, receive a typed result. That makes provider changes much easier later.

The request lifecycle

7. Create a URI and payload.
8. Encode Dart objects as JSON.

9. Send an asynchronous HTTP request.
10. Check status codes before trusting the body.
11. Decode JSON into a predictable Dart shape.
12. Convert the shape into domain models used by your UI.

> **Production boundary**
> Do not embed unrestricted server API secrets in a mobile binary and assume they are hidden. A backend can protect credentials, enforce user permissions, limit abuse, and normalize provider responses.

DATA MODELING

3. Typed Responses Beat Loose JSON

AI APIs return JSON, but your widgets should not spend their lives reading dynamic maps. Parse external data at the edge and expose typed Dart objects to the rest of the app.

TYPED RESULT

```
class AiReply {
  final String text;
  final String requestId;
  final int? inputTokens;
  final int? outputTokens;

  const AiReply({
    required this.text,
    required this.requestId,
    this.inputTokens,
    this.outputTokens,
  });

  factory AiReply.fromJson(Map<String, dynamic> json) {
    return AiReply(
      text: json['text'] as String? ?? '',
      requestId: json['request_id'] as String? ?? 'unknown',
      inputTokens: json['input_tokens'] as int?,
      outputTokens: json['output_tokens'] as int?,
    );
  }
}
```

Why this matters more with AI

- AI features often evolve faster than ordinary endpoints. Typed adapters isolate those changes.
- Nullable values are normal: usage metadata, citations, tool calls, and safety fields may be optional.
- A model can return syntactically valid JSON that is semantically wrong. Parsing is not the same as validation.
- UI code becomes simpler when it consumes AiReply instead of nested provider-specific maps.

Separate three layers

Transport data

Provider JSON and HTTP details. Changes when an API changes.

Domain data

ChatMessage, AiReply, ToolCall, SearchResult. Stable objects your application understands.

> **Good architecture**
> Provider JSON -> adapter -> domain model -> controller/state -> widget. Keep provider-specific keys out of your presentation layer.

ASYNCHRONY

4. async / await: AI Is Mostly Waiting Correctly

Model calls, file uploads, database reads, tool execution, and retrieval are all asynchronous. Dart Futures let your app wait for work without blocking the UI thread.

FUTURE

```
Future<AiReply> sendPrompt(String prompt) async {
  try {
    final reply = await aiRepository.ask(prompt);
    return reply;
  } on TimeoutException {
    throw AiException('The request timed out.');
  } catch (error) {
    throw AiException('Unable to get an AI response.');
  }
}
```

What await actually means

await pauses the current async function until the Future completes. It does not freeze the entire application. Flutter can continue drawing frames and processing other events while the network operation is pending.

Do this	Avoid this
Use async/await for readable sequencing. Catch errors near a layer that can translate them into meaningful app states.	Nested .then() chains everywhere, silent catch blocks, or starting multiple model calls accidentally from rebuilds.

Model UI states explicitly

STATE

```
enum AiStatus { idle, sending, streaming, completed, failed }

class AiState {
  final AiStatus status;
  final String text;
  final String? error;
  // ...
}
```

Flutter connection

The most common AI UX bugs are not “AI problems”; they are async-state problems: duplicate sends, stale responses, missing cancellation, or loading indicators that never reset.

REAL-TIME UX

5. Streaming Responses with Dart Streams

A chat feels dramatically faster when the UI displays partial output as it arrives. Dart Stream is a natural fit for token or text-chunk delivery.

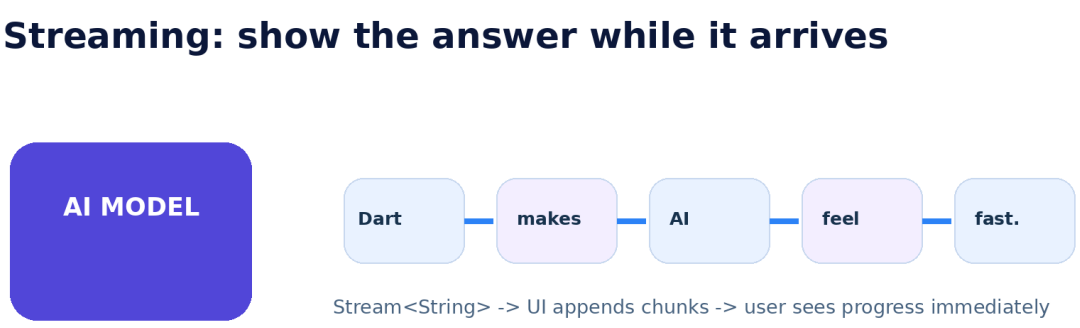

Figure 2. Streaming lets the UI render useful text before the full response is complete.

STREAMING PATTERN

```
Stream<String> streamReply(String prompt) async* {
  await for (final chunk in aiRepository.stream(prompt)) {
    yield chunk.text;
  }
}

Future<void> send(String prompt) async {
  buffer = '';
  await for (final chunk in streamReply(prompt)) {
    buffer += chunk;
    notifyListeners();
  }
}
```

A production stream needs more than text

Real systems often stream events such as response.started, text.delta, tool.requested, tool.result, response.completed, and error. Representing these as a sealed hierarchy or event model is cleaner than pretending every event is a String.

> **UI tip**
> Do not rebuild your entire chat list for every tiny chunk if it causes jank. Update only the active assistant bubble, batch rapid deltas when useful, and preserve scroll position deliberately.

APP DESIGN

6. Build a Clean AI Architecture

The best AI codebase does not begin with a giant ChatScreen. Split responsibilities so each layer can be tested and changed independently.

ARCHITECTURE

```
Flutter UI
   ↓
AiController / StateNotifier / Bloc
   ↓
AiRepository
   ↓
AiProvider interface
   ↓
Backend or model SDK

Optional branches:
   ├── ToolRouter
   ├── RetrievalService
   ├── Local storage
   └── Analytics / observability
```

A small provider abstraction

INTERFACE

```
abstract interface class AiProvider {
  Future<AiReply> complete(List<ChatMessage> messages);
  Stream<AiEvent> stream(List<ChatMessage> messages);
}
```

Your repository can depend on AiProvider rather than OpenAI, Gemini, or another vendor class. A provider adapter translates the stable Dart domain model into whichever external API you use.

Why abstraction helps

Swap models, add fallback providers, mock tests, or compare latency without rewriting UI code.

When not to overdo it

A prototype can start smaller. Add abstractions when they remove real coupling, not because every file must contain an interface.

PROMPTING

7. Prompt Design in Dart

Prompt engineering becomes more reliable when prompts are treated as structured application assets rather than long strings scattered through widgets.

PROMPT ASSET

```
class Prompts {
  static const supportSystem = '''
You are a support assistant.
- Answer only from supplied order data.
- Never invent delivery status.
- Ask for clarification when the request is ambiguous.
''';
}

final messages = [
  ChatMessage(role: 'system', content: Prompts.supportSystem),
  ChatMessage(role: 'user', content: userText),
];
```

Four useful prompt ingredients

13. Role - what job should the assistant perform?
14. Context - what data is allowed to influence the answer?
15. Constraints - what must it avoid or always do?
16. Output contract - plain text, JSON, tool call, short answer, or another known shape?

> **Do not confuse prompts with security**
>
> A system prompt can guide model behavior, but it cannot replace authentication, authorization, database permissions, or validation. Security controls belong in code and infrastructure.

Version prompts like code

For production features, keep important prompts centralized, review changes, test representative conversations, and log which prompt version produced a response. This makes regressions much easier to investigate.

JSON OUTPUT

8. Structured Output: Make AI Return Data Your App Can Use

Sometimes you do not want prose. You want a predictable object that your Dart code can validate and turn into UI.

DESIRED JSON

```
{
  "intent": "track_order",
  "orderId": "A1042",
  "confidence": 0.94
}
```

VALIDATION

```
class IntentResult {
  final String intent;
  final String? orderId;
  final double confidence;

  factory IntentResult.fromJson(Map<String, dynamic> json) {
    final confidence = (json['confidence'] as num).toDouble();
    if (confidence < 0 || confidence > 1) {
      throw FormatException('Invalid confidence');
    }
    return IntentResult(
      intent: json['intent'] as String,
      orderId: json['orderId'] as String?,
      confidence: confidence,
    );
  }
}
```

Structured output is useful for

- Intent classification
- Form autofill
- Extracting dates, names, IDs, and filters
- Generating UI configuration
- Routing a request to the correct workflow
- Returning machine-readable summaries

> **Important distinction**
> Valid JSON is only syntax. Your Dart code still validates required fields, ranges, allowed values, ownership, and business rules.

TOOLS

9. Function Calling: Let AI Request App Actions

Function calling turns a model from a text generator into a controlled decision layer. The model chooses from functions you describe; your Dart or backend code decides whether to execute them.

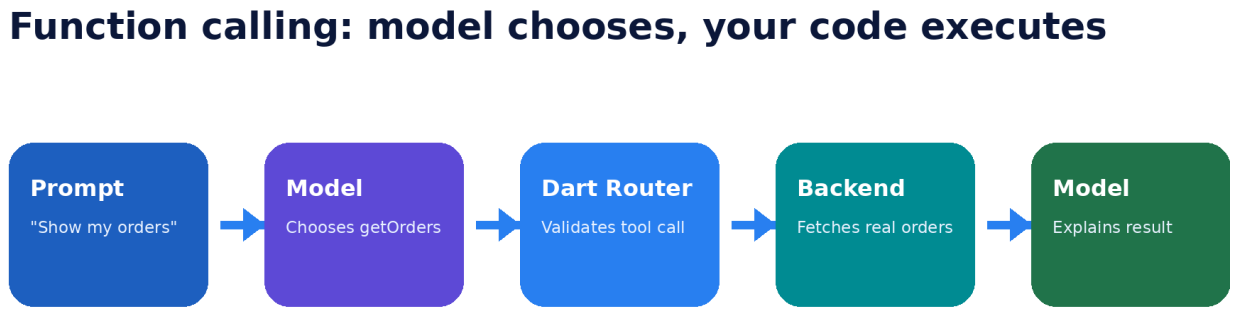

Figure 3. The model proposes the tool; your application validates and executes it.

DOMAIN MODEL

```
class ToolCall {
  final String name;
  final Map<String, dynamic> arguments;

  const ToolCall({required this.name, required this.arguments});
}
```

Example request

User: “Show my orders from yesterday.” The model may request getOrders(date: "2026-08-18"). Your application verifies the tool, date format, authenticated user, and permissions before making the real API request.

> **Core rule**
> The model never becomes the authority. It proposes an action. Your code owns validation, permission checks, confirmation, execution, and audit logs.

TOOL EXECUTION

10. Build a Tool Router in Dart

ROUTER

```
class AiToolRouter {
  final OrderRepository orders;

  AiToolRouter(this.orders);

  Future<Map<String, dynamic>> execute(ToolCall call) async {
    switch (call.name) {
      case 'get_orders':
        final date = call.arguments['date'] as String?;
        if (date == null || !isIsoDate(date)) {
          return {'ok': false, 'error': 'INVALID_DATE'};
        }
        final result = await orders.forDate(date);
        return {'ok': true, 'orders': result.map((e) => e.toJson()).toList()};

      default:
        return {'ok': false, 'error': 'UNKNOWN_TOOL'};
    }
  }
}
```

Read tools and write tools are different

Read tools	Write tools
get_orders	cancel_order
search_products	place_order
get_profile	update_address
check_inventory	delete_account
get_weather	submit_payment

Write tools should usually have stronger controls. A destructive or financial action may require a separate confirmation screen even when the model is highly confident.

Prevent runaway loops

GUARDRAIL

```
const maxToolSteps = 5;
var steps = 0;
while (response.hasToolCall && steps < maxToolSteps) {
  steps++;
  // validate -> execute -> send result back to model
}
```

Narrow tools win

get_orders is easier to test and authorize than execute_any_api. Specific tools reduce ambiguity and attack surface.

SECURITY

11. Keep API Keys and Sensitive Actions Off the Client

A compiled Flutter application is distributed to users. Secrets embedded in constants, assets, or environment files can be extracted. For sensitive production AI, place unrestricted provider credentials behind infrastructure you control.

SECURITY BOUNDARY

```
Flutter app
   ↓ authenticated request
Your backend
   ├── verify user
   ├── enforce quotas
   ├── choose model
   ├── hold provider secret
   ├── validate tools
   └── audit important actions
        ↓
AI provider / business APIs
```

Client-side AI can still be valid

Some ecosystems provide client SDKs designed around app-level protections, attestation, and controlled access. Use the security model documented by that platform rather than copying a server secret into Dart source code.

Checklist before any tool executes

- Is the user authenticated?
- Is this tool allowed for the user role?
- Are all arguments valid and within expected ranges?
- Does the referenced resource belong to this user?
- Does the action need explicit confirmation?
- Is the request rate-limited and auditable?

> **Prompt injection is not just a prompt problem**
> Treat model-produced tool arguments as untrusted input. Validate them with the same seriousness as form input or an external API request.

CONTEXT

12. Conversation Memory Without Losing Control

A chat can remember earlier turns by sending relevant history again, but endless history becomes expensive, slow, and noisy. Memory is a data-management problem.

Three useful memory layers

17. Short-term turns - the recent conversation needed to interpret pronouns and follow-ups.
18. Summarized memory - compressed conversation state such as goals, constraints, or decisions.
19. External memory - durable facts stored in your database and retrieved only when needed.

SIMPLE WINDOW

```
List<ChatMessage> recentWindow(
  List<ChatMessage> messages, {
  int maxMessages = 12,
}) {
  if (messages.length <= maxMessages) return messages;
  return messages.sublist(messages.length - maxMessages);
}
```

A fixed message count is only a beginner approximation. Real limits are usually based on tokens, model context size, content type, and product requirements.

Keep

Current question, relevant prior turns, required instructions, retrieved facts.

Drop or summarize

Old greetings, repeated answers, verbose logs, unrelated branches, data that can be fetched again.

> **Privacy matters**
> Do not store every conversation forever by default. Decide what your product actually needs, disclose it appropriately, and minimize sensitive data.

BEYOND TEXT

13. Multimodal AI: Images, Audio, and Files

Modern models can accept more than text. In Dart and Flutter, the hard part is often not the model call - it is file selection, size limits, MIME types, uploads, lifecycle, permissions, and user feedback.

A safe file pipeline

```
PIPELINE
User selects file
   ↓
Validate type + size
   ↓
Optionally resize/compress
   ↓
Upload or encode using provider-supported method
   ↓
Attach a clear text instruction
   ↓
Model response
   ↓
Dispose temporary resources
```

Do not base64 everything by habit

Large base64 payloads increase memory and request size. Prefer the upload mechanism recommended by your provider or backend, especially for larger images, PDFs, audio, or video.

Example product interactions

- “Explain this screenshot error.”
- “Summarize this PDF invoice.”
- “Extract items from this receipt image.”
- “Describe the plant disease symptoms in this photo.”
- “Transcribe and summarize this voice note.”

> **UX tip**
> Show upload progress and a visible file chip before sending. Users should always know which media is being shared with the model.

RETRIEVAL

14. RAG: Give the Model the Right Knowledge

Retrieval-Augmented Generation (RAG) is useful when answers should come from your own documents, knowledge base, manuals, policies, or frequently changing information rather than the model’s general knowledge alone.

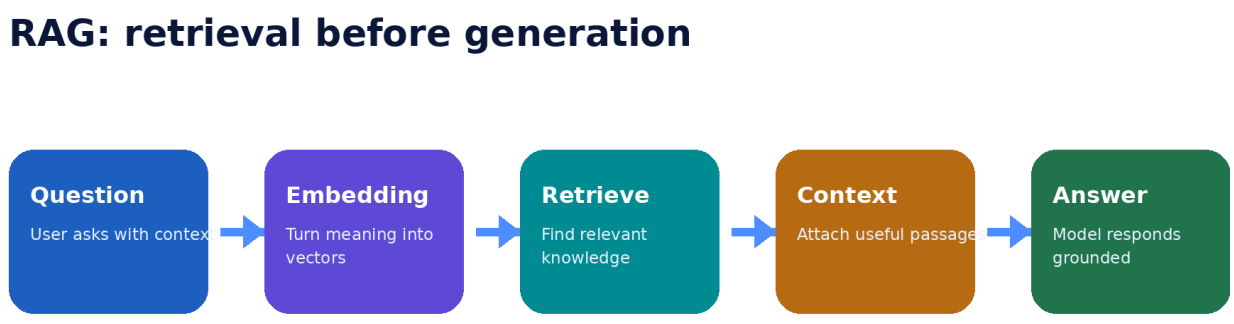

Figure 4. RAG retrieves relevant knowledge before the answer is generated.

What Dart usually does in a RAG app

- Send the user query to your backend.
- Receive retrieved passages and metadata.
- Display citations or source cards when useful.

- Stream the final grounded answer.
- Maintain UI state for search, retrieval, and generation separately.

Embedding generation and vector search often live on the server, but the Flutter/Dart client still orchestrates the experience and presents sources clearly.

> **RAG is not a truth guarantee**
> Retrieval improves grounding only when the source material is relevant and the application communicates it accurately. Validate critical workflows independently.

AGENTS

15. Agentic Workflows: Multiple Steps, Still Controlled

An agentic workflow may call several tools before producing the final response. For example, “Find my latest order and tell me whether I can cancel it” can require more than one operation.

MULTI-STEP FLOW

```
getLatestOrder()
      ↓
getOrderDetails(orderId)
      ↓
checkCancellationEligibility(orderId)
      ↓
Model explains result
      ↓
User confirms if cancellation is requested
```

What makes a workflow agent-like?

- The model selects among tools.
- Multiple tool results may influence the next step.
- The workflow may stop, ask a clarification question, or continue.
- The application enforces maximum steps, permissions, timeouts, and confirmation gates.

Represent steps explicitly

EVENT MODEL

```dart
sealed class AiEvent {}
class TextDelta extends AiEvent { /* ... */ }
class ToolRequested extends AiEvent { /* ... */ }
class ToolCompleted extends AiEvent { /* ... */ }
class AiFailed extends AiEvent { /* ... */ }
class AiFinished extends AiEvent { /* ... */ }
```

> **Do not hide automation**
> When an assistant is about to make a meaningful change, show what it intends to do. Good agent UX builds trust through visibility and control.

RELIABILITY

16. Errors, Retries, Rate Limits, and Cancellation

AI requests fail for ordinary reasons: networks disappear, providers throttle traffic, responses time out, and users leave the screen. Reliability code is part of the feature.

Map technical errors to product states

Technical	User-facing
SocketException	Offline - try again
Timeout	Taking too long
401 / 403	Sign in again
429	Too many requests
5xx	Service unavailable
malformed JSON	Unexpected response

Retry selectively

Retrying a transient 503 may make sense. Retrying a 401 without refreshing credentials does not. Retrying a payment or write tool blindly can be dangerous. Use idempotency and backend rules for operations that must not execute twice.

SIMPLE RETRY

```
Future<T> retryOnce<T>(Future<T> Function() action) async {
  try {
    return await action();
  } on TemporaryAiException {
    await Future<void>.delayed(const Duration(milliseconds: 500));
    return action();
  }
}
```

Cancellation matters

If the user taps Stop, leaves the screen, or starts a new request, cancel or ignore obsolete work so late responses cannot overwrite current UI state.

PERFORMANCE

17. Performance: Keep the AI Feature Feeling Native

Model latency is partly outside your control, but perceived performance is not. Streaming, caching, concurrency discipline, and lightweight parsing can make a large difference.

Practical wins

- Stream long responses instead of waiting for the entire answer.
- Cache static system instructions or reusable app metadata when the platform supports it.
- Avoid uploading the same large file repeatedly.
- Debounce AI-powered search or suggestions.
- Do not trigger network requests from build().
- Keep expensive local parsing away from frame-critical work.

When isolates help

Dart isolates are useful for CPU-heavy local work such as large JSON transformation, text processing, compression, or preprocessing that would otherwise cause UI jank. They do not make remote model latency faster.

ISOLATE

```
// Conceptual: move CPU-heavy preprocessing away from UI work.
final result = await Isolate.run(() {
  return preprocessLargeDocument(rawText);
});
```

Measure before optimizing

Track request latency, first-token latency, total generation time, tool latency, error rate, and cancellation rate. Product performance becomes much easier to improve when it is observable.

TESTING

18. Test AI Features Like Normal Software

Do not make every test depend on a live model. Your UI, tool router, JSON parsing, confirmation rules, and error handling should be deterministic and testable without network calls.

FAKE PROVIDER

```
class FakeAiProvider implements AiProvider {
  @override
  Future<AiReply> complete(List<ChatMessage> messages) async {
    return const AiReply(
      text: 'Mock response',
      requestId: 'test-1',
    );
  }

  @override
  Stream<AiEvent> stream(List<ChatMessage> messages) async* {
    yield TextDelta('Mock ');
    yield TextDelta('stream');
    yield AiFinished();
  }
}
```

Test four separate things

20. Deterministic code - parsing, repositories, routers, state transitions.
21. Tool policy - wrong role, invalid argument, destructive action, confirmation required.
22. Prompt behavior - representative evaluation cases against real models.
23. End-to-end UX - streaming, retry, stop, navigation, and accessibility.

> **Golden rule**
>
> A model can be nondeterministic. Your permission system must not be. Never write security logic that passes only when the model “behaves.”

PRACTICE

19. Mini Project: Build a Dart AI Assistant

Build one small project that exercises the full architecture. The goal is not a flashy chatbot; it is a clean system you understand.

Project: Smart Order Assistant

The user can ask about orders in natural language. The assistant can answer general questions, call get_orders and get_order_details, stream its response, and ask for confirmation before any cancellation action.

Milestone 1 - chat foundation

- Create ChatMessage and AiReply models.
- Build an AiProvider interface and a fake implementation.
- Create chat state with idle, sending, streaming, failed.
- Render messages and a Stop/Retry experience.

Milestone 2 - real AI

- Connect to your backend AI endpoint.
- Add prompt versioning and structured errors.
- Stream partial text into the current assistant bubble.

Milestone 3 - tools

- Define get_orders and get_order_details.
- Create AiToolRouter.
- Validate arguments and resource ownership on the backend.
- Add a cancellation tool only after a confirmation UI exists.

Milestone 4 - production polish

- Log request IDs and latencies.
- Add rate-limit UX.
- Test offline and timeout states.
- Create fake tool results for widget tests.

Success criterion

You should be able to explain exactly which layer owns the prompt, network request, tool permission, business data, state update, and UI rendering.

REFERENCE

20. Dart for AI Cheat Sheet

Core Dart

Future<T> = one async result
Stream<T> = many async events
Map<String,dynamic> = decoded JSON
jsonEncode/jsonDecode = JSON bridge
sealed class = model event families
Isolate = CPU concurrency

AI architecture

AiProvider = vendor adapter
AiRepository = app-level AI access
ToolRouter = controlled execution
ChatMessage = conversation item
AiEvent = streamed lifecycle
RAG = retrieve then generate

Use this decision guide

DECISION MAP

```
Need one remote result?         -> Future<T>
Need incremental output?        -> Stream<T>
Need machine-readable output?   -> JSON schema + validation
Need real app action?           -> Tool/function calling
Need private knowledge?         -> Retrieval / RAG
Need multiple controlled steps?-> Agentic workflow
Need CPU-heavy local work?       -> Isolate
Need provider flexibility?      -> AiProvider abstraction
```

Production checklist

- Typed domain models instead of provider JSON in widgets
- Backend or documented client-security model for credentials
- Timeouts, cancellation, retry policy, and rate-limit UX
- Structured tool results and maximum tool steps
- Explicit confirmation for destructive actions
- Logging with request IDs but minimal sensitive data
- Tests with fake providers plus real-model evaluations
- Useful loading and streaming states

The objective

You do not need Dart to train the model. You need Dart to make the model useful, safe, responsive, and understandable inside a real Flutter product.

REFERENCES

21. Keep Learning from Primary Documentation

AI SDKs and provider APIs evolve quickly. Build your architecture around stable Dart concepts, then check current provider documentation before copying package-specific code.

Dart asynchronous programming

Futures, async, and await.
https://dart.dev/language/async

Dart streams

Creating and consuming asynchronous streams.
https://dart.dev/libraries/async/creating-streams

Dart HTTP and JSON
HTTP requests and converting JSON into class-based structures.
https://dart.dev/server/fetch-data

Dart concurrency
Isolates, event loops, and concurrency.
https://dart.dev/language/concurrency

Gemini function calling
Connecting models to tools and external APIs.
https://ai.google.dev/gemini-api/docs/function-calling

Gemini structured output
Generating structured JSON output.
https://ai.google.dev/gemini-api/docs/structured-output

Firebase AI Logic for Flutter
Google-supported AI integration for mobile/web apps, including Flutter.
https://firebase.google.com/docs/ai-logic

OpenAI developer platform
Current OpenAI API documentation, tools, and model capabilities.
https://platform.openai.com/docs

Version-aware engineering
Treat provider-specific snippets as replaceable edges. Your core Dart models, validation, state management, repositories, and tool policies should remain useful even when an SDK changes.

BUILD WITH CLARITY.
SHIP WITH CONFIDENCE.

Dart gives you the structure. AI gives you new capabilities.
Good engineering connects them safely.

WHAT YOU SHOULD TAKE AWAY

- Use typed Dart models around unpredictable AI responses.
- Treat streaming, retries, errors and cancellation as product states.
- Keep secrets and sensitive tool execution behind trusted services.
- Use function calling and RAG only where they improve the user flow.

flutterfever.com

More practical Flutter engineering guides