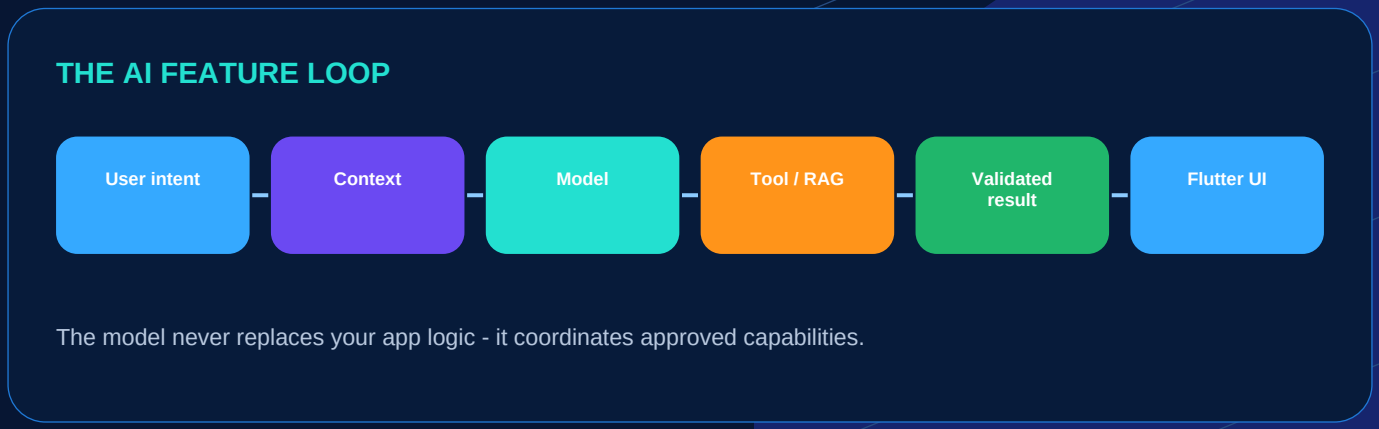
FLUTTERFEVER
DEVELOPER MAGAZINE

BUILD INTELLIGENT APPS. KEEP THE ARCHITECTURE CLEAN.

FLUTTER WITH AI

A practical deep guide to prompts, streaming, tools, RAG, multimodal UX, security, testing, and production AI apps.



Prompts · Streaming · Tools · RAG · Multimodal · Safety

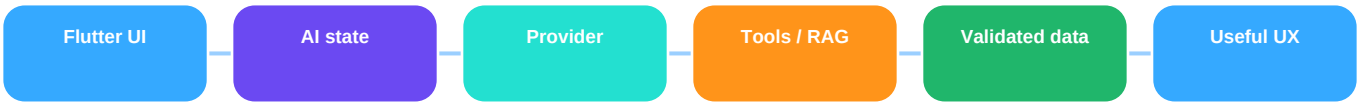
SCAN TO LEARN MORE

flutterfever.com

Flutter with AI - Deep Guide

A production-focused learning path for Flutter developers who want to build useful AI features without turning the app into a pile of provider-specific code.

CORE ARCHITECTURE



HOW TO USE THIS GUIDE

Read chapter by chapter. Each major topic continues for as many pages as needed. The goal is depth, not one-topic-per-page formatting. Code examples are intentionally small enough to adapt to Riverpod, Bloc, GetX, Provider or your own controller pattern.

Contents

01. Design the AI feature before choosing a model

The strongest Flutter AI features start with a user workflow, not with a provider SDK. Decide what the user is trying to accomplish, what data the feature needs, and what the app must validate before the model gets involved.

Start with the user job

A useful AI feature removes steps from a workflow. Instead of adding a generic chat screen, attach AI to a real user problem: explain an error, summarize a report, search private documents, turn speech into a form, or help the user operate an existing screen. This immediately clarifies input, output, permissions and failure states.

- Define one sentence: “The user wants to ___ without manually ___.”
- Identify the minimum context required to solve that job.
- Decide whether the model only generates content or also requests app actions.
- Define what happens when the AI is wrong, slow or unavailable.

DESIGN PRINCIPLE

AI should be an interaction layer over existing app capabilities. The underlying business rules should still work without the model.

Classify the feature by risk

Different AI features need different guardrails. A content explanation can usually be retried safely. A tool that cancels an order or changes a payment method cannot. Risk classification should influence whether the model may act automatically, whether the app asks for confirmation, and where execution occurs.

RISK MODEL

LOW RISK	MEDIUM RISK	HIGH RISK
Summarize, explain, classify, rewrite. Usually read-only and easy to retry.	Search private data, create drafts, prepare actions. Requires authorization and validation.	Delete, pay, publish, cancel, modify account state. Requires confirmation and server-side controls.

Design the success and failure experience together

AI UX is incomplete if it only designs the successful answer. The Flutter screen needs visible states for sending, waiting, streaming, tool execution, completion, cancellation and failure. When you know these states early, state management becomes simpler and the interface feels intentional.

MODEL EXPLICIT UI STATE

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

class AiState {
  final AiStatus status;
  final String text;
  final String? error;
  const AiState(this.status, this.text, this.error);
}
```

Decide what the model is allowed to know

Context should be minimal and purposeful. A model answering a question about one invoice does not need the user's entire account history. Smaller context reduces cost, lowers privacy exposure and usually improves relevance. Build context deliberately rather than serializing entire app models into prompts.

CONTEXT BOUNDARY

Send the smallest trustworthy slice of data that can solve the current task. Retrieve more only when the workflow requires it.

02. Networking, typed messages and provider boundaries

AI features still depend on ordinary app engineering: HTTP, JSON, timeouts, typed models, retries and authentication. The model is only one service in that pipeline.

Do not make widgets provider-aware

If a widget imports provider-specific request classes, the UI becomes tied to that vendor. Instead, expose a small domain interface such as AiProvider or AiGateway. Provider code translates your app's neutral message types into whichever external API is currently used.

PROVIDER BOUNDARY

```
abstract interface class AiProvider {
  Future<AiReply> complete(AiRequest request);
  Stream<AiChunk> stream(AiRequest request);
}

class AiRequest {
  final List<AiMessage> messages;
  final Map<String, Object?> context;
  const AiRequest(this.messages, this.context);
}
```

This abstraction makes testing easier because the controller can receive a fake provider. It also makes migrations manageable if models, pricing or SDKs change later.

Use typed messages instead of raw maps

Raw JSON maps are fine at the network boundary, but they should not spread across the app. Convert external data into typed Dart objects once. The rest of the application should work with predictable fields and enums.

TYPED CONVERSATION MODEL

```
enum AiRole { system, user, assistant, tool }

class AiMessage {
  final AiRole role;
  final String content;

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

  Map<String, Object?> toJson() => {
    'role': role.name,
    'content': content,
  };
}
```

Timeouts and retry policy

AI requests are often slower than ordinary REST calls. Still, the app needs an upper bound. A timeout should cancel the visible loading state, preserve the user message and allow a retry. Automatic retries should be conservative because repeated model calls cost money and may duplicate tool actions.

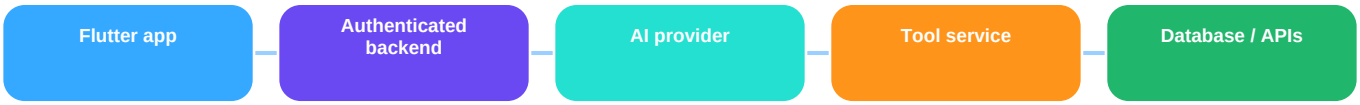
- Retry read-only model calls only when the failure is clearly transient.
- Never automatically replay a destructive tool action.
- Use exponential delay for network retry, but cap attempts.
- Keep a request ID so logs can connect UI, backend and provider events.

IMPORTANT DISTINCTION

Retrying “generate an answer” is different from retrying “execute a tool”. Treat model generation and business actions as separate operations.

Backend as a security and orchestration layer

RECOMMENDED PRODUCTION ROUTE



For production apps, a backend can protect provider secrets, enforce quotas, validate tool calls, attach user permissions, redact sensitive fields and record usage. Flutter remains responsible for responsive UI and local state; the backend owns privileged operations.

03. Prompt design, context and structured output

Prompt engineering in an app is less about clever phrases and more about consistent contracts. Define the task, the trusted context, the constraints and the exact output shape your UI expects.

Separate instructions, context and user input

Do not concatenate everything into one uncontrolled string. Keep system or developer instructions separate from user text. When passing app data, label it as context. This makes prompts easier to reason about and reduces the chance that user-provided text is interpreted as a higher-priority rule.

SEPARATE CONCERNS

```
final request = AiRequest(
  messages: [
    AiMessage(role: AiRole.system, content: systemRules),
    AiMessage(role: AiRole.user, content: userText),
  ],
  context: {
    'selectedError': errorLog,
    'experienceLevel': 'beginner',
  },
);
```

Ask for structured output when UI depends on structure

If the model output becomes cards, filters, actions or form fields, plain prose is fragile. Prefer a schema that your app can validate. The model may still produce invalid data, so parsing must be defensive.

VALIDATED OUTPUT MODEL

```
class FixSuggestion {
  final String title;
  final List<String> steps;
  final String? code;

  FixSuggestion.fromJson(Map<String, Object?> json)
      : title = json['title'] as String,
        steps = (json['steps'] as List).cast<String>(),
        code = json['code'] as String?;
}
```

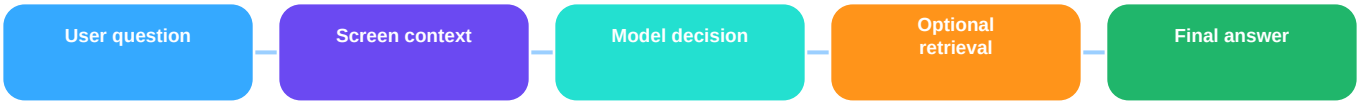
UI CONTRACT

Treat AI output like an external API response. Validate fields, provide defaults and show a fallback when parsing fails.

Build context progressively

Large prompts are expensive and can reduce signal. Start with the current screen context, then retrieve additional data only if needed. For example, a support assistant may first receive the visible error and app version; only if the model needs documentation should the workflow query the knowledge base.

PROGRESSIVE CONTEXT



Conversation memory is a product decision

A conversation list is not automatically useful memory. Decide what should persist: recent messages, a compact summary, selected preferences or task state. Sensitive data should have a clear retention policy. Long conversations can be summarized periodically rather than sending the full transcript on every request.

Prompt injection awareness

If the model receives untrusted text from websites, documents or user files, that content can contain instructions. Your app should treat retrieved text as data, not authority. Tool permissions and safety rules must be enforced outside the model.

RULE

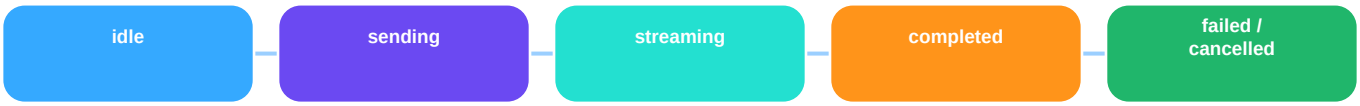
Never rely on the model to protect its own tool permissions. Authorization belongs in code and on the server.

04. Streaming responses and state management

Streaming improves perceived speed because the user sees progress before the full response is ready. It also introduces partial text, cancellation, ordering, duplicate events and lifecycle problems that ordinary request/response screens do not have.

Understand the streaming lifecycle

VISIBLE STATE PROGRESSION



A good controller starts a request, emits a sending state, appends chunks while streaming, and always settles into a terminal state. The terminal transition must happen in finally-like cleanup logic so loading indicators do not remain stuck after exceptions or cancellation.

STREAMING CONTROLLER

```
Future<void> send(String prompt) async {
  state = state.copyWith(status: AiStatus.sending, text: '');
  try {
    await for (final chunk in provider.stream(buildRequest(prompt))) {
      state = state.copyWith(
        status: AiStatus.streaming,
        text: state.text + chunk.text,
      );
    }
    state = state.copyWith(status: AiStatus.completed);
  } catch (e) {
    state = state.copyWith(status: AiStatus.failed, error: '$e');
  }
}
```

Avoid rebuilding the whole chat screen for each token

Streaming can produce many small updates. If the entire page rebuilds on every token, performance may suffer. Keep message widgets stable and update only the actively streaming message. State libraries can help, but the underlying principle is selective observation.

- Give each message a stable ID.
- Store the streaming draft separately from completed messages.
- Observe only the active message text in the streaming bubble.
- Throttle UI updates if the provider emits extremely small chunks.

Cancellation must be real, not cosmetic

A Stop button should cancel the active subscription or request, not merely hide the spinner. The controller also needs to ignore late chunks from a cancelled request. One simple pattern is a monotonically increasing request token; chunks are applied only when they belong to the current request.

IGNORE STALE STREAMS

```
int _requestVersion = 0;

Future<void> send(String prompt) async {
  final version = ++_requestVersion;
  await for (final chunk in provider.stream(buildRequest(prompt))) {
    if (version != _requestVersion) break;
    appendChunk(chunk.text);
  }
}

void cancel() => _requestVersion++;
```

Scrolling behavior matters

Auto-scroll only when the user is already near the bottom. If they scroll up to read older content, do not keep pulling them back down on every chunk. A “jump to latest” button is often better than forced scrolling.

Smooth streaming

Update one bubble, keep message IDs stable, and avoid layout jumps.

Failure recovery

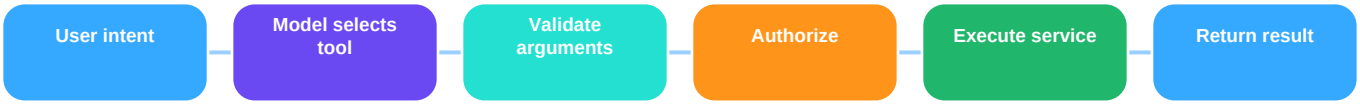
Keep the user prompt visible and expose retry without duplicating the conversation.

05. Function calling and safe tool execution

Function calling lets the model request a named capability such as getOrders, searchProducts or createDraft. The model selects the tool and proposes arguments; your app or backend decides whether the request is valid and executes the real code.

The model requests - your code executes

TOOL EXECUTION PIPELINE



This distinction is the foundation of safe AI actions. A model must not directly hold database credentials or bypass your repositories. Tool definitions describe approved capabilities, while the tool router maps a tool name to normal application services.

CENTRAL TOOL ROUTER

```
class ToolRouter {
  final OrderRepository orders;

  Future<Object?> execute(ToolCall call, UserSession session) async {
    switch (call.name) {
      case 'get_orders':
        final date = requireIsoDate(call.args['date']);
        return orders.getForUser(session.userId, date);
      default:
        throw UnsupportedError('Tool not allowed');
    }
  }
}
```

Design narrow tools

Tools should describe business capabilities, not infrastructure. A tool named executeSql is dangerous and difficult to authorize. A tool named getUserOrders is specific, testable and easier to restrict. Narrow tools also help the model choose correctly.

- Prefer get_order_status over call_any_endpoint.
- Prefer search_catalog over execute_database_query.
- Include clear parameter types and descriptions.
- Return structured results with stable error codes.

Read tools and write tools need different policy

A read tool can often execute after authentication. A write tool changes app state and usually needs a confirmation step. The confirmation should be generated by your Flutter UI, not only by the model's wording.

EXAMPLE

Model requests cancel_order(orderId: 1042). Flutter shows a native confirmation sheet.
Only after explicit confirmation does the backend execute the cancellation.

Multi-tool workflows

Advanced requests may require several steps. "Find my latest order and tell me whether it can still be cancelled" can call getLatestOrder followed by checkCancellationEligibility. Put a maximum tool-call count on each user turn to prevent loops and runaway cost.

BOUNDED TOOL LOOP

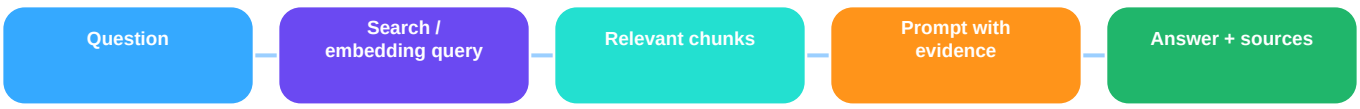
```
const maxToolCalls = 5;
var calls = 0;

while (response.hasToolCall && calls < maxToolCalls) {
  calls++;
  final result = await router.execute(response.toolCall!, session);
  response = await provider.continueWithToolResult(result);
}
```

06. RAG and grounding answers in trusted data

Retrieval-Augmented Generation (RAG) is useful when the model needs information from your own documents, product catalog, policies, manuals or knowledge base. The goal is not to make the model "know" the data permanently; it is to retrieve relevant evidence for the current question.

RAG pipeline in a Flutter product

GROUNDED ANSWER FLOW



Flutter normally triggers the workflow, but retrieval is often implemented on a backend. Documents are split into chunks, indexed, searched and ranked. Only the relevant chunks are attached to the model request. The final answer can include source labels that the UI renders as tappable references.

Chunking affects answer quality

Chunks that are too small lose context; chunks that are too large waste tokens and mix unrelated ideas. A practical strategy is to split by meaningful headings or paragraphs and keep metadata such as document ID, title, section and last-updated time.

- Store source metadata alongside each chunk.

- Prefer semantic sections over arbitrary character counts when possible.
- Retrieve a small candidate set, then rerank or filter.
- Do not send every matched chunk to the model.

Show evidence in the UI

A grounded answer is more useful when the user can inspect where it came from. Flutter can render small source chips under the response. Tapping a chip can open the relevant document section. This is especially valuable for enterprise, support and policy applications.

MODEL GROUNDED REPLIES EXPLICITLY

```
class Citation {
  final String sourceId;
  final String title;
  final String snippet;
}

class GroundedReply {
  final String answer;
  final List<Citation> citations;
}
```

RAG does not replace authorization

Retrieval must filter by the current user's permissions before data reaches the model. Do not retrieve across all tenants and hope the model refuses to reveal a private document. Security must be enforced by the search layer and backend.

> **SECURITY BOUNDARY**
>
> Permission filtering happens before retrieval results are attached to the model request.

When not to use RAG

If the required information already exists in a structured API, use a tool call instead. Orders, account balances and live inventory are better fetched from business services. RAG is strongest for unstructured or semi-structured knowledge such as docs, FAQs, manuals and policies.

07. Multimodal AI: images, voice and files

Flutter AI features can accept more than text. Camera images, screenshots, audio, documents and voice can reduce typing and unlock workflows that feel native on mobile devices.

Image input should start with a clear user goal

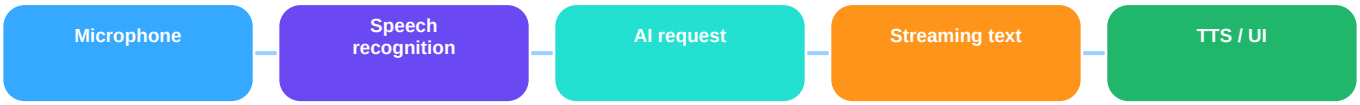
Do not upload an image just because the model accepts images. Define what the app needs: extract text, classify a product, explain a chart, inspect a UI screenshot, summarize a receipt or identify fields for a form. The goal determines image quality, cropping and privacy requirements.

- Resize large images before upload when full resolution is unnecessary.
- Allow the user to review the selected image before sending.
- Strip metadata when it is not needed.
- Explain when an image is uploaded to a remote service.

Voice interaction is two separate systems

A voice assistant usually combines speech-to-text for input and text-to-speech for output. Treat them as separate services. The AI model receives text or audio depending on your architecture, but Flutter must still manage microphone permission, recording state, interruption and playback lifecycle.

VOICE WORKFLOW



File analysis requires preprocessing

Large PDFs or documents should not be loaded blindly into a prompt. Extract or index their content, preserve page or section metadata, and retrieve only relevant sections. For temporary one-file analysis, a backend can parse the file and create a short-lived retrieval index.

MOBILE UX

Always show upload/progress/cancel states for large media. The user should know whether the app is still uploading, the model is thinking, or the request has failed.

Multimodal result design

The output may include text plus structured detections, bounding boxes or extracted fields. Model those results so Flutter can render the right widgets instead of forcing everything into Markdown.

DIFFERENT OUTPUT TYPES

```
sealed class AiResult {}

class TextResult extends AiResult {
  final String text;
}

class ExtractedFields extends AiResult {
  final Map<String, String> fields;
}

class DetectedItems extends AiResult {
  final List<DetectedItem> items;
}
```

08. On-device and hybrid AI architecture

On-device models can reduce latency, work offline and keep some data local, but mobile constraints matter. Model size, RAM, battery, thermal load and device compatibility influence the user experience.

Choose on-device AI for the right tasks

Small local models are useful for classification, lightweight summarization, embeddings, offline assistance and privacy-sensitive preprocessing. They may not match large cloud models for complex reasoning. A hybrid architecture can route simple tasks locally and difficult tasks to the server.

HYBRID ROUTING

LOCAL	ROUTER	CLOUD
Fast, offline, private. Best for small predictable tasks.	Decides based on task, connectivity, model availability and user policy.	Stronger models, larger context and external tools, but adds network and cost.

Do not block the UI isolate

Model inference, tokenization or large preprocessing can be expensive. If the SDK performs work synchronously on the Dart UI isolate, animations and touch input may stutter. Use APIs that run work natively or off the UI isolate, and profile the actual device instead of assuming performance.

Download and lifecycle

If the app downloads model files, make the state explicit: not installed, downloading, ready, update available, corrupted and unsupported. Users also need storage visibility and a way to remove downloaded models.

MODEL LIFECYCLE STATE

```
enum LocalModelStatus {
  notInstalled,
  downloading,
  ready,
  unsupported,
  failed,
}

class LocalModelState {
  final LocalModelStatus status;
  final double progress;
}
```

Privacy messaging

If a task stays on-device, say so clearly. If the app falls back to cloud processing, make that behavior explicit when privacy matters. Hybrid AI is a product decision as much as a technical decision.

09. Security, privacy, cost and abuse controls

AI features introduce new attack surfaces and operating costs. Security cannot be added after the UI is finished; it affects prompt design, tool routing, logging, retention and backend architecture.

Protect secrets and privileged actions

Provider API keys, service credentials and database secrets should not be treated as hidden simply because they are compiled into a mobile app. Privileged model calls and tools belong behind authenticated server APIs whenever possible.

Prompt data may be sensitive

Before sending screen context, ask whether every field is necessary. Redact tokens, passwords, payment details and unrelated personal data. Logging should prefer request IDs and error codes over full prompts. If you must log content for debugging, use an explicit environment and retention policy.

Rate limits and quotas

A single user can accidentally or intentionally generate expensive traffic. Apply per-user and per-device quotas, request size limits, tool-call limits and maximum context length. The UI should explain when a quota is reached instead of repeatedly retrying.

EXPLICIT OPERATING LIMITS

```
class AiBudget {
  final int maxRequestsPerMinute;
  final int maxToolCallsPerTurn;
  final int maxContextCharacters;
  const AiBudget({
    this.maxRequestsPerMinute = 12,
    this.maxToolCallsPerTurn = 5,
    this.maxContextCharacters = 12000,
  });
}
```

Cost-aware UX

Streaming, long history and repeated retries can increase usage. Summarize old conversations, cache safe deterministic results, retrieve only relevant context, and avoid calling a powerful model when a small local rule can solve the task.

- Cache static explanations when appropriate.
- Do not resend full documents on every turn.
- Collapse old chat history into a compact summary.
- Use tool APIs for live structured data instead of asking the model to guess.

FAIL CLOSED

If authorization, validation or policy checks fail, do not execute the tool. Return a structured error and let the UI explain what the user can do next.

10. Testing, observability and production reliability

AI output is variable, but the surrounding app logic is testable. Focus tests on contracts: state transitions, tool permissions, parser behavior, retries, cancellation and UI rendering.

Unit-test the controller without a real model

Inject a fake AiProvider that emits predictable chunks or errors. You can then verify that sending transitions to streaming, chunks append in order, cancellation stops updates and failure clears the loading state.

DETERMINISTIC TEST PROVIDER

```
class FakeAiProvider implements AiProvider {
  @override
  Stream<AiChunk> stream(AiRequest request) async* {
    yield const AiChunk('Hello');
    yield const AiChunk(' world');
  }

  @override
  Future<AiReply> complete(AiRequest request) async =>
      const AiReply('Hello world');
}
```

Test tool authorization separately

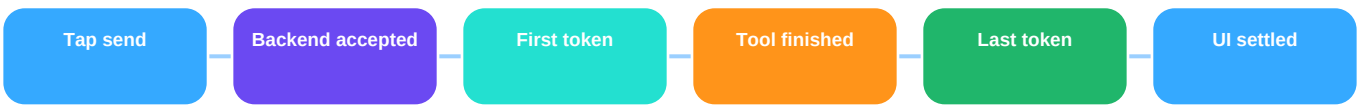
Tool tests should not depend on model behavior. Call the router directly with allowed and disallowed users, malformed arguments and missing resources. This is where you verify that the server never trusts a user ID proposed by the model.

Golden and widget tests for AI UI

The UI has predictable visual states even when the answer is not predictable: empty, sending, streaming, long content, error, citation list, tool confirmation and retry. Widget or golden tests can cover these states with fixed data.

Observe latency in stages

USEFUL TIMING POINTS



A single total duration is not enough. Measure time to first token, model duration, tool latency, retrieval latency and render completion. Slow first token feels different from a fast start followed by a slow tool call, and the fix is different.

Log with request IDs

Assign a request ID before the network call and include it in backend logs. Store provider, model, status, latency and tool names where appropriate. Avoid storing raw sensitive prompts by default.

PRODUCTION HABIT

When a user reports “AI is stuck”, a request ID should let you see whether the delay happened in Flutter, the backend, retrieval, the provider or a tool.

11. End-to-end mini architecture: AI support assistant

This mini architecture combines the concepts into one realistic feature: a support assistant inside a Flutter app that explains errors, searches trusted documentation and can retrieve the user’s account status through approved tools.

Feature requirements

- User can paste an error or select one from a diagnostics screen.
- Assistant streams a concise explanation.
- If documentation is needed, the backend retrieves relevant help content.
- If account-specific status is needed, the model may request an approved read-only tool.
- Sources are shown below the answer.
- The user can stop generation or retry after failure.

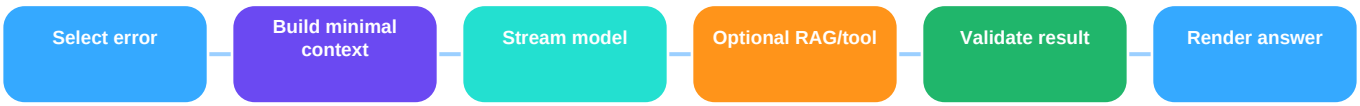
Folder boundaries

FEATURE-FIRST STRUCTURE

```
lib/features/ai_support/
  presentation/
    support_screen.dart
    support_controller.dart
    support_state.dart
  domain/
    ai_message.dart
    support_reply.dart
    citation.dart
  data/
    ai_provider.dart
    support_repository.dart
    tool_router.dart
  widgets/
    streaming_message.dart
    citation_chip.dart
    tool_confirmation.dart
```

Request flow

SUPPORT ASSISTANT FLOW



Controller responsibilities

The controller coordinates state and delegates. It should not parse provider-specific JSON, perform REST calls directly or know database details. Its job is to create the domain request, consume the provider stream, route tool requests through a service and emit UI state.

CONTROLLER ORCHESTRATION

```
Future<void> explain(ErrorContext context) async {
  final requestId = idGenerator.next();
  state = AiSupportState.sending(requestId);

  try {
    final session = repository.start(context);
    await for (final event in session.events) {
      state = reduce(state, event);
    }
  } catch (e) {
    state = state.fail(normalizeError(e));
  }
}
```

Backend responsibilities

- Authenticate the user and enforce quotas.
- Attach system rules and redact sensitive fields.
- Call retrieval only when required.
- Validate every tool call and enforce ownership checks.
- Normalize provider errors into stable app error codes.
- Return streaming events with a request ID.

UI responsibilities

The screen renders state and sends user intent. It shows a stable conversation, an active streaming bubble, citation chips, a stop button and a retry action. Tool confirmations are native UI elements rather than text buried inside the model response.

> **WHY THIS ARCHITECTURE SCALES**
>
> Each layer can change independently: provider, retrieval engine, state library, backend framework or UI design. The feature contract remains stable.

12. Production checklist and quick reference

Use this final section before shipping any Flutter AI feature. It is intentionally practical: if an item is unclear, revisit the chapter that owns that decision.

Architecture checklist

- UI does not construct provider-specific JSON.
- AI provider is behind a small domain interface.
- Tool execution goes through a central allow-listed router.
- Privileged tools and secrets are behind an authenticated backend.
- RAG retrieval applies authorization before returning chunks.
- State explicitly handles streaming, cancellation and failure.

UX checklist

- User understands what the AI feature does and does not do.
- Sending, streaming and tool activity are visible.
- Stop actually cancels work or ignores stale events.
- Long responses do not force-scroll a user who is reading older content.
- Errors preserve the original prompt and provide a safe retry.
- High-risk actions require explicit confirmation.

Data and security checklist

- Only necessary context is sent to the model.
- Logs avoid secrets and unnecessary prompt content.

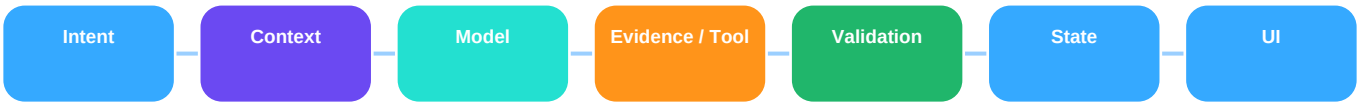
- Rate limits and usage quotas exist.
- Tool arguments are validated in code.
- Ownership and role permissions are checked on the server.
- Conversation retention has an explicit policy.

Performance checklist

- Time to first token is measured.
- Large files and images are preprocessed before upload.
- Old chat history is summarized or trimmed.
- Heavy local inference does not block the UI isolate.
- Streaming updates rebuild the smallest practical widget subtree.

A compact mental model

EVERY AI TURN



FINAL RULE

The model may suggest. Your app decides. Your backend authorizes. Your UI explains. Keeping those responsibilities separate is what turns an AI demo into a production Flutter feature.

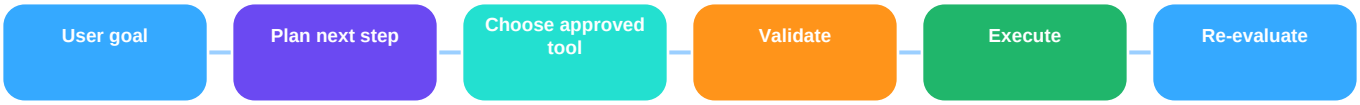
13. Agentic workflows without losing control

An agentic workflow is simply a model loop that can decide what step to take next. In Flutter products, the safest approach is bounded orchestration: a small set of tools, a clear goal, strict limits and visible progress.

Start with a workflow graph, not an open-ended agent

If the task can be described as a known sequence with a few decisions, encode that structure in code. Let the model choose among approved branches, but keep the outer loop deterministic. This makes behavior easier to test and prevents the model from inventing unsupported steps.

BOUNDED AGENT LOOP



Expose progress as events

Longer workflows should emit meaningful events such as searching documents, checking inventory, comparing options or preparing a draft. Flutter can render these as a compact activity timeline. Progress labels should map to real stages in your orchestrator.

AGENT PROGRESS EVENTS

```
sealed class AgentEvent {}
class Thinking extends AgentEvent {}
class ToolStarted extends AgentEvent {
  final String name;
}
class ToolFinished extends AgentEvent {
  final String name;
  final Duration elapsed;
}
class AgentCompleted extends AgentEvent {
  final String answer;
}
```

Set hard limits

- Maximum tool calls per turn.
- Maximum total workflow duration.
- Maximum retrieval operations.
- Maximum output size.
- Explicit cancellation support.

DO NOT CONFUSE AUTONOMY WITH QUALITY

More loops do not automatically make a better AI feature. A short, predictable workflow is usually easier for users to trust and easier for developers to operate.

14. Conversation history, persistence and memory

Users expect continuity, but storing every message forever is rarely the right design. Separate visible chat history from the compact context the model actually needs for the next turn.

Three kinds of memory

MEMORY LAYERS

VISIBLE HISTORY	WORKING CONTEXT	LONG-TERM FACTS
Messages the user can scroll through and manage.	Recent turns and current task state sent to the model.	Explicit preferences or facts stored only when the product needs them.

Visible history is a UI concern. Working context is a model input concern. Long-term facts are a product data concern. Mixing all three leads to huge prompts and unclear privacy behavior.

Summarize old conversation

When a conversation becomes long, generate or compute a compact task summary and keep only recent turns verbatim. The summary should capture decisions, selected entities and unresolved goals rather than every sentence.

COMPACT CONVERSATION CONTEXT

```
class ConversationContext {
  final String summary;
  final List<AiMessage> recentMessages;
  final Map<String, Object?> taskState;
}
```

Persist deliberately

If chats are saved locally, protect sensitive content according to the app threat model. If chats are synchronized to a server, make deletion and retention behavior clear. Memory should never silently become a permanent profile simply because the model mentioned something once.

GOOD DEFAULT

Persist only what the user expects to find again. Derive temporary model context from that data rather than storing every internal prompt.

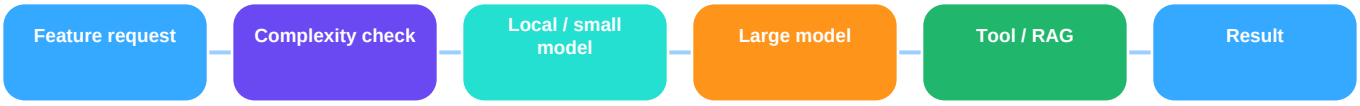
15. Cost, model routing and graceful degradation

Production AI needs a budget strategy. The goal is not always to use the strongest model; it is to use the cheapest reliable path that solves the current task with acceptable latency.

Route by task complexity

Classification, formatting and simple extraction may work with smaller or local models. Complex reasoning, long context or multimodal analysis may require a larger cloud model. A routing layer can select the path based on feature, context size, network state and user tier.

COST-AWARE ROUTING



Use deterministic code before AI

Do not ask a model to do work that ordinary Dart code can perform exactly. Date formatting, arithmetic, validation, permission checks and known navigation rules belong in code. Every deterministic step moved out of the model improves cost, speed and reliability.

Graceful degradation

When the provider is unavailable, the app should still be useful. Disable only the AI-dependent action, preserve typed input, show a clear retry, and keep ordinary app navigation working. For some features, a local fallback or cached result can provide partial value.

Online path

Stream the model answer, retrieve live context and use tools when allowed.

Fallback path

Keep the workflow usable with search, cached help or manual actions.

Budget visibility

Track requests per feature, average input size, output size, tool calls and failure rate. Cost problems are easier to fix when metrics are attributed to a concrete workflow rather than one global AI number.

16. Common Flutter AI mistakes and how to correct them

Most production problems are not caused by the model itself. They come from state, architecture, permissions, context size or UI behavior. These patterns are worth recognizing early.

Mistake: one giant AI service

A single class that builds prompts, performs HTTP, parses JSON, executes tools, stores history and updates UI state becomes impossible to test. Split provider transport, orchestration, tools, repositories and presentation state.

Mistake: generic chat as the entire product

A blank chat screen makes users invent the workflow. Better AI UX is contextual: provide suggested actions, selected-screen context, source chips and normal UI controls around the model.

Mistake: boolean state explosion

USE ONE EXPLICIT STATE MACHINE

```
// Fragile
bool loading;
bool streaming;
bool failed;
bool waitingForTool;

// Better
enum AiStatus { idle, sending, streaming, waitingForTool, completed, failed }
```

Mistake: trusting tool arguments

The model can propose an order ID, date, file path or user ID. Treat every argument as untrusted input. Validate format, enforce ownership and derive identity from the authenticated session.

Mistake: sending too much context

Large context raises cost and can bury the important signal. Retrieve narrowly, summarize history and attach only data that is relevant to the current turn.

Mistake: no cancellation

Mobile users change their minds. A request that cannot be stopped feels broken. Support cancellation or stale-response suppression from the beginning.

Mistake: unbounded conversation history

Sending the entire chat on every request looks simple at first, but it increases cost and eventually makes the prompt noisy. Keep recent turns, summarize older decisions, and preserve only task state that still matters.

Mistake: hiding every provider error behind one message

A generic “Something went wrong” message is safe for users but useless for engineering. Normalize failures into categories such as authentication, rate limit, timeout, malformed response, tool failure and provider unavailable. The UI can stay simple while logs remain actionable.

Mistake: testing only the happy prompt

Real users provide short, vague, multilingual, duplicated and contradictory input. Test empty text, very long text, rapid repeated sends, cancellation during streaming, malformed structured output and tool calls with invalid arguments.

REFACTORING SIGNAL

If adding a second AI feature requires copying the first feature’s provider code, prompt builder and tool handling, create shared domain contracts before adding more screens.

RELEASE CHECK

Before shipping, run one manual test with slow network, one with the provider unavailable, one with a long streaming answer, and one with a rejected tool action. These four tests reveal many production-only UX problems.

FLUTTERFEVER
DEVELOPER MAGAZINE

AI should make the app simpler, not the architecture messier.

Use this guide as a production checklist: model state explicitly, stream safely, validate tool calls, ground answers in trusted data, and keep secrets behind a backend boundary. Build one useful workflow first - then scale.

PRODUCTION READINESS CHECK

- The model has a narrow, clearly defined job.
- User state covers idle, sending, streaming, success, failure and cancellation.
- Tool calls are allow-listed, validated and authorized.
- Private data is retrieved through trusted services, not pasted blindly into prompts.
- Costs, latency, errors and request IDs are observable.
- The UI still works when the AI provider is slow or unavailable.

FLUTTERFEVER
flutterfever.com
Practical Flutter engineering for production apps.