Article

TN3193: Managing the on-device foundation model's context window

Learn how to budget for the context window limit of Apple's on-device foundation model and handle the error when reaching the limit.

Overview

The Foundation Models framework provides access to Apple's on-device foundation model at the core of Apple Intelligence. With the framework, you can build AI-powered features that enhance your app.

Like other Large Language Models (LLMs), Apple's on-device foundation model processes text in units called tokens. A token corresponds to roughly three to four characters in Latin alphabet languages like English, as elaborated in the WWDC25 session 286: Meet the Foundation Models framework. For multi-byte languages like Chinese, Japanese, and Korean, it is roughly one character per token.

The maximum number of tokens a LLM can process at once is called the context window. Context window size is determined by model architecture and hardware limits. An on-device LLM running on your iPhone needs a relatively small context window to run efficiently.

Apple's on-device foundation model has a context window of 4096 tokens per language model session. To adapt to this limit, consider techniques to effectively budget for the context window, achieve your use-cases using fewer tokens, and handle the error when you reach the limit.

Understand how your app consumes tokens

When you interact with a LLM, the model converts your input to a token sequence, and uses it to generate a probability distribution that reflects which token in the model's vocabulary could be the next. Based on the decoding strategy, which is derived from the generation options when you use

the Foundation Models framework, the model picks the next token, appends it to the sequence, and uses the updated sequence as the input for the next generation cycle. This process repeats until it reaches a stop condition. The final token sequence, after being converted to human-readable content, is the response you get.

With the Foundation Models framework, you interact with the model using instructions, prompts, tool calling, and Generable types, which are passed to the model as part of the input. All the input and response in the generation process contribute tokens to the context window of the current language model session, including instructions, all prompts, the information of tools (schemas, input, and output), `Generable` schemas, and all the model's responses.

The Foundation Models instrument allows you to profile your app to observe token consumption while your app is running. To use the instrument:

1. In Xcode, open your project and click Product > Profile to launch Instruments.

2. Select the Blank template, and click the Choose button.

3. Click the + Instrument button, and choose the Foundation Models instrument from the list.

4. Start recording your app, have your app interact with the models, and observe the token count.

For more information about profiling your app with the instrument, see the WWDC25 session 259: Code-along: Bring on-device AI to your app using the Foundation Models framework.

Split a large task into multiple language model sessions

When doing a task that needs a larger context window size, explore if you can split the task into smaller steps, run each step with a new language model session, and then assemble the results together.

As an example, to generate a summary for a long article on device, consider separating the article into smaller chunks that the model can handle, summarizing each chunk with a new session, combining the results together, and then repeating this process, until getting a summary with ideal size. To avoid completely losing the context of the article when summarizing a chunk, consider adding the result of the previous summarization to the prompt so it conveys the contextual information.

Ask the model for less content

One way to budget tokens is to ask the model to produce fewer response tokens. If you notice the model producing long, detailed responses, try:

- Include your target response length in your prompt, for example, “In 3 sentences....” or “List 3 reasons that...”.
- Add a `@Guide` to any `Generable` arrays (for example, tag lists or name lists) you are generating and specify the max count using `maximumCount(_:)`.

Use `maximumResponseTokens` only when you need to protect against unexpectedly verbose responses and runaway generations, since enforcing a strict token response limit can lead the model to produce malformed results or grammatically incorrect partial responses like “A cat is a small.”

Reduce the prompt size

Prompts and instructions can consume a lot of tokens, especially in multi-turn scenarios where you’re sending multiple prompts to the same language model session. Writing shorter prompts and instructions is a way to save tokens without sacrificing response quality.

Give the model only the information needed for a specific task. Avoid instructions that give the model significant amount background information, policy, or extra context. Long instructions with large amount information consume more tokens, and may lower the quality of the responses.

Use concise and imperative language in your instructions and prompts. Make sure your prompt has a clear verb that tells the model what to do, for example, “Generate a story...” or “List 5 things that...”. Avoid indirect language, formal language, or jargon the foundation model might misinterpret. Aim for a maximum of 1–3 paragraphs for prompts and instructions, and in general, use the shortest prompt that can achieve your task.

Use Generable types efficiently

`Generable` types consume tokens in multiple ways. For every `Generable` type in your generation request, the framework converts its type and format information to a JSON schema, and passes that schema text to the model. Any `@Guide` descriptions you write also consume tokens. To make your `Generable` types more efficient:

- Reduce the size and complexity of your type. As a rule of thumb, think about how much screen space your `@Generable` code takes with normal code formatting. More screen space roughly corresponds to more token use.
- Give your properties short, clear names.
- Use `@Guide` only where needed. If your `Generable` type’s properties are clearly named, the model may have all the information it needs to generate your type. Try generating first with no `@Guide` annotations, and then add in `@Guide` annotations where needed to improve response quality.

Use tool calling efficiently

Tool calling allows the model to request and incorporate the information from your code, but consumes tokens in a similar fashion to `Generable` types. When you provide a tool in your generation request, the framework puts the tool definitions, which includes the tool name, description, and parameter information, in the prompt so the model can decide when and how often to call the tool. When the model calls a tool, the framework returns the tool's output back to the model for further processing, which consumes additional tokens. To use tool calling efficiently:

- Keep your tool description and `@Guide` descriptions to a short phrase each.
- Give the model a maximum of 3–5 tools to choose from.

Look for opportunities to save tokens by skipping tool calling entirely. Use a tool only when you need the model to decide if it needs the tool. In the cases where the model should always have information from a tool, run the tool directly before you call the model and integrate the tool's output to the prompt directly.

If you're reaching the context window limit, consider breaking up tool calls across multiple language model sessions, if appropriate for your use case. In cases where you need the model to generate appropriate tool arguments, consider asking the model to generate those in one session, then run your tool using normal programming, then have the model process the tool's output in a new, second session.

Integrate a retrieval system to fetch information dynamically

Retrieval-Augmented Generation, or RAG, is a technique that combines a retrieval system (like a search engine or vector database) with a language model. If your use case has a large amount of information, notes, or documents you'd like the model to reference, you may have too much information to fit in the context window. Using RAG, you can dynamically fetch snippets of the relevant information when needed, and pass only the snippets to the model to stay within the context window.

There are many methods for RAG, but they typically follow these general steps:

1. Choose an approach that fits your use case for text chunking and embedding.
2. Split your knowledge base into chunks, vectorize the chunks into embeddings, and store the result in a database.
3. Gather a user query, vectorize it, and use the result to retrieve the most relevant chunks from the database.
4. Feed the query and the most relevant chunks to the model and collect the response.

For the first step, consider using a chunking model and an embedding model. The former splits large pieces of text to smaller ones; the latter takes text as input, vectorizes it, and outputs a list of numbers that represents the text. After determining the models that work for your use case, integrate them into your app using APIs such as Core ML. The Natural Language framework provides APIs for tokenizing and embedding text — if that meets your needs. See Tokenizing natural language text and Finding similarities between pieces of text for more information.

In the second step, vectorizing the whole knowledge base may be computationally heavy and take time. You can do it with a dedicated data preparation process that runs separate from the app and then stores the final chunks and embeddings in your app, or makes them available to the app through a server your app uses. RAG can be used as a tool call, or as a step you run before calling the on-device foundation model.

Handle the exceeding context window size error elegantly

Even with carefully designed architecture and prompts, you might still exceed the context window limit in some cases. In an open-ended conversation implemented with one large language session, for example, people can continue the chat for long time, and eventually reach the limit.

When that happens, the Foundation Models framework throws an `.exceededContextWindowSize` error, and the session won't be able to respond. To catch the error:

```
do {
    let response = try await largeLanguageSession.respond(to: <prompt>)
    ...
} catch let error as LanguageModelSession.GenerationError.exceededContextWindowSize(
    ... // Handle .exceededContextWindowSize error.
} catch let error {
    ... // Handle other errors.
}
```

To handle the error, consider creating a new session to continue your workflow. A new session has a new context window, but doesn't convey the state of the original session. If you need to keep the state, consider the following options:

- Collect the content of the original session through its `transcript` property, do a summary, and create a new session with the result.
- Pick some important entries from the original session's transcript, and use them to create a new session.

The following example shows how to create a new session with the first and last entries of the original session:

```
func newContextualSession(with originalSession: LanguageModelSession) -> LanguageMod
    let allEntries = originalSession.transcript
    let condensedEntries = [allEntries.first, allEntries.last].compactMap { $0 }
    let condensedTranscript = Transcript(entries: condensedEntries)
    var newSession = LanguageModelSession(transcript: condensedTranscript)
    newSession.prewarm()
    return newSession
}
```

For an example that includes instructions and tool calling, see Generate dynamic game content with guided generation and tools.

Revision History

- **2025-10-06** First published.

See Also

Latest

TN3190: USB audio device design considerations

Learn the best techniques for designing devices that conform to the USB Audio Device Class specifications.

TN3194: Handling account deletions and revoking tokens for Sign in with Apple

Learn the best techniques for managing Sign in with Apple user sessions and responding to account deletion requests.

TN3115: Bluetooth State Restoration app relaunch rules

Learn about the conditions under which an iOS app will be relaunched by Bluetooth State Restoration.

TN3192: Migrating your iPad app from the deprecated UIRequiresFullScreen key

Support iPad multitasking and dynamic resizing while updating your app to remove the deprecated full-screen compatibility mode.

TN3151: Choosing the right networking API

Learn which networking API is best for you.

TN3111: iOS Wi-Fi API overview

Explore the various Wi-Fi APIs available on iOS and their expected use cases.

TN3191: IMAP extensions supported by Mail for iOS, iPadOS, and visionOS

Learn which extensions to the RFC 3501 IMAP protocol are supported by Mail for iOS, iPadOS, and visionOS.

TN3134: Network Extension provider deployment

Explore the platforms, packaging, OS versions, and device configurations for Network Extension provider deployment.

TN3179: Understanding local network privacy

Learn how local network privacy affects your software.

TN3189: Managing Mail background traffic load

Identify iOS Mail background traffic and manage its impact on your IMAP server.

TN3187: Migrating to the UIKit scene-based life cycle

Update your app to receive scene-based life-cycle events and manage your user interface using scene objects and methods.

TN3188: Troubleshooting In-App Purchases availability in the App Store

Verify your In-App Purchases are approved and available for sale in the App Store.

TN3186: Troubleshooting In-App Purchases availability in the sandbox

Identify common configurations that make your In-App Purchases unavailable in the sandbox environment.

TN3185: Troubleshooting In-App Purchases availability in Xcode

Inspect your active StoreKit configuration file for unexpected configurations.

TN3182: Adding privacy tracking keys to your privacy manifest

Declare the tracking domains you use in your app or third-party SDK in a privacy manifest.