# On-Device Intelligence — Chapter 3 Sample

Digital Foundry

# Contents

# Chapter 3: Your First On-Device Feature

*Draft v0.2 · Baseline: iOS 26 / Xcode 26 · Verified against SDK 26.5*

You have a notes app. Users paste in raw meeting notes and want a digest: a one-line summary, the action items, who owns them. In the cloud era this is a `POST` to someone's API, a per-request invoice, and a privacy-policy update. On iOS 26 it's a framework import. By the end of this chapter the feature works end to end: offline, at zero marginal cost, with the model output arriving as a *typed Swift value*, not a JSON string you pray over.

We build it in five moves: check availability, define the output type, run the request, stream it into SwiftUI, and handle the errors that will absolutely occur in production.

## 1. Availability is a product decision, not a guard clause

The system model only exists on Apple Intelligence–capable hardware with Apple Intelligence turned on. That's a meaningful slice of your user base excluded: iPhone 15 non-Pro and earlier, plus everyone who toggled the feature off. Before writing any AI code, decide what those users see. "Nothing" is a valid answer; a silently broken button is not.

```
import FoundationModels

struct DigestAvailability {
    static func check() -> String? {
        let model = SystemLanguageModel.default
        switch model.availability {
        case .available:
            return nil // good to go
        case .unavailable(.deviceNotEligible):
            return "Digest requires a device that supports Apple Intelligence."
        case .unavailable(.appleIntelligenceNotEnabled):
            return "Turn on Apple Intelligence in Settings to use Digest."
        case .unavailable(.modelNotReady):
            return "The on-device model is still downloading. Try again shortly."
        case .unavailable(_):
            return "Digest isn't available on this device."
        }
```

```
    }
}
```

Two production notes the docs won't stress:

- `.modelNotReady` is *transient*. It shows up right after OS updates and on fresh devices while assets download. Treat it as "retry later," not "unsupported," and re-check on `scenePhase` changes.
- The enum has non-frozen semantics; always keep the wildcard `unavailable(_)` arm so a new reason in iOS 27 degrades gracefully instead of trapping in a `switch` you wrote a year ago.

## 2. Define the output as a type, not a prompt

The single best thing about this framework is *guided generation*: you hand the model a Swift type and constrained decoding guarantees the response conforms. There is no "sometimes it returns markdown-fenced JSON" failure mode. There is no JSON at all, as far as you're concerned.

```swift
import FoundationModels

@Generable
struct MeetingDigest {
    @Guide(description: "A one-sentence summary of the meeting, 25 words or fewer.")
    let summary: String

    @Guide(description: "Every concrete action item mentioned in the notes.")
    let actionItems: [ActionItem]
}

@Generable
struct ActionItem {
    @Guide(description: "The task, phrased as an imperative, e.g. 'Ship the beta build'.")
    let task: String

    @Guide(description: "The person responsible, exactly as named in the notes, or 'Unassigne
    let owner: String
}
```

`@Generable` synthesizes a generation schema from the type; `@Guide` descriptions are the part of your prompt that lives next to the field it describes, which is exactly where a 3B-parameter model needs it. Small models follow *local* instructions far better than a wall of rules at the top of a prompt. Chapter 4 goes deep on constraints (`anyOf`, ranges, element counts); plain descriptions are enough here.

## 3. The session and the request

```swift
import FoundationModels
```

```
func digest(notes: String) async throws -> MeetingDigest {
    let session = LanguageModelSession(
        instructions: """
        You extract structured digests from raw meeting notes. \
        Be literal: never invent action items that are not in the notes.
        """
    )
    let response = try await session.respond(
        to: "Digest these meeting notes:\n\n\(notes)",
        generating: MeetingDigest.self
    )
    return response.content
}
```

That's the whole feature, minimally. Three details that matter:

- **Instructions vs. prompt.** Instructions come from you and are privileged; the prompt contains user content. Keep untrusted text out of `instructions`. This is the framework's prompt-injection boundary, and Chapter 11 covers why it matters for review.
- **Sessions are cheap; context is not.** A session accumulates a transcript. For a one-shot feature like this, make a fresh session per request so a long day of digests can't overflow the context window. The window is about 4K tokens *total*: instructions, prompt, and output share it. Long meeting notes are your first real-world failure; we handle it in §5.
- **Latency.** If you can predict the user is about to run the feature (they opened the digest sheet), call `session.prewarm()` to page the model in before the tap.

## 4. Streaming into SwiftUI

A full digest takes a few seconds on an A17 Pro. A spinner for five seconds reads like a hang; streaming reads like speed. Guided generation streams *snapshots of the typed value*, a `MeetingDigest.PartiallyGenerated`, which is the same shape with every field optional, so your UI fills in as fields complete.

```
import FoundationModels
import SwiftUI

@Observable @MainActor
final class DigestViewModel {
    var summary: String = ""
    var items: [String] = []
    var errorMessage: String?

    func run(notes: String) async {
        let session = LanguageModelSession(
            instructions: "You extract structured digests from raw meeting notes."
        )
        do {
            let stream = session.streamResponse(
```

```
                to: "Digest these meeting notes:\n\n\(notes)",
                generating: MeetingDigest.self
            )
            for try await snapshot in stream {
                summary = snapshot.content.summary ?? ""
                items = (snapshot.content.actionItems ?? [])
                    .compactMap(\.task)
            }
        } catch {
            errorMessage = friendlyMessage(for: error)
        }
    }
}
```

Two things to remember:

- Snapshots are *cumulative*, not deltas. Assign, don't append.
- Don't animate every snapshot individually; snapshots arrive fast enough to fight the animation system. `withAnimation` on the assignment with a short duration is plenty.

## 5. The errors you will actually see

`LanguageModelSession.GenerationError` is where production quality lives. Three cases dominate field reports:

| Error | Real-world cause | What to do |
|---|---|---|
| `exceededContextWindowSize` | User pasted a 90-minute transcript | Chunk the notes, digest per chunk, then digest the digests (map-reduce). Or truncate with consent. |
| `guardrailViolation` | Notes contained content the safety layer flags; HR meetings are a classic | Don't retry (it's near-deterministic). Show "couldn't digest these notes" and *don't* log the content. |
| `unsupportedLanguageOrLocale` | Notes in a language the model doesn't handle | Check `supportedLanguages` up front if your user base is multilingual. |

```
func friendlyMessage(for error: Error) -> String {
    if let generationError = error as? LanguageModelSession.GenerationError {
        switch generationError {
        case .exceededContextWindowSize:
            return "These notes are too long to digest in one pass."
        case .guardrailViolation:
            return "Digest couldn't process these notes."
        default:
            return "Digest hit a snag. Please try again."
```

```
        }
    }
    return "Digest hit a snag. Please try again."
}
```

The `guardrailViolation` row deserves emphasis: your first instinct will be to log the failing input to debug it. The input is a user's private meeting. The entire premise of shipping local AI is that this content never leaves the device, and your analytics pipeline counts as leaving. Log the error case, never the content. Chapter 11 turns this into policy.

## What ships

- ☐ Availability checked at feature entry, with distinct copy for `deviceNotEligible` / `appleIntelligenceNotEnabled` / `modelNotReady`, re-checked on foreground
- ☐ Output modeled as `@Generable` types; zero JSON parsing anywhere
- ☐ Fresh session per digest; `prewarm()` on sheet presentation
- ☐ Streaming UI with cumulative snapshot assignment
- ☐ All three headline `GenerationError` cases handled; user content never logged
- ☐ Feature flag so the whole thing can be disabled remotely (Chapter 7)

Next chapter: what `@Generable` is actually doing under the hood, and how constrained decoding lets you delete an entire class of output-validation code.