ChatGPT

Integrating Apple's On-Device LLM (Foundation Models) into Your Chatbot App

Current App Architecture Overview

Your **"Bryan's Brain"** iOS app is structured with SwiftUI views (e.g. `ChatView`, `TodoListView`, etc.) and a `ChatViewModel` that orchestrates AI interactions. Currently, the chatbot uses a remote OpenAI model (`gpt-4o-mini`) via an `OpenAIService` class. User messages and context are sent to OpenAI's API, which returns either a text reply or a **tool function call** request. The app defines a set of **tools** (functions like `addTaskToList`, `listCurrentTasks`, `createCalendarEvent`, etc.) to manage the to-do list and calendar [1] [2]. When the model responds with a tool request, `ChatViewModel` executes the corresponding Swift function (updating the task list, calendar, etc.), then sends the function's result back into the conversation loop [3] [4]. This loop continues until the model returns a final answer. In summary:

- **UI Layer**: SwiftUI displays chat messages and to-do items.
- **ViewModel Layer**: Maintains `messages` history and calls `OpenAIService.processConversation()` to get model replies [5]. Also handles tool-call results (adding tasks, listing tasks, etc.) and appends those results as special `.tool` messages in the chat history [6] [7].
- **Backend/AI Layer**: `OpenAIService` prepares JSON for the OpenAI Chat Completion API (including function definitions as "tools") [8] [9], and sends it to the cloud endpoint. It then parses the response into either a plain text answer or a structured tool-call request [10] [11]. This system requires an internet connection and incurs OpenAI API costs (and latency).

Apple's On-Device Foundation Models (WWDC 2025)

With iOS 26 and Xcode 26 (announced at WWDC 2025), Apple introduced the **Foundation Models framework** – a new API that lets developers run Apple's large language model **on-device** [12]. This is the same core model behind *Apple Intelligence* features (like advanced Siri, translations, etc.), but now accessible to third-party apps. The key benefits of using Apple's on-device foundation model are:

- **No Cloud Needed**: All inference runs locally on the device's Neural Engine or CPU/GPU, so you **avoid remote servers and API costs entirely** [12]. This preserves user privacy and works offline by design [13].
- **Performance and Optimization**: The default model is approximately a 3-billion-parameter LLM, **quantized to 2 bits** for efficiency [14]. Apple has optimized it for speed and low memory usage on Apple Silicon. It runs on modern hardware (iPhone 15 Pro and later, iPads/Macs with M1 or newer chips) [15]. Older devices may not support it.
- **Native Swift API**: The Foundation Models framework is built into iOS 26+ with first-class Swift support. Basic usage is extremely simple – Apple touted that you can integrate the model with **as few as three lines of code** [16]. There's no need to manage HTTP calls or JSON; you interact with the model using Swift types and async/await.

- **Capabilities**: The on-device model is general-purpose (trained on a wide domain) and supports advanced features like **guided generation** (structured output with type-safety) and **tool calling** (the model can invoke custom Swift functions you provide) [14]. These align closely with what your app already does with OpenAI function calls, meaning you can replicate that agentic behavior locally.

In short, Apple's framework will let your app's AI assistant run entirely on the user's device – no internet required – while leveraging Apple's optimized LLM. Next, we'll dive into how to use this new API in your Swift code and integrate it into your existing app structure.

Getting Started: Using the FoundationModels Framework in Swift

1. Importing and Setup: To use the on-device model, import the new framework and create a model session. In Xcode 26, Apple provides the `FoundationModels` module. Typically, you begin with:

```
import FoundationModels

// Create a new language model session (stateful conversation context)
let session = try LanguageModelSession()
```

This initializes a session with Apple's default foundation model. You don't need to specify a model name or weight file – the system automatically selects the on-device LLM (currently a ~3B parameter general model) available on the device [14]. If desired, you can provide an initial **system prompt** (Apple calls them **"instructions"**) when creating the session. For example:

```
let session = try LanguageModelSession(
    instructions: "You are a helpful task assistant. Reply with step-by-step
plans."
)
```

The `instructions` parameter sets the **global context or persona** for the session (analogous to a system message) [17]. This is useful to guide the model's tone or role – in your case, you might instruct it to act as a productivity coach or to focus on managing tasks. If no instructions are given, the model just uses its default behavior.

2. Sending Prompts and Receiving Responses: Once you have a session, you send user input to the model using the `respond` function. This is an async call that yields the model's answer. For example:

```
// User prompt (could come from a TextField in your UI):
let prompt = "Plan my day. I have 3 tasks and one meeting at 2 PM."

// Ask the model for a response
let responseText = try await session.respond(to: prompt)
print("Assistant:", responseText)
```

The above will asynchronously generate a response string from the on-device LLM. Under the hood, the session maintains a **conversation transcript** – it remembers previous prompts and responses in this `LanguageModelSession` context. Each `respond(to:)` call **appends** the new user prompt to the session's history and produces the next assistant reply [18]. This means your chatbot can handle multi-turn conversations naturally. (If you ever need to start a fresh chat context, you can create a new `LanguageModelSession` instance or reset the session.)

By default, `respond(to:)` returns a `String` containing the model's full answer. This is great for basic usage or free-form outputs. Apple's API design hides the complexity – you don't manually manage tokens or parse JSON. For instance, one official example shows how simple it can be:

```
let session = LanguageModelSession()
let prompt = "Rum old fashioned cocktail"
let response = try await session.respond(to: prompt,
                                        options: GenerationOptions(temperature:
2.0))
```

This produces a cocktail recipe or instructions, all with one call [17]. The FoundationModels framework handles feeding the prompt to the model (along with any prior context in the session) and returns the generated text.

3. Controlling Generation (Temperature & Options): The framework provides a `GenerationOptions` structure to tweak generation parameters for each request. You can adjust things like `temperature` (randomness/creativity), maximum tokens, and other decoding settings. For example, to get more creative or varied answers, you might set a higher temperature:

```
let options = GenerationOptions(temperature: 1.2)  // 1.2 for more randomness
(default might be ~0.7)
let response = try await session.respond(to: prompt, options: options)
```

Lower temperatures (e.g. 0.2) make outputs more deterministic/repetitive, while higher values make the model more inventive (at the risk of sometimes straying off-track) [19]. You can adjust this per query as needed. `GenerationOptions` likely also includes other parameters (top-p, presence/frequency penalties, etc.), though Apple's documentation emphasizes simplicity – you might not need to fine-tune much for your use case initially.

Note on model selection: At the time of WWDC 2025, you do *not* manually select from multiple language models – the API uses Apple's built-in general-purpose model by default. (The framework name is plural "Foundation Models" because it encompasses various modalities and allows Apple to add more models in the future, possibly including vision or domain-specific models. But for now, think of it as one primary LLM that's automatically chosen based on device capabilities.) There is no API key or model ID to supply – if the device supports on-device inference, your `LanguageModelSession` will load the appropriate model from the system. This greatly simplifies integration: just instantiate and go.

Integrating the On-Device LLM into Your App's Chat Flow

Replacing your current OpenAI-based logic with the FoundationModels framework will substantially simplify the code:

- **Single-step prompt/response:** You no longer need the complex loop of sending a request, checking if it contains tool calls, executing them, and calling the API again for the final answer. The on-device model (with proper setup) can handle tool usage internally and return a final answer in one `respond()` call [20]. Your `ChatViewModel.sendMessage` logic can thus be streamlined to: append the new user query to the UI, call `session.respond` to get the assistant's answer, then append the answer for display. The session keeps track of context on its own [18], so you don't need to manually build a `messages` array to resend on every turn (unlike constructing the OpenAI API payload each time).

- **Session management:** You might create one `LanguageModelSession` when the chat starts (for example, when your `ChatViewModel` initializes), and reuse it for the whole conversation. This ensures the AI remembers earlier messages. If the user resets the chat or starts a new planning session, you can drop that session and make a new one (or use a fresh `instructions` prompt to "reset" context). Each session is analogous to one chat transcript.

- **Error handling:** Interacting with the on-device model is done with `try/await` because it can throw errors – e.g., if the model isn't available or if something goes wrong during generation. You should be prepared to catch errors. For instance, the first time you call `LanguageModelSession()`, it might throw if the device hasn't downloaded the model or if running on an unsupported simulator. In such a case, you can inform the user or fall back to a cloud service. Generally, on a supported device with iOS 26+, the model should load automatically (it may download in the background or be pre-installed).

- **Device compatibility:** When testing, **use a real device** with the required hardware or a Mac with Apple Silicon. The simulator may not have the model available. (Apple's developer forums indicate the models might not run in Simulator, producing an error about missing assets [21].) So plan to run on an iPhone 15 Pro/16 or later for development testing. Also ensure the app's deployment target is iOS 26.0+ for the FoundationModels framework.

Now, let's address how to integrate your **tool/functionalities** (to-do list and calendar actions) with Apple's framework.

Tool Calling with FoundationModels

One of the most powerful features of Apple's on-device LLM is built-in **tool calling support**, very similar in spirit to OpenAI's function calling that your app already uses. The model can decide to invoke a custom function (Apple calls it a "Tool") to, say, fetch data or perform an action, and then incorporate the result into its reply. Apple designed this to be *autonomous* and seamless: *"Tool calls will happen transparently and the model will incorporate the tool's outputs into its final response."* [22]. This means you do not have to manually loop and resend context as you did with the OpenAI approach – the framework handles it.

Defining a Tool: In the FoundationModels framework, a tool is any Swift type conforming to the `Tool` protocol. You essentially wrap one of your app's functions in a type that the LLM can call. The protocol's requirements (based on WWDC examples) include: - A `name` (String) – the identifier the model will use to call the tool (e.g. `"addTaskToList"`).
- A `description` (String) – a natural language description of what the tool does, for the model's understanding.
- An **arguments type** (conforming to `Generable` or `Codable`) – this defines what parameters the tool expects when invoked. You can use a struct to represent these. Apple's system uses Swift's powerful type system to ensure the model's calls are valid (this is part of *Guided Generation*).
- A `call(...)` method – the implementation of the tool's action. This is the code that runs when the model decides to use the tool. It should return a `ToolOutput` value, which encapsulates the result to give back to the model.

Let's walk through an example by converting your **"add task"** functionality into a tool. In your current code, adding a task is done via a function (perhaps in `TodoListStore`) that takes a task description (and possibly category or project info) and updates the list. We'll expose this as a tool:

```
import FoundationModels

struct AddTaskTool: Tool {
    // 1. Define the input parameters structure
    struct Arguments: Codable, Sendable {
        let taskDescription: String
        let category: String?
        let projectOrPath: String?
        // ...include other fields like priority or difficulty if needed
    }

    // 2. Tool metadata for the LLM
    let name = "addTaskToList"
    let description =
"Adds a new task to the to-do list with a description, and optionally a category
or project."

    // 3. The function to execute when the LLM calls this tool
    func call(arguments: Arguments) throws -> ToolOutput {
        // Perform the actual task addition in your app's data model:
        TodoListStore.shared.addTask(text: arguments.taskDescription,
                                     category: arguments.category,
                                     project: arguments.projectOrPath)
        // Prepare a result to return to the model:
        let confirmation = "Added task: \"\(arguments.taskDescription)\""
        return ToolOutput(confirmation)  // wrap a simple confirmation message
    }
}
```

Let's unpack that: We created an `AddTaskTool` struct conforming to `Tool`. The `Arguments` inner struct defines what data the model should provide – here a required `taskDescription` and optional `category` and `projectOrPath` (matching the parameters your OpenAI function expected [23]). The `call` method uses those arguments to add the task via your existing `TodoListStore` and returns a result. We simply return a `ToolOutput` created from a String message. (The `ToolOutput` type is provided by the framework; it can be initialized with a plain string for a natural language result, or with a `GeneratedContent` for structured data [24].)

Attaching Tools to the Session: To let the model use this tool, you attach it when creating the `LanguageModelSession`. The session initializer has a parameter `tools: [Tool]`. For example:

```
// Initialize session with our tools
let session = try LanguageModelSession(
    tools: [AddTaskTool(), ListTasksTool(), /* ... other tools ... */],
    instructions: "You are an AI assistant for a to-do app. Use tools like
addTaskToList to help manage tasks."
)
```

Here we pass an array of tool instances. Each tool's `name` and `description` are provided to the model under the hood, so it knows what functions are available and when to use them [25]. **Important:** Attach all needed tools at session creation – you cannot dynamically add tools mid-session. In your case, you would create tools for each capability: adding tasks, listing tasks, marking complete, creating calendar events, etc., and include them in the session's tool list.

Apple's framework will then autonomously decide during a `respond(to:)` call if a tool is needed. For instance, if the user says *"Add 'Buy groceries' to my to-do list"*, the model might invoke `addTaskToList` internally. When `respond` is called, the framework detects the model's intent to use the tool, runs your `AddTaskTool.call(...)` method (on a background thread since tools must be thread-safe/Sendable [26]), gets the `ToolOutput` (e.g., confirmation text or data), and gives that back to the model to continue generation. The **final output** you receive from `session.respond` will already incorporate the tool's result, e.g., *"OK, I've added 'Buy groceries' to your to-do list."* – without you writing extra logic to handle it. All those steps happen within the single `respond` call!

This is a big difference from your current OpenAI loop where the app had to handle the function call. With on-device FoundationModels, the model orchestrates tool use internally, thanks to the tool definitions you provided. It uses a mechanism called **Guided Generation** to ensure it produces valid tool names/arguments and parses the output. As Apple explained, the framework uses Swift's type system and *constrained decoding* to **guarantee the model's tool call outputs match your Swift types** [27] [28]. That means if your `Arguments` expects a `String` and an optional `Int`, the model will be guided to only emit those types (preventing nonsense or malformed requests). This yields more reliable tool usage.

ToolOutput and Structured Data: If a tool needs to return structured data (say, a list of tasks or events), you can use `ToolOutput(GeneratedContent(...))`. For example, imagine a `ListTasksTool` that fetches all current tasks and returns them as an array. You could do:

```
func call(arguments: Void) -> ToolOutput {
    let tasks = TodoListStore.shared.allTasks()  // returns [TodoItem]
    // Convert to an array of strings or dicts for each task:
    let taskStrings = tasks.map { $0.text }
    return ToolOutput( GeneratedContent(taskStrings) )
}
```

Here, `GeneratedContent(taskStrings)` creates a content object the model can treat as an array of strings. The framework would ensure the model receives it as such (for example, it might incorporate it into the final answer like "Here are your tasks: ['Task1', 'Task2', ...]"). Using `GeneratedContent` is optional – if you prefer, you can format the list into a single string and return `ToolOutput("Task1, Task2, ...")`. But leveraging structured output can make the model's job easier and the integration more robust (Apple's guided generation excels with structure). In either case, the model's final answer to the user can be a fluent description that includes the tool-derived data.

Tool Errors: If something goes wrong inside a tool (e.g., a calendar insertion fails), your `call` can throw an error. The framework will catch it and deliver a `ToolCallError` back to the model, which the model can then handle or apologize for [29]. You can also choose to return an error message as a normal `ToolOutput` so the model sees it. This way, the AI can respond, "Sorry, I couldn't add that event." Handling errors gracefully will ensure a good UX.

In summary, to integrate all your agent's capabilities: - Implement each action (task management, calendar ops) as a `Tool` conforming to Apple's API. (Much of this mirrors your existing functions, just wrapped in the `Tool` structure.) - Instantiate the `LanguageModelSession` with all these tools. Provide a suitable `instructions` string so the AI knows it should use tools for certain requests (e.g., *"As an assistant, you can manage a task list and calendar. Use the provided tools to do so when necessary."*). - Then simply call `session.respond(to: userInput)` for each user message. The model will choose on its own when to call a tool. You'll get the final answer which you can display in the chat.

Your SwiftUI `ChatView` can still display a running list of messages, but you won't explicitly show the intermediate "tool invoked" messages to the user (unless you want to for transparency). In your current UI, you filtered those out anyway [30]. With the new approach, the model's reply already includes any tool results, so you'll typically just append one assistant message per user query.

Streaming Responses for a Better UX

Apple's framework also supports **streaming generation**, which can improve the responsiveness of your UI by showing the answer as it's being composed (just like many AI chat apps that display text token-by-token). Under FoundationModels, streaming is handled via **snapshots** of the output rather than raw tokens, which is especially handy if you're generating structured data [31] [32].

To use streaming, you call a different API: `session.streamResponse(to:prompt)`. This returns an `AsyncSequence` of **partial results**. You can iterate over it in an async context. For example:

```
for try await partial in session.streamResponse(to: prompt) {
    // Update UI with the partial result
    liveText = partial  // `partial` might be a String or a partially-filled
struct
}
```

In SwiftUI, you could bind a state variable to `partial` so that as each new chunk arrives, the view updates. The "partial" type is actually a special struct (automatically generated) that has the same shape as the final output type but with all fields optional [32] [33]. Each iteration gives a newer snapshot with more fields/tokens filled in. If you're just streaming plain text, this effectively gives you an incremental build-up of the string. If you're streaming a complex type (say an `Itinerary` with multiple properties), you'll see its properties appear one by one. The final iteration of the sequence is the complete output.

Using streaming in your app could mean the assistant's message bubbles appear as they're typed out, which is great for longer answers. You'll want to handle the UI carefully (maybe append a temporary "typing..." message and update it). Also consider using SwiftUI animations for a smooth experience while new content appears [34].

If you don't need streaming right away, you can initially stick with the simpler `respond(to:)` (which internally buffers until completion). But it's nice to know you can easily upgrade to streaming with a bit of async sequence handling. For completeness, here's a pseudo-code snippet integrating streaming in ChatViewModel:

```
@MainActor
func sendMessageStreaming(_ text: String) {
    messages.append(ChatMessage(role: .user, content: text))
    // Start streaming response
    Task {
       do {
           let stream = session.streamResponse(to: text)  // could add options
or generating type if needed
           var accumulated = ""
           for try await partial in stream {
               // Assuming partial is a String here
               accumulated = partial
               // Update a published property to show partial text
               self.partialResponse = accumulated
           }
           // Once done, move partial to final message list
           messages.append(ChatMessage(role: .assistant, content: accumulated))
           self.partialResponse = nil
       } catch {
           print("Error during streaming: \(error)")
       }
```

```
    }
}
```

This way, `partialResponse` could be bound to a SwiftUI text view representing the typing answer. The final result is added to `messages` when the stream completes.

Testing and Debugging the Integration

Adopting this new on-device model will require **Xcode 26** (or newer) and iOS 26 for your app. Here are some tips to test and debug effectively:

- **Use Supported Devices:** As mentioned, run the app on real hardware with Apple Neural Engine (iPhone 15 Pro/16, iPad with M-series, or Mac with M1/M2 for Mac Catalyst apps). The foundation model may not run in the iOS Simulator. If you attempt to and see errors like *"no underlying assets for asset set com.apple.MobileAsset.UAF.FM..."*, that's a sign to switch to a real device [35]. On first run, the device might download the model in the background – allow some time or ensure it's connected to Wi-Fi.

- **Performance Considerations:** The 3B model is fairly lightweight as far as LLMs go, but generation time can vary with prompt length and device. In testing, try queries of different sizes. You can also experiment with the `maxTokens` option in `GenerationOptions` to limit how much the model will generate (if you want to cap response length for speed). Apple's optimizations (2-bit quantization and Neural Engine usage) should make it quite snappy for moderate-length prompts.

- **Debugging Tool Calls:** Verify that your tools are being invoked as expected. You can place `print()` statements or breakpoints in your tool `call()` methods. For instance, in `AddTaskTool.call`, print a message whenever it's called, to see that the model is indeed triggering it. Apple's framework might call tools in parallel if multiple are requested, so be mindful of thread-safety (your `TodoListStore` should handle concurrent calls or you may serialize tool usage if needed). The tools conform to `Sendable` so the system may run them concurrently [26].

- **Logging Model Behavior:** While you don't have direct access to the model's "thought process" (like you did by inspecting intermediate tool call messages), you can glean some insight by the outcomes. If the model isn't using a tool when it should, consider refining the tool's description or the session instructions to "nudge" it. For example, if you find the assistant sometimes lists tasks by itself instead of calling your `listCurrentTasks` tool, you might adjust the instruction: *"You have tools to manage the user's tasks and calendar — always use them for those operations."* This helps the model decide to use the provided functions rather than rely on its own text knowledge. You can also test tool calling with known prompts: e.g., ask "What are my pending tasks?" and see if it calls the list tool. Apple's model was trained with tool use in mind [14], so it should handle this gracefully, but small tweaks in prompt wording can help.

- **Fallbacks and Errors:** If the on-device model fails or isn't available (for instance, on an older phone), you might implement a fallback to your previous cloud approach. Since your architecture is modular, you could detect at runtime if FoundationModels is supported, and if not, use `OpenAIService` as a backup. This way, users on unsupported devices still get functionality (albeit with network usage).

- **Xcode Previews:** SwiftUI previews likely won't be able to run the model (since previews run on Mac but not in a full app context). So don't expect the AI to function in canvas preview. Test in the simulator (for UI layout only) and on devices for the AI itself.

- **Memory and Battery:** Monitor memory usage when the model loads. The 3B model quantized is quite small (perhaps a few hundred MB at most in RAM), but it will still consume memory. It should fit comfortably on modern devices. The first response might be slower due to model loading; subsequent calls are faster. Also be aware of battery impact if the user has long back-and-forth chats – on-device AI uses CPU/GPU intensively. In practice, Apple likely optimized for bursts of usage, but keep an eye on it during testing.

Conclusion

By integrating Apple's Foundation Models framework, you can update your chatbot app to use **on-device AI** seamlessly. The **existing app structure** – with a chat interface and defined tool functions – actually maps very well onto Apple's new API. You'll eliminate the dependency on external services, which means no more latency from network calls and no ongoing API costs [12]. User data stays private on the device. And with Apple's optimizations, the experience should feel native and responsive.

In summary, the steps to implement are:

1. **Upgrade to Xcode 26** and set iOS 26 as a deployment target. Import `FoundationModels` in your project.
2. **Create a LanguageModelSession** (likely one per chat session) with any initial instructions and all your relevant tools attached [25].
3. **Implement Tools** for each action (to-do and calendar functions) by conforming to `Tool`. Return results via `ToolOutput` (using strings or `GeneratedContent` for structured info) [24].
4. **Replace OpenAI calls** in your ViewModel with `session.respond(to:userMessage)` calls. Handle the returned answer by updating your `messages` array to display it. The heavy lifting of function-calling and context management is handled internally by the framework.
5. **Test on real devices**, ensuring the on-device model works and that each tool function is invoked correctly when the user asks for those operations. Fine-tune prompt wording or tool descriptions if needed to encourage the right behavior.
6. **Consider enabling streaming** output to the UI for a more dynamic feel, using `streamResponse` and updating SwiftUI views progressively [33].

By following this integration path, your app's chatbot logic will be both simpler and more powerful. You'll have a self-contained AI assistant that leverages Apple's state-of-the-art on-device model – delivering quick, privacy-preserving responses and deeply integrating with your app's features. And since everything runs locally, your users can trust that their data (tasks, schedule, conversations) never leaves their device, fulfilling one of Apple's key promises for this technology [13].

Sources:

- Apple WWDC 2025 news – Foundation Models on-device AI [12] [16]
- Apple Developer Documentation and WWDC sessions on FoundationModels usage [17] [25] [24] [33]
- "Bryan's Brain" code (GitHub) – existing architecture and OpenAI integration for context [2] [23]

- TechRepublic WWDC 2025 summary (iOS 26 features and device support) [15]

---

[1] README.md
https://github.com/actonbp/deep/blob/b83cbc371ce3a137f24ff10a1c87ed2a6a362d1f/README.md

[2] [8] [9] [10] [11] [23] OpenAIService.swift
https://github.com/actonbp/deep/blob/b83cbc371ce3a137f24ff10a1c87ed2a6a362d1f/deep_app/deep_app/OpenAIService.swift

[3] [4] [5] [6] [7] ChatViewModel.swift
https://github.com/actonbp/deep/blob/b83cbc371ce3a137f24ff10a1c87ed2a6a362d1f/deep_app/deep_app/ChatViewModel.swift

[12] [15] Apple WWDC 2025: iOS 26 Will Be Available This Fall
https://www.techrepublic.com/article/apple-wwdc-keynote-ios-macos/

[13] WWDC 2025 Recap: A New Era for Apple Developers | by Pushpsen Airekar | Jun, 2025 | Medium
https://pushpsenairekar.medium.com/wwdc-2025-recap-a-new-era-for-apple-developers-51a1086c1a1c

[14] [16] [17] WWDC: Apple supercharges its tools and technologies for developers
https://simonwillison.net/2025/Jun/9/apple-wwdc/

[18] [20] [22] [24] [25] [27] [28] [31] [32] [33] [34] Meet the Foundation Models framework - WWDC25 - Videos - Apple Developer
https://developer.apple.com/videos/play/wwdc2025/286/

[19] Generating content and performing tasks with Foundation Models
https://developer.apple.com/documentation/foundationmodels/generating-content-and-performing-tasks-with-foundation-models

[21] [35] Apple Developer Forums
https://developer.apple.com/forums/topics/machine-learning-and-ai

[26] Tool | Apple Developer Documentation
https://developer.apple.com/documentation/foundationmodels/tool

[29] Expanding generation with tool calling - Apple Developer
https://developer.apple.com/documentation/foundationmodels/expanding-generation-with-tool-calling

[30] ChatView.swift
https://github.com/actonbp/deep/blob/b83cbc371ce3a137f24ff10a1c87ed2a6a362d1f/deep_app/deep_app/ChatView.swift