Article

Improving the safety of generative model output

Create generative experiences that appropriately handle sensitive inputs and respect people.

Overview

Generative AI models have powerful creativity, but with this creativity comes the risk of unintended or unexpected results. For any generative AI feature, safety needs to be an essential part of your design.

The Foundation Models framework has two base layers of safety, where the framework uses:

- An on-device language model that has training to handle sensitive topics with care.
- *Guardrails* that aim to block harmful or sensitive content, such as self-harm, violence, and adult materials, from both model input and output.

Because safety risks are often contextual, some harms might bypass both built-in framework safety layers. It's vital to design additional safety layers specific to your app. When developing your feature, decide what's acceptable or might be harmful in your generative AI feature, based on your app's use case, cultural context, and audience.

For more information on designing generative AI experiences responsibly, see Human Interface Guidelines > Foundations > Generative AI.

Handle guardrail errors

When you send a prompt to the model, SystemLanguageModel.Guardrails check the input prompt and the model's output. If either fails the guardrail's safety check, the model session throws a LanguageModelSession.GenerationError.guardrailViolation(_:) error:

```
do {
    let session = LanguageModelSession()
    let topic = // A potentially harmful topic.
    let prompt = "Write a respectful and funny story about \(topic)."
    let response = try await session.respond(to: prompt)
} catch LanguageModelSession.GenerationError.guardrailViolation {
    // Handle the safety error.
}
```

If you encounter a guardrail violation error for any built-in prompt in your app, experiment with re-phrasing the prompt to determine which phrases are activating the guardrails, and avoid those phrases. If the error is thrown in response to a prompt created by someone using your app, give people a clear message that explains the issue. For example, you might say "Sorry, this feature isn't designed to handle that kind of input" and offer people the opportunity to try a different prompt.

Handle model refusals

The on-device language model may not be suitable for handling all requests and may refuse requests for a topic. When you generate a string response, and the model refuses a request, it generates a message that begins with a refusal like "Sorry, I can't help with".

Design your app experience with refusal messages in mind and present the message to the person using your app. You might not be able to programmatically determine whether a string response is a normal response or a refusal, so design the experience to anticipate both. If it's critical to determine whether the response is a refusal message, initialize a new LanguageModelSession and prompt the model to classify whether the string is a refusal.

When you use guided generation to generate Swift structures or types, there's no placeholder for a refusal message. Instead, the model throws a LanguageModelSession.GenerationError.refusal(_:_:) error. When you catch the error, you can ask the model to generate a string refusal message:

```swift
do {
    let session = LanguageModelSession()
    let topic = ""  // A sensitive topic.
    let response = try session.respond(
        to: "List five key points about: \(topic)",
        generating: [String].self
    )
} catch LanguageModelSession.GenerationError.refusal(let refusal, _) {
    // Generate an explanation for the refusal.
    if let message = try? await refusal.explanation {
        // Display the refusal message.
    }
}
```

Display the explanation in your app to tell people why a request failed, and offer people the opportunity to try a different prompt. Retrieving an explanation message is asynchronous and takes time for the model to generate.

If you encounter a refusal message, or refusal error, for any built-in prompts in your app, experiment with re-phrasing your prompt to avoid any sensitive topics that might cause the refusal.

For more information about guided generation, see Generating Swift data structures with guided generation.

Build boundaries on input and output

Safety risks increase when a prompt includes direct input from a person using your app, or from an unverified external source, like a webpage. An untrusted source makes it difficult to anticipate what the input contains. Whether accidentally or on purpose, someone could input sensitive content that causes the model to respond poorly.

> **Tip**
>
> The more you can define the intended usage and outcomes for your feature, the more you can ensure generation works great for your app’s specific use cases. Add boundaries to limit out-of-scope usage and minimize low generation quality from out-of-scope uses.

Whenever possible, avoid open input in prompts and place boundaries for controlling what the input can be. This approach helps when you want generative content to stay within the bounds of a particular topic or task. For the highest level of safety on input, give people a fixed set of prompts to choose from. This gives you the highest certainty that sensitive content won’t make its way into your app:

```
enum TopicOptions {
    case family
    case nature
    case work
}
let topicChoice = TopicOptions.nature
let prompt = """
    Generate a wholesome and empathetic journal prompt that helps \
    this person reflect on \(topicChoice)
    """
```

If your app allows people to freely input a prompt, placing boundaries on the output can also offer stronger safety guarantees. Using guided generation, create an enumeration to restrict the model’s output to a set of predefined options designed to be safe no matter what:

```
@Generable
enum Breakfast {
    case waffles
    case pancakes
    case bagels
    case eggs
}
let session = LanguageModelSession()
let userInput = "I want something sweet."
let prompt = "Pick the ideal breakfast for request: \(userInput)"
let response = try await session.respond(to: prompt, generating: Breakfast.self)
```

Instruct the model for added safety

Consider adding detailed session `Instructions` that tell the model how to handle sensitive content. The language model prioritizes following its instructions over any prompt, so instructions are an effective tool for improving safety and overall generation quality. Use uppercase words to emphasize the importance of certain phrases for the model:

```
do {
    let instructions = """
        ALWAYS respond in a respectful way. \
        If someone asks you to generate content that might be sensitive, \
        you MUST decline with 'Sorry, I can't do that.'
        """
    let session = LanguageModelSession(instructions: instructions)
    let prompt = // Open input from a person using the app.
    let response = try await session.respond(to: prompt)
} catch LanguageModelSession.GenerationError.guardrailViolation {
    // Handle the safety error.
}
```

> Note
>
> A session obeys instructions over a prompt, so don’t include input from people or any unverified input in the instructions. Using unverified input in instructions makes your app vulnerable to prompt injection attacks, so write instructions with content you trust.

If you want to include open-input from people, instructions for safety are recommended. For an additional layer of safety, use a format string in normal prompts that wraps people’s input in your own content that specifies how the model should respond:

```
let userInput = // The input a person enters in the app.
let prompt = """
    Generate a wholesome and empathetic journal prompt that helps \
    this person reflect on their day. They said: \(userInput)
    """
```

Add a deny list of blocked terms

If you allow prompt input from people or outside sources, consider adding your own deny list of terms. A deny list is anything you don't want people to be able to input to your app, including unsafe terms, names of people or products, or anything that's not relevant to the feature you provide. Implement a deny list similarly to guardrails by creating a function that checks the input and the model output:

```
let session = LanguageModelSession()
let userInput = // The input a person enters in the app.
let prompt = "Generate a wholesome story about: \(userInput)"

// A function you create that evaluates whether the input
// contains anything in your deny list.
if verifyText(prompt) {
    let response = try await session.respond(to: prompt)

    // Compare the output to evaluate whether it contains anything in your deny li
    if verifyText(response.content) {
        return response
    } else {
        // Handle the unsafe output.
    }
} else {
    // Handle the unsafe input.
}
```

A deny list can be a simple list of strings in your code that you distribute with your app. Alternatively, you can host a deny list on a server so your app can download the latest deny list when it's connected to the network. Hosting your deny list allows you to update your list when you need to and avoids requiring a full app update if a safety issue arise.

Use permissive guardrail mode for sensitive content

The default `SystemLanguageModel` guardrails may throw a `LanguageModelSession.GenerationError.guardrailViolation(_:)` error for sensitive source material. For example, it may be appropriate for your app to work with certain inputs from people and unverified sources that might contain sensitive content:

- When you want the model to tag the topic of conversations in a chat app when some messages contain profanity.
- When you want to use the model to explain notes in your study app that discuss sensitive topics.

To allow the model to reason about sensitive source material, use `permissiveContentTransformations` when you initialize `SystemLanguageModel`:

```
let model = SystemLanguageModel(guardrails: .permissiveContentTransformations)
```

This mode only works for generating a string value. When you use guided generation, the framework runs the default guardrails against model input and output as usual, and generates `LanguageModelSession.GenerationError.guardrailViolation(_:)` and `LanguageModelSession.GenerationError.refusal(_:_:)`errors as usual.

Before you use permissive content mode, consider what's appropriate for your audience. The session skips the guardrail checks in this mode, so it never throws a `LanguageModelSession.GenerationError.guardrailViolation(_:)` error when generating string responses.

However, even with the `SystemLanguageModel` guardrails off, the on-device system language model still has a layer of safety. For some content, the model may still produce a refusal message that's similar to, "Sorry, I can't help with."

Create a risk assessment

Conduct a risk assessment to proactively address what might go wrong. Risk assessment is an exercise that helps you brainstorm potential safety risks in your app and map each risk to an actionable mitigation. You can write a risk assessment in any format that includes these essential elements:

- List each AI feature in your app.
- For each feature, list possible safety risks that could occur, even if they seem unlikely.
- For each safety risk, score how serious the harm would be if that thing occurred, from mild to critical.

- For each safety risk, assign a strategy for how you'll mitigate the risk in your app.

For example, an app might include one feature with the fixed-choice input pattern for generation and one feature with the open-input pattern for generation, which is higher safety risk:

Feature	Harm	Severity	Mitigation
Player can input any text to chat with nonplayer characters in the coffee shop.	A character might respond in an insensitive or harmful way.	Critical	Instructions and prompting to steer characters responses to be safe; safety testing.
Image generation of an imaginary dream customer, like a fairy or a frog.	Generated image could look weird or scary.	Mild	Include in the prompt examples of images to generate that are cute and not scary; safety testing.
Player can make a coffee from a fixed menu of options.	None identified.		
Generate a review of the coffee the player made, based on the customer's order.	Review could be insulting.	Moderate	Instructions and prompting to encourage posting a polite review; safety testing.

Besides obvious harms, like a poor-quality model output, think about how your generative AI feature might affect people, including real-world scenarios where someone might act based on information generated by your app.

Write and maintain safety tests

Although most people will interact with your app in respectful ways, it's important to anticipate possible failure modes where certain input or contexts could cause the model to generate something harmful. Especially if your app takes input from people, test your experience's safety on input like:

- Input that is nonsensical, snippets of code, or random characters.

- Input that includes sensitive content.
- Input that includes controversial topics.
- Vague or unclear input that could be misinterpreted.

Create a list of potentially harmful prompt inputs that you can run as part of your app's tests. Include every prompt in your app — even safe ones — as part of your app testing. For each prompt test, log the timestamp, full input prompt, the model's response, and whether it activates any built-in safety or mitigations you've included in your app. When starting out, manually read the model's response on all tests to ensure it meets your design and safety goals. To scale your tests, consider using a frontier LLM to auto-grade the safety of each prompt. Building a test pipeline for prompts and safety is a worthwhile investment for tracking changes in how your app responds over time.

Someone might purposefully attempt to break your feature or produce bad output — especially someone who won't be harmed by their actions. But, keep in mind that it's very important to identify cases where someone might *accidentally* be harmed during normal app use.

> **Tip**
>
> Prioritize protecting people using your app with good intentions. Accidental safety failures often only occur in specific contexts, which make them hard to identify during testing. Test for a longer series of interactions, and test for inputs that could become sensitive only when combined with other aspects of your app.

Don't engage in any testing that could cause you or others harm. Apple's built-in responsible AI and safety measures, like safety guardrails, are built by experts with extensive training and support. These built-in measures aim to block egregious harms, allowing you to focus on the borderline harmful cases that need your judgement. Before conducting any safety testing, ensure that you're in a safe location and that you have the health and well-being support you need.

Report safety concerns

Somewhere in your app, it's important to include a way that people can report potentially harmful content. Continuously monitor the feedback you receive, and be responsive to quickly handling any safety issues that arise. If someone reports a safety concern that you believe isn't being properly handled by Apple's built-in guardrails, report it to Apple with Feedback Assistant.

The Foundation Models framework offers utilities for feedback. Use `LanguageModelFeedback` to retrieve language model session transcripts from people using your app. After collecting feedback, you can serialize it into a JSONL file and include it in the report you send with Feedback Assistant.

Monitor safety for model or guardrail updates

Apple releases updates to the system model as part of regular OS updates. If you participate in the developer beta program you can test your app with new model version ahead of people using your app. When the model updates, it’s important to re-run your full prompt tests in addition to your adversarial safety tests because the model’s response may change. Your risk assessment can help you track any change to safety risks in your app.

Apple may update the built-in guardrails at any time outside of the regular OS update cycle. This is done to rapidly respond, for example, to reported safety concerns that require a fast response. Include all of the prompts you use in your app in your test suite, and run tests regularly to identify when prompts start activating the guardrails.

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.

Support languages and locales with Foundation Models

Generate content in the language people prefer when they interact with your app.

Adding intelligent app features with generative models

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

`class SystemLanguageModel`

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

`struct UseCase`

A type that represents the use case for prompting.