Article

Loading and using a custom adapter with Foundation Models

Specialize the behavior of the system language model by using a custom adapter you train.

Overview

Use an adapter to adapt the on-device foundation model to fit your specific use case without needing to retrain the entire model from scratch. Before you can load a custom adapter, you first need to train one with an adapter training toolkit. The toolkit uses Python and Pytorch, and requires familiarity with training machine-learning models. After you train an adapter, you can use the toolkit to export a package in the format that Xcode and the Foundation Models framework expects.

When you train an adapter you need to make it available for deployment into your app. An adapter file is large — 160 MB or more — so don't bundle them in your app. Instead, use App Store Connect, or host the asset on your server, and download the correct adapter for a person's device on-demand.

Important

Each adapter is compatible with a *single specific* system model version. You must train a new adapter for every new base model version. A runtime error occurs if your app runs on a person's device without a compatible adapter.

For more information about the adapter training toolkit, see Get started with Foundation Models adapter training. For more information about asset packs, see Background Assets.

Test a local adapter in Xcode

After you train an adapter with the adapter training toolkit, store your `.fmadapter` package files in a different directory from your app. Then, open `.fmadapter` packages with Xcode to locally preview each adapter's metadata and version compatibility before you deploy the adapter.

If you train multiple adapters:

1. Find the adapter package that's compatible with the macOS version of the Mac on which you run Xcode.
2. Select the compatible adapter file in Finder.
3. Copy its full file path to the clipboard by pressing Option + Command + C.
4. Initialize `SystemLanguageModel.Adapter` with the file path.

```
// The absolute path to your adapter.
let localURL = URL(filePath: "absolute/path/to/my_adapter.fmadapter")

// Initialize the adapter by using the local URL.
let adapter = try SystemLanguageModel.Adapter(fileURL: localURL)
```

After you initialize an `Adapter`, create an instance of `SystemLanguageModel` with it:

```
// An instance of the the system language model using your adapter.
let customAdapterModel = SystemLanguageModel(adapter: adapter)

// Create a session and prompt the model.
let session = LanguageModelSession(model: customAdapterModel)
let response = try await session.respond(to: "Your prompt here")
```

Important

Only import adapter files into your Xcode project for local testing, then remove them before you publish your app. Adapter files are large, so download them on-demand by using Background Assets.

Testing adapters requires a physical device and isn't supported on Simulator. When you're ready to deploy adapters in your app, you need the `com.apple.developer.foundation-model-adapter` entitlement. You don't need this entitlement to train or locally test adapters. To request access to use the entitlement, log in to Apple Developer and see Foundation Models Framework Adapter Entitlement.

Bundle adapters as asset packs

When people use your app they only need the specific adapter that's compatible with their device. Host your adapter assets on a server and use Background Assets to manage downloads. For hosting, you can use your own server or have Apple host your adapter assets. For more information about Apple-hosted asset packs, see Overview of Apple-hosted asset packs.

The Background Assets framework has a type of asset pack specific to adapters that you create for the Foundation Models framework. The Foundation Models adapter training toolkit helps you bundle your adapters in the correct asset pack format. The toolkit uses the `ba-package` command line tool that's included with Xcode 16 or later. If you train your adapters on a Linux GPU machine, see How to train adapters to set up a Python environment on your Mac. The adapter toolkit includes example code that shows how to create the asset pack in the correct format.

After you generate an asset pack for each adapter, upload the asset packs to your server. For more information about uploading Apple-hosted adapters, see Upload Apple-Hosted asset packs.

Configure an asset-download target in Xcode

To download adapters at runtime, you need to add an asset-downloader extension target to your Xcode project:

1. In Xcode, choose File > New > Target.
2. Choose the Background Download template under the Application Extension section.
3. Click next.
4. Enter a descriptive name, like "AssetDownloader", for the product name.
5. Select the type of extension.
6. Click Finish.

The type of extension depends on whether you self-host them or Apple hosts them:

Apple-Hosted, Managed
: Apple hosts your adapter assets.

Self-Hosted, Managed
: You use your server and make each device's operating system automatically handle the download life cycle.

Self-Hosted, Unmanaged
: You use your server and manage the download life cycle.

After you create an asset-downloader extension target, check that your app target's info property list contains the required fields specific to your extension type:

Apple-Hosted, Managed

- `BAHasManagedAssetPacks` = YES
- `BAAppGroupID` = The string ID of the app group that your app and downloader extension targets share.
- `BAUsesAppleHosting` = YES

Self-Hosted, Managed

- `BAHasManagedAssetPacks` = YES
- `BAAppGroupID` = The string ID of the app group that your app and downloader extension targets share.

If you use *Self-Hosted, Unmanaged*, then you don't need additional keys. For more information about configuring background assets with an extension, see Configuring an unmanaged Background Assets project

Choose a compatible adapter at runtime

When you create an asset-downloader extension, Xcode generates a Swift file — `BackgroundDownloadHandler.swift` — that Background Assets uses to download your adapters. Open the Swift file in Xcode and fill in the code based on your target type. For *Apple-Hosted, Managed* or *Self-Hosted, Managed* extension types, complete the function `shouldDownload` with the following code that chooses an adapter asset compatible with the runtime device:

```swift
func shouldDownload(_ assetPack: AssetPack) -> Bool {
    // Check for any non-adapter assets your app has, like shaders. Remove the
    // check if your app doesn't have any non-adapter assets.
    if assetPack.id.hasPrefix("mygameshader") {
        // Return false to filter out asset packs, or true to allow download.
        return true
    }

    // Use the Foundation Models framework to check adapter compatibility with the
    return SystemLanguageModel.Adapter.isCompatible(assetPack)
}
```

If your extension type is *Self-Hosted, Unmanaged*, the file Xcode generates has many functions in it for manual control over the download life cycle of your assets.

Load adapter assets in your app

After you configure an asset-downloader extension, you can start loading adapters. Before you download an adapter, remove any outdated adapters that might be on a person's device:

```
SystemLanguageModel.Adapter.removeObsoleteAdapters()
```

Create an instance of `SystemLanguageModel.Adapter` using your adapter's base name, but exclude the file extension. If a person's device doesn't have a compatible adapter downloaded, your asset-downloader extension starts downloading a compatible adapter asset pack:

```
let adapter = try SystemLanguageModel.Adapter(name: "myAdapter")
```

Initializing a `SystemLanguageModel.Adapter` starts a download automatically when a person launches your app for the first time or their device needs an updated adapter. Because adapters can have a large data size they can take some time to download, especially if a person is on Wi-Fi or a cell network. If a person doesn't have a network connection, they aren't able to use your adapter right away. This method shows how to track the download status of an adapter:

```
func checkAdapterDownload(name: String) async -> Bool {
    // Get the ID of the compatible adapter.
    let assetpackIDList = SystemLanguageModel.Adapter.compatibleAdapterIdentifiers(
        name: name
    )

    if let assetPackID = assetpackIDList.first {
        // Get the download status asynchronous sequence.
        let statusUpdates = AssetPackManager.shared.statusUpdates(forAssetPackWithID

        // Use the current status to update any loading UI.
        for await status in statusUpdates {
            switch status {
            case .began(let assetPack):
                // The download started.
            case .paused(let assetPack):
                // The download is in a paused state.
            case .downloading(let assetPack, let progress):
                // The download in progress.
            case .finished(let assetPack):
                // The download is complete and the adapter is ready to use.
                return true
            case .failed(let assetPack, let error):
                // The download failed.
                return false
            @unknown default:
                // The download encountered an unknown status.
                fatalError()
            }
        }
    }
}
```

For more details on tracking downloads for general assets, see Downloading Apple-hosted asset packs.

Before you attempt to use the adapter, you need to wait for the status to be in a `AssetPackManager.DownloadStatusUpdate.finished(_:)` state. The system returns `AssetPackManager.DownloadStatusUpdate.finished(_:)` immediately if no download is necessary.

```
// Load the adapter.
let adapter = try SystemLanguageModel.Adapter(name: "myAdapter")

// Wait for download to complete.
if await checkAdapterDownload(name: "myAdapter") {
    // Adapt the base model with your adapter.
    let adaptedModel = SystemLanguageModel(adapter: adapter)

    // Start a session with the adapted model.
    let session = LanguageModelSession(model: adaptedModel)

    // Start prompting the adapted model.
}
```

Compile your draft model

A draft model is an optional step when training your adapter that can speed up inference. If your adapter includes a draft model, you can compile it for faster inference:

```
// Load the adapter.
let adapter = try SystemLanguageModel.Adapter(name: "myAdapter")

// Wait for download to complete.
```

```swift
if await checkAdapterDownload(name: "myAdapter") {
    do {
        // You can use your adapter without compiling the draft model, or during
        // compilation, but running inference with your adapter might be slower.
        try await adapter.compile()
    } catch let error {
        // Handle the draft model compilation error.
    }
}
```

For more about training draft models, see the "Optionally train the draft model" section in Get started with Foundation Models adapter training.

Compiling a draft model is a computationally expensive step, so use the Background Tasks framework to configure a background task for your app. In your background task, call `compile()` on your adapter to start compilation. For more information about using background tasks, see Using background tasks to update your app.

Compilation doesn't run every time a person uses your app:

- The first time a device downloads a new version of your adapter, a call to `compile()` fully compiles your draft model and saves it to the device.
- During subsequent launches of your app, a call to `compile()` checks for a saved compiled draft model and returns it immediately if it exists.

Important

Rate limiting protects device resources that are shared between all apps and processes. If the framework determines that a new compilation is necessary, it rate-limits the compilation process on all platforms, excluding macOS, to three draft model compilations per-app, per-day.

The full compilation process runs every time you launch your app through Xcode because Xcode assigns your app a new UUID for every launch. If you receive a rate-limiting error while testing your app, stop your app in Xcode and re-launch it to reset the rate counter.

See Also

Loading the model with an adapter

`com.apple.developer.foundation-model-adapter`

A Boolean value that indicates whether the app can enable custom adapters for the Foundation Models framework.

```
convenience init(adapter: SystemLanguageModel.Adapter, guardrails:
SystemLanguageModel.Guardrails)
```

Creates the base version of the model with an adapter.

```
struct Adapter
```

Specializes the system language model for custom use cases.