Sample Code

Adding intelligent app features with generative models

Build robust apps with guided generation and tool calling by adopting the Foundation Models framework.

Download

iOS 26.0+ | iPadOS 26.0+ | macOS 26.0+ | visionOS 26.0+ | Xcode 26.0+

Overview

This sample project shows how to integrate generative AI capabilities into an app using the Foundation Models framework. The sample app showcases intelligent trip planning features that help people discover landmarks and generate personalized itineraries.

The app creates an interactive experience where people can:

- Browse curated landmarks with rich visual content
- Generate trip itineraries tailored to a chosen landmark
- Discover points of interest using a custom tool
- Experience real-time content generation with streaming responses

> **Note**
>
> This sample code project is associated with WWDC25 session 259: Code-along: Add Intelligence to your App using the Foundation Models framework.

Configure the sample code project

To run this sample, you'll need to:

1. Set the developer team in Xcode for the app target so it automatically manages the provisioning profile. For more information, see Set the bundle ID and Assign the project to a team.
2. In the Developer portal, enable the WeatherKit app service for your bundle ID so the app can access location-based weather information.

Check model availability

Before using the on-device model in the app, check that the model is available by creating an instance of `SystemLanguageModel` with the `default` property:

```
let landmark: Landmark
private let model = SystemLanguageModel.default

var body: some View {
    switch model.availability {
    case .available:
        LandmarkTripView(landmark: landmark)
    case .unavailable(.appleIntelligenceNotEnabled):
        MessageView(
            landmark: self.landmark,
            message: """
                     Trip Planner is unavailable because \
                     Apple Intelligence hasn't been turned on.
                     """
        )
    case .unavailable(.modelNotReady):
        MessageView(
            landmark: self.landmark,
            message: "Trip Planner isn't ready yet. Try again later."
        )
    }
}
```

The app handles two unavailability scenarios: Apple Intelligence isn't enabled or the model isn't ready for usage. If Apple Intelligence is off, the app tells the person they need to turn it on and if the model isn't ready, it tells the person the Trip Planner isn't ready and to try the app again later.

> **Note**
>
> To use the on-device language model, people need to turn on Apple Intelligence on their device. For a list of supported devices, see Apple Intelligence.

Define structured data for generation

The app starts by defining data structures with specific constraints to control what the model generates. The `Itinerary` type uses the `Generable` macro to create structured content that includes travel plans with activities, hotels, and restaurants.

The `@Generable` macro automatically converts Swift types into schemas that the model uses for constrained sampling, so you can specify guides to control the values you associate with it. For example, the app uses `Guide(description:)` to make sure the model creates an exciting name for the trip. It also uses `anyOf(_:)` and `count(_:)` to choose any destination from our `ModelData` and show exactly 3 `DayPlan` objects per destination, respectively.

```
@Generable
struct Itinerary: Equatable {
    @Guide(description: "An exciting name for the trip.")
    let title: String
    @Guide(.anyOf(ModelData.landmarkNames))
    let destinationName: String
    let description: String
    @Guide(description: "An explanation of how the itinerary meets the person's spec
    let rationale: String

    @Guide(description: "A list of day-by-day plans.")
    @Guide(.count(3))
    let days: [DayPlan]
}

@Generable
struct DayPlan: Equatable {
    @Guide(description: "A unique and exciting title for this day plan.")
    let title: String
    let subtitle: String
    let destination: String

    @Guide(.count(3))
    let activities: [Activity]
```

```
}

@Generable
struct Activity: Equatable {
    let type: Kind
    let title: String
    let description: String
}

@Generable
enum Kind {
    case sightseeing
    case foodAndDining
    case shopping
    case hotelAndLodging
}
```

The `@Generable` macro automatically creates two versions of each type: the complete structure and a `PartiallyGenerated` version which is a mirror of the outer structure except every property is optional. The app uses this `PartiallyGenerated` version when streaming and displaying the itinerary generation.

Configure the model session

After checking that the model is available, the app configures a `LanguageModelSession` object with custom tools and detailed instructions in `ItineraryPlanner`. Given a location, the initializer creates the session with structured guidance for generating personalized trip recommendations.

```
init(landmark: Landmark) {
    self.landmark = landmark
    Logging.general.log("The landmark is... \(landmark.name)")
    let pointOfInterestTool = FindPointsOfInterestTool(landmark: landmark)
    self.session = LanguageModelSession(
        tools: [pointOfInterestTool],
        instructions: Instructions {
            "Your job is to create an itinerary for the person."

            "Each day needs an activity, hotel and restaurant."

            """
            Always use the findPointsOfInterest tool to find businesses \
```

```
            and activities in \(landmark.name), especially hotels \
            and restaurants.

            The point of interest categories may include:
            """
            FindPointsOfInterestTool.categories

            """
            Here is a description of \(landmark.name) for your reference \
            when considering what activities to generate:
            """
            landmark.description
        }
    )
    self.pointOfInterestTool = pointOfInterestTool
}
```

In a generated itinerary, the model instructions ensure that each day contains an activity, hotel, and restaurant. To get the location-specific businesses and activities, the sample uses a custom tool, called `FindPointsOfInterestTool`, with the chosen landmark. The instructions also call the landmark description property as added context when generating the activities.

Create a custom tool

You can use custom tools to extend the functionality of a model. Tool-calling allows the model to interact with external code you create to fetch up-to-date information, ground responses in sources of truth that you provide, and perform side effects.

The model in this app uses the `FindPointsOfInterestTool` tool to enable dynamic discovery of specific businesses and activities for the chosen landmark. The tool uses the `@Generable` macro to make its categories and arguments available to the model.

```
@Observable
final class FindPointsOfInterestTool: Tool {
    let name = "findPointsOfInterest"
    let description = "Finds points of interest for a landmark."

    let landmark: Landmark

    @MainActor var lookupHistory: [Lookup] = []

    init(landmark: Landmark) {
```

```
        self.landmark = landmark
    }

@Generable
enum Category: String, CaseIterable {
    case campground
    case hotel
    case cafe
    case museum
    case marina
    case restaurant
    case nationalMonument
}

@Generable
struct Arguments {
    @Guide(description: "This is the type of destination to look up for.")
    let pointOfInterest: Category

    @Guide(description: "The natural language query of what to search for.")
    let naturalLanguageQuery: String
}
```

When you prompt the model with a question or make a request, the model decides whether it can provide an answer or if it needs the help of a tool. The app explicitly instructs the model to always use the `findPointsOfInterestTool` in the `ItineraryPlanner` instructions. This allows the model to automatically call the tool to find relevant hotels, restaurants, and activities for the destinations.

Stream and display partial responses in real time

The app shows real-time content generation by streaming partial responses from the model. The `ItineraryPlanner` uses `streamResponse(generating:includeSchemaInPrompt:options:prompt:)` to generate `Itinerary.PartiallyGenerated` objects so itinerary items are shown incrementally to the person.

You can opt for specific `GenerationOptions` to adjust the way the model generates these responses. For generating the itinerary, the app opts for a `greedy` sampling mode so the model always results in the same output for a given input. This ensures the prompt generates consistent recommendations for an itinerary specific to the given landmark.

```
private(set) var itinerary: Itinerary.PartiallyGenerated?

func suggestItinerary(dayCount: Int) async throws {
    let stream = session.streamResponse(
        generating: Itinerary.self,
        includeSchemaInPrompt: false,
        options: GenerationOptions(sampling: .greedy)
    ) {
        "Generate a \(dayCount)-day itinerary to \(landmark.name)."

        "Give it a fun title and description."

        "Here is an example, but don't copy it:"
        Itinerary.exampleTripToJapan
    }

    for try await partialResponse in stream {
        itinerary = partialResponse.content
    }
}
```

The app presents the responses in a SwiftUI view. The `ItineraryPlanningView` displays real-time visual feedback as the model searches for points of interest, showing people what's happening when generating content:

```
ForEach(planner.pointOfInterestTool.lookupHistory) { element in
    HStack {
        Image(systemName: "location.magnifyingglass")
        Text("Searching **\(element.history.pointOfInterest.rawValue)** in \(landmar
    }
    .transition(.blurReplace)
}
```

The app displays messages like "Searching **hotel** in Yosemite…" and "Searching **restaurant** in Yosemite…" to let people know which point of interest category the model provided as input to the tool when actively searching for nearby points of interest. In the background, however, the tool executes and provides updates to the view. The view shows a blurred overlay while generating each day plan, then reveals the full itinerary after the search completes.

Tag content dynamically

The app uses content tagging on the provided landmarks to help people quickly understand the characteristics of each destination. A content tagging model produces a list of categorizing tags based on the input text you provide. When you prompt the content tagging model, it produces a tag that uses one to a few lowercase words. The `LandmarkDescriptionView` prompts the content tagging model to automatically generate relevant hashtags for landmark descriptions, like `#nature`, `#hiking`, or `#scenic`, based on each landmark's description. For more information on initializing content tagging, see Categorizing and organizing data with content tags.

```
let contentTaggingModel = SystemLanguageModel(useCase: .contentTagging)

.task {
    if !contentTaggingModel.isAvailable { return }
    do {
        let session = LanguageModelSession(model: contentTaggingModel)
        let stream = session.streamResponse(
            to: landmark.description,
            generating: TaggingResponse.self,
            options: GenerationOptions(sampling: .greedy)
        )
        for try await newTags in stream {
            generatedTags = newTags.content
        }
    } catch {
        Logging.general.error("\(error.localizedDescription)")
    }
}
```

Integrate with other framework features

You can combine these generative model features with other Apple frameworks. For example, the `LocationLookup` class uses MapKit to search for addresses for our points of interest, showing how to combine model-generated content with weather information and location data for complete travel planning.

```
@Observable @MainActor
final class LocationLookup {
    private(set) var item: MKMapItem?
    private(set) var temperatureString: String?

    func performLookup(location: String) {
        Task {
```

```
            let item = await self.mapItem(atLocation: location)
            if let location = item?.location {
                self.temperatureString = await self.weather(atLocation: location)
            }
        }
    }

    private func mapItem(atLocation location: String) async -> MKMapItem? {
        let request = MKLocalSearch.Request()
        request.naturalLanguageQuery = location

        let search = MKLocalSearch(request: request)
        do {
            return try await search.start().mapItems.first
        } catch {
            Logging.general.error("Failed to look up location: \(location). Error: \
        }
        return nil
    }
}
```

The model generates location names as text, and the `LocationLookup` class converts them into real, mappable locations using the natural language search capabilities in MapKit.

See Also

Essentials

- Generating content and performing tasks with Foundation Models
  Enhance the experience in your app by prompting an on-device large language model.
- Improving the safety of generative model output
  Create generative experiences that appropriately handle sensitive inputs and respect people.
- Support languages and locales with Foundation Models
  Generate content in the language people prefer when they interact with your app.

`class` SystemLanguageModel
An on-device large language model capable of text generation tasks.

`struct` UseCase

A type that represents the use case for prompting.