

On-Device LLMs on Apple Platforms: llama.cpp vs LiteRT-LM vs Core AI

Matthew Schwartz

June 2026

On-device language models are moving from demo territory into real product architecture. The benefits are obvious: lower latency, better privacy, no per-token cloud bill, and a user experience that can keep working when the network does not.

The hard part is choosing the runtime.

For Apple-platform apps today, three paths are worth comparing:

- **llama.cpp**, the pragmatic open-model runtime with broad GGUF support.
- Google's LiteRT-LM, a promising but still preview-feeling path for mobile LLM inference.
- Apple's Core AI, the new Apple-native stack for running AI models on device.

This article is not a benchmark paper. It is a field guide from the perspective of building real apps that need to ship, survive App Review, support model updates, and behave well under mobile lifecycle pressure.

Scope matters: my production experience here is with **llama.cpp** in an iOS app. LiteRT-LM and Core AI are evaluated as migration candidates, not runtimes I have already shipped in production.

The runtime comparison is useful, but the bigger lesson is architectural: performance without lifecycle stability is not enough. A model that is fast in a benchmark but stalls, blocks the UI, or fails when the user switches apps is not production-ready.

The Short Version

If you need to ship an iOS app with open local models today, **llama.cpp** is still the most practical default. If you need a cross-platform story today, it is also the safest answer.

If you are experimenting with cross-platform mobile ML or want to track Google's direction, LiteRT-LM is worth watching. It looks like the rising option for mobile portability and speed, especially with multi-token prediction (MTP), but preview Swift APIs should be isolated behind an abstraction.

If you are building for the Apple ecosystem long term, Core AI deserves serious attention. Apple is positioning it as the native path for on-device AI models on Apple Silicon, with Swift APIs, model specialization, ahead-of-time compilation, Xcode integration, profiling, and optimization tooling. The tradeoff is clear: Core AI is Apple-specific by design.

My recommendation: do not rewrite a working app blindly. Put an **InferenceEngine** protocol between your UI and runtime, keep shipping with the stable engine, and run measured spikes against the newer stacks.

What Each Platform Gives You

llama.cpp

llama.cpp is the practical workhorse of local LLM apps. Its biggest advantage is model availability. The GGUF ecosystem is broad, fast-moving, and easy to test. If a model appears on Hugging Face, there is a good chance a quantized GGUF build exists quickly.

For product teams, that matters. You can try Gemma, Qwen, Llama, Mistral, Phi, DeepSeek-family models, and many others without waiting for a vendor-specific conversion pipeline.

What it gives you:

- Broad GGUF model support.
- Mature quantization workflows.
- Good performance across CPU, Metal, and other backends.
- Strong portability across Apple platforms, Linux, Windows, Android, and server environments.
- A large community and fast model ecosystem.
- Full control over prompts, sampling, tokenization, model files, and runtime behavior.

What you pay for:

- C/C++ integration inside a Swift app.
- Manual lifecycle handling around model load, unload, cancellation, and backgrounding.
- Manual prompt template selection per model family.
- Manual model catalog and download logic.
- Potential Metal edge cases under mobile app lifecycle pressure.
- Less native Xcode-level introspection than Apple-owned frameworks.

In practice, `llama.cpp` is not “easy”, but it is understandable. You own the runtime boundary. When something fails, you can inspect the engine, logs, model file, prompt, and token stream directly.

That control is why many indie and small-team apps can ship with it.

LiteRT-LM

LiteRT-LM is interesting because Google is clearly investing in mobile inference. The promise is attractive: a runtime aligned with Google’s model and mobile ML ecosystem, with a path toward efficient deployment on constrained devices.

The caution is maturity. I would treat the current Swift path as preview-stage: useful for evaluation, promising for speed, but not something I would make my only production runtime until the APIs, docs, and lifecycle behavior are proven in a real app.

What it is worth evaluating for:

- A Google-backed mobile inference path.
- Alignment with Android and cross-platform model deployment.
- A future route for optimized local LLM inference outside the Apple-only stack.
- A useful comparison point for latency, memory, MTP speedups, and app integration.

What to watch:

- Swift API stability.
- Documentation quality and version churn.
- Model conversion workflow.
- iOS lifecycle behavior.
- Debuggability when inference hangs or crashes.
- Whether the runtime can handle real app patterns, not just demos.

For now, I would place LiteRT-LM behind an abstraction and use it in an experimental app or branch. If it becomes stable, the abstraction lets it graduate into production without forcing a rewrite.

Core AI

Core AI is the strategic Apple-native option.

Apple describes Core AI as a set of technologies purpose-built for Apple Silicon, with a Swift API for loading and running AI models entirely on device. The pitch is strong: private on-device execution, zero server dependency, zero token cost, hardware specialization, ahead-of-time compilation, profiling, debugging, and optimization tools.

Apple’s Core AI material highlights:

- Running models entirely on device.
- Memory-safe Swift APIs.
- Hardware specialization for Apple devices.

- Ahead-of-time compilation to reduce first-load work.
- Fine-grained inference memory control.
- Zero-copy data paths, as described by Apple.
- Stateful execution.
- Model conversion from PyTorch through Core AI PyTorch extensions.
- Model compression through Core AI Optimization.
- Xcode and Core AI Debugger support.
- Instruments templates for runtime profiling.
- A Core AI Models repository with export recipes for popular models.

That is the right direction for Apple-platform apps.

The question is not whether Core AI matters. It does. The question is when it becomes the default choice for your product.

What it gives you:

- A native Apple framework path.
- Better alignment with Apple Silicon.
- Better tool integration with Xcode and Instruments.
- A clearer App Store story than shipping a custom C++ inference stack.
- A path to ahead-of-time compiled model artifacts.
- Potentially better power and memory behavior once models are optimized.
- A model workflow Apple can improve over time.

What to watch:

- iOS 26 or newer deployment requirements.
- Model availability in Core AI format.
- No direct GGUF support; existing GGUF model catalogs require conversion or replacement.
- Conversion complexity.
- First-run model specialization latency.
- Artifact size and background asset strategy.
- Whether your preferred models are supported well.
- API churn while the ecosystem matures.
- Whether Core AI can match your current model quality and sampling controls.

Apple's WWDC 2026 Core AI integration session specifically calls out first-run specialization as something developers need to plan for. In plain terms, the model may need device-side compilation or optimization the first time it is prepared for execution. That can turn first launch or first model load into a visible 30-60 second experience if you do not design around it. Future loads can be fast once cached, and ahead-of-time compilation can reduce the first-run cost, but this still changes your app's onboarding and model-loading design.

That is not a blocker. It is an architectural requirement.

Comparison Table

Area	llama.cpp	LiteRT-LM	Core AI
Best use today	Shipping open-model local LLM apps	Experimental mobile LLM work	Apple-native strategic prototype for iOS 26+
Model ecosystem	Very broad GGUF ecosystem	Emerging/runtime-dependent	No GGUF; curated or converted Core AI assets
Swift integration	Manual bridge around C/C++	Preview Swift path may be unstable	Native Swift API
Apple tooling	Limited native framework tooling	Depends on integration	Xcode, Core AI Debugger, Instruments
Model conversion	Usually download GGUF	Runtime-specific	PyTorch extensions, Core AI models repo
Performance control	High, but manual	Runtime-dependent	Apple Silicon-specialized
Portability	Excellent	Potentially cross-platform	Apple platforms only

Area	llama.cpp	LiteRT-LM	Core AI
Debuggability	Source/runtime visibility, but low-level	Depends on preview tooling	Strong Apple tooling story
Lifecycle risk	You own cancellation/background behavior	Unknown until measured	Apple-native, but still must measure
Vendor lock-in	Low	Medium	High to Apple platforms
Shipping risk today	Moderate but manageable	Higher while Swift APIs and docs mature	High for existing GGUF apps: iOS 26 minimum, no GGUF, model conversion required

The Big Architectural Lesson

The wrong move is to wire your UI directly to a specific inference engine.

The right move is to define an engine boundary:

```
protocol InferenceEngine {
    var state: InferenceState { get }

    func loadModel(at path: String, config: EngineConfiguration) async throws
    func generate(prompt: String, config: EngineConfiguration) -> AsyncThrowingStream<String, Error>
    func cancelGeneration()
    func unloadModel() async
}
```

Then implement:

```
final class LlamaCppEngine: InferenceEngine { ... }
final class CoreAIEngine: InferenceEngine { ... }
final class LiteRTLEngine: InferenceEngine { ... }
```

The rest of the app should not care which runtime is underneath. The UI should know about messages, streaming text, cancellation, model loading, and errors. It should not know about GGUF, `.aimodel`, Metal command buffers, tokenizer quirks, or model specialization.

The exact protocol may need runtime-specific adapters. For example, one engine may stream token-by-token, another may stream chunks, and another may expose callbacks or batched decode results. The app-level contract is still the same: start generation, surface incremental output when available, support cancellation, and report completion or failure.

This boundary also forces you to separate three concerns that often get tangled:

- **Runtime:** How tokens are generated.
- **Prompting:** How messages are formatted for a model family.
- **Model management:** How models are discovered, downloaded, verified, loaded, and updated.

That separation matters because each runtime changes those concerns differently.

Lessons Learned From Real App Integration

Most production problems in on-device LLM apps are not unique to one inference engine. They come from putting long-running, memory-heavy, stateful work inside a mobile app that can be interrupted at any time.

These lessons apply whether the backend is `llama.cpp`, LiteRT-LM, Core AI, or something else.

Keep Inference Off the Main Thread

Token generation must never block the main thread. The UI should be able to scroll, cancel, navigate, and render partial output while inference continues elsewhere.

In Swift terms, keep the engine behind an actor or another serialized async boundary. Let the UI talk to a view model, let the view model talk to a coordinator, and let the coordinator own access to the engine.

The engine may be C++, Swift, Metal-backed, or Apple-native. The rule is the same: main-thread responsiveness is a product requirement, not an optimization.

Serialize Engine Access

Most local inference runtimes do not behave well if model load, unload, generation, cancellation, and context mutation happen concurrently without discipline.

Use a coordinator or state machine that makes invalid transitions impossible:

- `idle` -> `loading` -> `ready`
- `ready` -> `generating`
- `generating` -> `cancelling` -> `ready`
- `failed` -> `idle` or `failed` -> `loading`

This is independent of runtime choice. The exact failure mode changes by engine, but the app-level invariant does not: only one generation should own the model session at a time.

Add a Watchdog

A local model can stall before the first token or stop streaming partway through a response. The UI cannot wait forever.

A practical watchdog has at least two timers:

- **First-token timeout:** cancels generation if the model never starts responding.
- **Token-stall timeout:** cancels generation if streaming stops after output has begun.

The timeout should account for context size and prompt length. A long prompt may need more startup time than a short one, but it still needs a ceiling.

This is engine-independent. The watchdog observes user-visible behavior: no token arrived, or tokens stopped arriving.

Cancellation Must Reach the Runtime

Cancelling a Swift `Task` is not always enough. If the actual work is inside C++, Metal, a native runtime, or a compiled model executor, the app needs a cancellation path that reaches that runtime.

Examples:

- `llama.cpp` needs a native abort callback or equivalent polling hook.
- Core AI or LiteRT-LM should be wrapped so their cancellation semantics are explicit and tested.
- The UI should preserve partial output when cancellation happens after tokens have streamed.

The interface can stay generic:

```
func cancelGeneration()
```

The implementation is engine-specific.

Treat Backgrounding as a First-Class Test

Users will switch apps during generation. The app may be suspended, memory pressure may rise, and GPU or accelerator work may be interrupted.

Test these flows deliberately:

- Background during model load.
- Background before first token.
- Background mid-generation.
- Foreground after cancellation.
- Memory warning while a model is loaded.

- App relaunch after an interrupted session.

Do not assume an engine that works in a foreground demo is production-stable. Mobile lifecycle behavior is part of the runtime evaluation.

Budget Context Before Generation

Do not discover context overflow after the user has already started waiting for an answer.

Preflight the prompt:

- Count or estimate prompt tokens.
- Reserve tokens for the response.
- Keep the system prompt protected.
- Trim old conversation context before building the final prompt.
- Surface context pressure in the UI when it matters.

This logic should live above the engine. Engines differ in tokenizer details, but the product still needs a budget manager.

Stream Carefully

Streaming every token directly into SwiftUI can create unnecessary render pressure. Buffer tokens and update the UI on a short throttle interval.

The user still sees live output, but the app avoids reparsing Markdown and relaying layout work on every tiny token fragment.

A useful pattern is:

- Plain text while streaming.
- Markdown rendering after the assistant message is complete.
- Partial transcript preservation on cancel or timeout.

This is not tied to `llama.cpp`. Any streaming engine can overwhelm the UI if updates are too granular.

Validate Model Files Before Loading

Model management is part of inference reliability.

For downloaded or imported models:

- Verify expected file type or magic bytes.
- Verify checksums for catalog downloads.
- Store models in app-controlled storage.
- Track metadata separately from transient UI state.
- Handle retired or stale catalog entries.

This is especially visible with GGUF, but the same principle applies to Core AI artifacts or any converted model package.

Keep Prompt Templates Out of the Engine

The engine should generate tokens from a prompt. It should not own product-specific chat formatting.

Prompt templates belong in a model-family layer. That lets the same runtime support Gemma, Qwen, Llama-style, or future model formats without changing engine code.

This also makes migration easier. If you move from `llama.cpp` to Core AI but keep the same model family, prompt rendering should not have to move with the runtime.

Log the Boundary, Not Just the Error

When local inference fails, a generic “generation failed” message is not enough for engineering.

Log events around the boundary:

- Model selected.
- Runtime selected.
- Load start and finish.
- Prompt size estimate.
- Generation start.
- First token latency.
- Cancellation requested.
- Timeout reason.
- Engine error.
- Recovery action.

Keep user-facing errors short, but make developer logs specific enough to diagnose lifecycle and runtime issues.

Prompt Templates Are Product Bugs

One easy mistake is assuming all chat models speak the same prompt format.

They do not.

Gemma-style instruct models may expect tags like:

```
<start_of_turn>user
Hello<end_of_turn>
<start_of_turn>model
```

Qwen chat models commonly use ChatML-style tags:

```
<|im_start|>user
Hello<|im_end|>
<|im_start|>assistant
```

If your app supports both model families but hardcodes one template, half your catalog can produce degraded or broken output. This is not a model-quality issue. It is an app bug.

The fix is to make prompt rendering model-aware and test it.

What I Would Measure Before Migrating

Before replacing `llama.cpp` with Core AI or LiteRT-LM, I would run the same product-level tests across all three engines.

Measure:

- Cold model load time.
- First-run specialization or compilation time.
- Warm model load time.
- Time to first token.
- Tokens per second.
- Peak memory.
- App size impact.
- Download size.
- Battery and thermal behavior.
- Backgrounding during generation.
- Cancellation latency.
- Recovery after memory warning.
- Model switch latency.
- Quality on the same prompt set.
- Prompt-template correctness.

- Streaming stability.
- Crash behavior under repeated load/generate/cancel cycles.

Use real prompts, not just benchmark text.

For a chat app, I would include:

- Short factual questions.
- Long-context follow-ups.
- Code generation.
- Markdown formatting.
- Tool-call style JSON.
- Refusal/safety edge cases.
- Multilingual prompts.
- Prompts that hit max-token limits.
- Backgrounding while generating.

Performance without lifecycle stability is not enough. A model that is fast in a benchmark but crashes when the user switches apps is not production-ready.

Migration Strategy

I would not migrate in one step.

I would do this:

1. Keep shipping the current `llama.cpp` path.
2. Extract a clean **InferenceEngine** protocol.
3. Move prompt formatting into model-family templates.
4. Add a runtime selector behind a developer flag.
5. Build a Core AI proof of concept with one supported Qwen model.
6. Measure against the existing `llama.cpp` Qwen path.
7. Try the same app lifecycle tests: load, generate, cancel, background, foreground, memory warning.
8. Decide based on data.

The key is to compare user-visible behavior, not framework marketing.

When I Would Choose Each Runtime

Choose `llama.cpp` when:

- You need broad open-model support now.
- You want GGUF model flexibility.
- You need to ship across Apple and non-Apple platforms.
- You want maximum control over sampling and runtime behavior.
- You can afford low-level engine ownership.

Choose LiteRT-LM when:

- You are experimenting with Google's mobile LLM direction.
- You care about future cross-platform alignment and want to track the strongest emerging alternative to `llama.cpp`.
- You want to evaluate LiteRT-LM's speed profile, especially with MTP.
- You can tolerate preview API churn.
- You have an abstraction layer and can swap it out.

Choose Core AI when:

- You are Apple-platform-first.
- Your target OS versions support it.
- Your model exists or can be converted cleanly.
- You want native Swift integration and Apple tooling.
- You can design around model specialization and deployment artifacts.

- You want to align with Apple’s long-term on-device AI stack.

My Current Position

For a production iOS local-chat app today, I would keep `llama.cpp` as the default runtime and invest in the abstraction layer.

For a research branch, I would prototype Core AI immediately, and I would keep a LiteRT-LM spike active specifically because its speed profile can be compelling, especially when multi-token prediction (MTP) is available.

For cross-platform apps, the current production answer is still `llama.cpp`. LiteRT-LM is the one I would watch most closely as the mobile cross-platform contender on the rise. Core AI is different: it may become the best Apple-platform answer, but it is not a portability strategy.

For a public technical write-up, I would avoid declaring a winner too early. The more useful message is this:

Core AI is the strategic Apple-native path. `llama.cpp` is the pragmatic shipping and cross-platform path today. LiteRT-LM may be the rising speed and portability story, especially with MTP, but preview runtimes should be isolated until they prove stable.

That is not fence-sitting. It is engineering risk management.

The runtime is not the product. The product is the experience: private, responsive, reliable intelligence that behaves well on a real user’s phone.

Pick the engine that lets you deliver that today, and design the system so you can adopt the better engine tomorrow.

References

- ggml-org: [llama.cpp](#)
- Google AI Edge: [LiteRT-LM](#)
- Apple Developer: [AI & Machine Learning](#)
- Apple Developer: [Core AI](#)
- Apple WWDC 2026: [Integrate on-device AI models into your app using Core AI](#)