Article

Expanding generation with tool calling

Build tools that enable the model to perform tasks that are specific to your use case.

Overview

Tools provide a way to extend the functionality of the model for your own use cases. 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, like turning on dark mode.

You can create tools that enable the model to:

- Query entries from your app's database and reference them in its answer.
- Perform actions within your app, like adjusting the difficulty in a game or making a web request to get additional information.
- Integrate with other frameworks, like Contacts or HealthKit, that use existing privacy and security mechanisms.

Create a custom tool for your task

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. When the model determines that a tool can help, it calls the tool with additional arguments that the tool can use. After the tool completes the task, it returns control back to the model with information about what the tool did. The model can then use the output of the tool when it provides the final response.

Before creating a tool, it's helpful to understand the pattern the framework follows when using the tool you provide. The framework processes a request in six phases:

1. You present a list of available tools and their parameters to the model.

2. You submit your prompt to the model.

3. The model generates arguments to the tool(s) it wants to invoke.

4. Your tool runs code on behalf of the model, using the model's generated arguments.

5. Your tool passes its output back to the model.

6. The model produces a final response to the prompt, based on the tool output.

A tool conforms to `Tool` and contains the arguments that the tool accepts, and a method that the model calls when it wants to use the tool. You can call `call(arguments:)` concurrently with itself or with other tools. The following example shows a tool that accepts a search term and a number of recipes to retrieve:

```
struct BreadDatabaseTool: Tool {
    let name = "searchBreadDatabase"
    let description = "Searches a local database for bread recipes."

    @Generable
    struct Arguments {
        @Guide(description: "The type of bread to search for")
        var searchTerm: String
        @Guide(description: "The number of recipes to get", .range(1...6))
        var limit: Int
    }

    struct Recipe {
        var name: String
        var description: String
        var link: URL
    }

    func call(arguments: Arguments) async throws -> [String] {
        var recipes: [Recipe] = []

        // Put your code here to retrieve a list of recipes from your database.

        let formattedRecipes = recipes.map {
            "Recipe for '\($0.name)': \($0.description) Link: \($0.link)"
        }
        return formattedRecipes
    }
```

```
}
```

When you provide descriptions to generable properties, you help the model understand the semantics of the arguments. Keep descriptions as short as possible because long descriptions take up context size and can introduce latency. For more information on managing the context window size, see TN3193: Managing the on-device foundation model's context window.

Tools use guided generation for the `Arguments` property. For more information about guided generation, see Generating Swift data structures with guided generation.

Provide a session with the tool you create

When you create a session, you can provide a list of tools that are relevant to the task you want to complete. The tools you provide are available for all future interactions with the session. The following example initializes a session with a tool that the model can call when it determines that it would help satisfy the prompt:

```
let session = LanguageModelSession(
    tools: [BreadDatabaseTool()]
)

let response = try await session.respond(
    to: "Find three sourdough bread recipes"
)
```

Tool output can be a string, or a `GeneratedContent` object. The model can call a tool multiple times in parallel to satisfy the request, like when retrieving weather details for several cities:

```
struct WeatherTool: Tool {
  let name = "getWeather"
  let description = "Retrieve the latest weather information for a city"

  @Generable
  struct Arguments {
      @Guide(description: "The city to get weather information for")
      var city: String
  }

  struct Forecast: Encodable {
      var city: String
      var temperature: Int
```

```
  }

  func call(arguments: Arguments) async throws -> String {
      // Get a random temperature value. Use `WeatherKit` to get
      // a temperature for the city.
      let temperature = Int.random(in: 30...100)
      let formattedResult = """
          The forecast for '\(arguments.city)' is '\(temperature)' \
          degrees Fahrenheit.
          """
      return formattedResult
  }
}

// Create a session with default instructions that guide the requests.
let session = LanguageModelSession(
    tools: [WeatherTool()],
    instructions: "Help the person with getting weather information"
)

// Make a request that compares the temperature between several locations.
let response = try await session.respond(
    to: "Is it hotter in Boston, Wichita, or Pittsburgh?"
)
```

Handle errors thrown by a tool

When an error happens during tool calling, the session throws a `LanguageModelSession.ToolCallError` with the underlying error and includes the tool that throws the error. This helps you understand the error that happened during the tool call, and any custom error types that your tool produces. You can throw errors from your tools to escape calls when you detect something is wrong, like when the person using your app doesn't allow access to the required data or a network call is taking longer than expected. Alternatively, your tool can return a string that briefly tells the model what didn't work, like "Cannot access the database."

```
do {
    let answer = try await session.respond("Find a recipe for tomato soup.")
} catch let error as LanguageModelSession.ToolCallError {

    // Access the name of the tool, like BreadDatabaseTool.
    print(error.tool.name)
```

```
    // Access an underlying error that your tool throws and check if the tool
    // encounters a specific condition.
    if case .databaseIsEmpty = error.underlyingError as? SearchBreadDatabaseToolErro
        // Display an error in the UI.
    }

} catch {
    print("Some other error: \(error)")
}
```

Inspect the call graph

A session contains an observable `transcript` property that allows you to track when, and how many times, the model calls your tools. A transcript also provides the ability to construct a representation of the call graph for debugging purposes and pairs well with SwiftUI to visualize session history.

```
struct MyHistoryView: View {

    @State
    var session = LanguageModelSession(
        tools: [BreadDatabaseTool()]
    )

    var body: some View {
        List(session.transcript) { entry in
            switch entry {
            case .instructions(let instructions):
                // Display the instructions the model uses.
            case .prompt(let prompt):
                // Display the prompt made to the model.
            case .toolCall(let call):
                // Display the call details for a tool, like the tool name and argum
            case .toolOutput(let output):
                // Display the output that a tool provides back to the model.
            case .response(let response):
                // Display the response from the model.
            }
        }.task {
            do {
                try await session.respond(to: "Find a milk bread recipe.")
            } catch let error {
```

```
                // Handle the error.
            }
        }
    }

}
```

See Also

Tool calling

{} Generate dynamic game content with guided generation and tools

Make gameplay more lively with AI generated dialog and encounters personalized to the player.

`protocol Tool`

A tool that a model can call to gather information at runtime or perform side effects.