Building a Voice App with AI: What I Actually Learned

The plain-English version

The Itch

I was using Wispr Flow. It is a voice dictation app for Mac -- you talk, it types. It works across all your apps, it auto-formats, it learns your vocabulary, and it supports over 100 languages. It costs $12-15 a month.

It is a good product. But I was paying a subscription for something where I only used a fraction of the features. I did not need 100 languages. I did not need AI-powered reformatting or tone adjustment. I did not need it to work in Slack, Notion, and 40 other apps simultaneously. I did not need team collaboration or enterprise compliance.

What I actually needed was this: press a keyboard shortcut, say something, and have the text appear on my clipboard. That is it. One shortcut, one action, text on clipboard. Optionally, paste it into whatever app I was in. And I wanted it to run entirely on my Mac -- no audio sent to anyone's servers, no cloud processing, no data leaving my machine.

Wispr Flow is built for a wide audience with broad needs. I had a narrow need and a preference for privacy. The question was whether I could build exactly what I wanted using AI to write the code.

I built it using Claude Code (Anthropic's AI coding assistant) and a workflow system called GSD that structures how the AI works. I do not write the code myself -- the AI does -- but the decisions, the debugging, and the learning were very much mine.

Here is what I learned along the way.

Wave 1: The Idea That Died in Research

My first plan was to piggyback on Voice Memos -- the recording app already on every Mac. Press a shortcut, tell Voice Memos to start recording, then grab the transcription it produces. Why reinvent the wheel when Apple has already built one?

Before I wrote a single line of code, I asked the AI to research whether this was even possible. It sent out four separate research agents in parallel, each investigating a different angle. Within minutes, three deal-breakers came back:

- Voice Memos cannot be controlled by other apps. Apple never built that capability in.
- Reading Voice Memos' stored data requires a permission so aggressive that most users would (rightly) refuse to grant it.
- Even with that permission, two apps reading the same data file at the same time causes conflicts.

The entire idea was dead before it started. The "simple" approach of stitching together existing Apple tools was not simple at all.

The learning: Research that kills a bad idea saves more time than research that confirms a good one. If I had started building the Voice Memos approach, I would have discovered these problems days later and thrown everything away. Instead, I pivoted to building the recording and transcription directly into my own app -- a simpler, more reliable approach -- before any code existed.

The AI learning: The AI works best when you send multiple investigators at a problem simultaneously. One agent looked at what technologies to use, another at how to structure the code, another at what features competitors have, and a fourth whose entire job was to find reasons things would fail. It was that fourth agent -- the pessimist -- that saved the project.

Wave 2: The Power of Writing Down What You Won't Do

After the pivot, I defined exactly what version 1 needed to do. Eighteen specific requirements, grouped into categories: recording, menu bar display, transcription, feedback, and error handling.

This is where the Wispr Flow comparison became a useful filter. Wispr Flow has dozens of features. I deliberately chose the handful I actually use and excluded the rest:

What I kept (the bits I actually use from Wispr Flow): - Global keyboard shortcut to start and stop recording - Voice-to-text transcription - Text on clipboard automatically - Auto-paste into the active app - Visual status indicator

What I left out (features Wispr Flow has that I do not need): - AI-powered reformatting and tone adjustment - Command mode for editing text by voice - Multi-language auto-detection and switching - Cross-platform sync (Windows, iPhone) - Team collaboration and shared context - Cloud processing of any kind - Snippet libraries and templates - App-specific formatting (code syntax in VS Code, etc.)

What I added (things Wispr Flow does not do): - Entirely on-device processing -- no audio leaves my Mac, ever - Text replacements for recurring transcription quirks ("dot dot dot" becomes "...") - Choice of voice recognition model (trade speed for accuracy) - No subscription, no account, no word limits

Writing down the exclusions was as important as writing down the inclusions. In practice, "maybe I should add AI reformatting like Wispr Flow has" is exactly the kind of thought that turns a weekend project into an abandoned project. Every nice-to-have idea is a potential rabbit hole. Having the explicit list of "no, not this, not yet" stopped me from scope-creeping towards rebuilding the full product I was trying to replace.

I also wrote down version 2 ideas (live preview, context awareness, AI formatting, history) and explicitly labelled them "later." This is different from ignoring them. It acknowledges they matter while keeping them out of the current scope.

The learning: Scope discipline is not about saying no. It is about saying "not yet" and writing it down so you stop re-debating the same decision every session. Having a reference product like Wispr Flow makes this easier -- you can point at specific features and say "that one yes, that one no" rather than imagining requirements from scratch.

The AI learning: The written requirements became a contract between me and the AI. Every requirement was assigned to a specific phase. When the AI planned each phase, a separate checker agent verified the plan against those requirements. The AI could not quietly drop a feature because there was a paper trail holding it accountable.

Wave 3: 20 Minutes to a Working App

Here is the number that surprises people: the actual coding took about 20 minutes.

The planning took 2.5 hours. Research, requirements, architecture decisions, risk analysis, detailed plans for each phase. But once all of that was done, the AI translated those decisions into a working app across seven sequential plans in roughly 20 minutes.

This is not because the code was simple. There are real engineering decisions in there -- how the app tracks its own state, how audio flows through the system, how the clipboard is verified, how permissions are checked. But every single decision had already been made during planning. The AI was not thinking during execution. It was typing.

The speed also increased as the project progressed:

- Phase 1 (foundations): 5.5 minutes per plan
- Phase 2 (recording): 3.0 minutes per plan
- Phase 3 (error handling): 1.3 minutes per plan

The first phase establishes patterns. Later phases follow them. Once the AI knows how your project is structured, it gets faster.

At this point I had a working app that did the core thing Wispr Flow does for me -- keyboard shortcut, voice recording, text on clipboard -- in about three hours of total effort. No subscription. No cloud. No data leaving my machine.

But "working" and "reliable" are different things. That distinction would take another two weeks to close.

The learning: Front-loading decisions into planning is not overhead. It is what makes execution fast. The 2.5 hours of planning were not in addition to the 20 minutes of building -- they were the reason the building only took 20 minutes.

The AI learning: GSD gives each coding task a completely fresh AI session. This sounds like a problem -- doesn't the AI lose track of what it was doing? -- but it is actually the key advantage. Each task starts with a clean slate, the plan, and the relevant code. No accumulated confusion. No degraded attention from a conversation that has gone on too long. Clean desk, clear instructions. The seventh task is as sharp as the first.

Wave 4: The Bug That Reported Success While Failing

The first real bug was invisible. Recording worked perfectly the first time. The second time, it silently failed. The microphone system reported that everything was running fine. But no audio was being captured. No error, no crash, no warning.

This is the kind of bug that would never survive in a product like Wispr Flow -- their team would catch it in testing. But as a solo builder, I only had my own usage to surface it. And it only appeared on the second recording, which made it easy to miss initially.

The fix was counterintuitive: instead of reusing the microphone system between recordings, throw it away completely after each one and create a fresh one next time. This takes a fraction of a second and completely eliminates the problem.

Apple's documentation suggests the system is designed to be started and stopped repeatedly. In practice, it corrupts its own internal state after a few cycles.

The learning: Silent failure is the most dangerous kind of bug. The system says everything is fine while doing nothing. The only way to catch it is to check the actual output ("did audio actually arrive?") rather than trusting the system's self-reported status. This principle applies far beyond software.

The AI learning: This bug was not in the AI's training data. No documentation, no forum post, no blog describes this specific failure. It was discovered by describing the symptoms ("works once, fails silently on retry") and letting the AI investigate systematically. The AI is good at structured debugging when you tell it what you observed rather than what you think the cause is.

Wave 5: Order Matters More Than You Think

When the recording stops, two things need to happen: the transcription system needs to produce its final text, and the microphone needs to shut down. The order is

critical.

If you shut down the microphone first, the transcription system either waits forever (because it is expecting more audio that will never come) or panics and returns whatever it has so far (usually a truncated fragment).

The correct order: tell the transcription system "no more audio is coming" first, let it finish its work, then shut down the microphone. Consumer before producer.

Apple's documentation does not mention this. The two systems are presented as independent. But they have an unwritten contract.

The learning: When two systems share a resource, the shutdown order is almost always: the reader stops before the writer. This pattern appears everywhere -- not just in audio, but in databases, networks, messaging. Signal the consumer, then close the producer.

The AI learning: The AI figured this out from general engineering knowledge, not from Apple-specific documentation. Understanding patterns is more useful than knowing specific APIs.

Wave 6: Choosing My Own Engine

The initial version used Apple's built-in speech recognition. It worked, but it was a black box -- Apple chooses the model, Apple decides the accuracy, and the newer, better version requires a future operating system I cannot run yet.

I switched to WhisperKit, an open-source engine built by a company called Argmax that runs OpenAI's Whisper voice models entirely on the Mac's own hardware. This was a deliberate choice to get something Wispr Flow does not offer: complete transparency and control over the voice recognition.

With WhisperKit, I can choose between five different model sizes. A small model (~40MB) gives fast, rough results. A large model (~3GB) gives slower, highly accurate results. I picked the middle ground as the default -- about 150MB, processes in 1-2 seconds, good enough accuracy for everyday use. The user can switch at any time.

But switching engines revealed a new problem. The first transcription after launching the app took 30 to 60 seconds. Every subsequent transcription was instant.

The cause: the Mac's neural processing chip needs to compile the AI model before it can use it. This compilation only happens the first time the model actually runs. So the first real use triggers a massive one-time cost.

The fix: after downloading the model, run a fake transcription on a second of silence. The user sees a "loading model" indicator during this warm-up. By the time they press the shortcut for real, the slow compilation has already happened.

The learning: Sometimes you need to force expensive setup to happen during a moment when the user expects to wait (loading screen) rather than deferring it to a moment when they expect instant response (first use). "Lazy" is not always smart.

The AI learning: I upgraded from Claude Opus 4.5 to 4.6 between the initial build and this feature work. The transition was seamless because all the decisions, patterns, and accumulated knowledge were written in files, not stored in any single conversation. The AI is replaceable. The documented decisions are not.

Wave 7: Simulating a Keyboard Press is Surprisingly Hard

This is where I replicated one of Wispr Flow's most useful features: auto-paste. Instead of just putting text on the clipboard and making you press Cmd+V yourself, the app simulates the paste keystroke for you. You speak, you stop, the text appears in whatever app you were using.

Conceptually simple: simulate pressing Cmd+V. In practice, three separate timing problems had to be solved:

Problem 1: The 50-millisecond gap. When simulating a key press, there is a "key down" and a "key up" event. If they happen too close together, many apps simply ignore it. A 50ms gap between them -- completely imperceptible to humans -- is enough for apps to register the keystroke.

Problem 2: The half-second delay. After the menu bar interaction finishes, the user's original app needs time to regain focus. Without a 500ms pause before pasting, the paste goes to the wrong app (or nowhere).

Problem 3: The identity crisis. macOS grants the "paste into other apps" permission based on the app's digital signature. Every time I rebuilt the app during development, the signature changed, and macOS revoked the permission. Worse, the System Settings screen still showed the app as "trusted" even when it was not -- the interface lied about the actual permission state.

The fix for the signature problem: create a permanent signing certificate that stays the same across rebuilds, so the app's identity remains stable.

Products like Wispr Flow handle all of this invisibly because they ship a single signed build through the App Store. When you are building your own tool and rebuilding it constantly during development, these problems are front and centre.

The learning: macOS permissions are based on cryptographic identity, not on the app's name or location. This is invisible until it breaks, and when it breaks, the system actively misleads you about the state of affairs.

The AI learning: These three problems were discovered across separate sessions over multiple days. Each session found one piece of the puzzle. What made this work was persistent memory -- each discovery was saved and available in the next session. Without that, every new conversation would have started from scratch, and I would have re-discovered the same issues repeatedly.

Wave 8: The Safety Net with a Hole In It

The app has a 30-second timeout: if transcription takes too long, it gives up and tells the user. Or at least, that was the intent.

The original implementation set up two parallel tasks: one doing the transcription, one counting to 30 seconds. The idea was "whichever finishes first wins." But the underlying mechanism did not work that way. It actually waited for BOTH tasks to finish before reporting a result. So when transcription took 60 seconds, the timeout fired at 30 seconds... and then sat there waiting another 30 seconds for the transcription to finish before telling anyone.

The timeout never actually timed out. It was decoration.

The fix used a completely different approach: two fully independent tasks with a shared flag. Whichever one finishes first sets the flag and claims the result. The other one sees the flag is set and quietly stops.

The learning: A safety mechanism that appears to work but does not is worse than having no safety mechanism at all. It creates false confidence. The timeout looked right in the code, compiled without errors, and worked perfectly under normal conditions. It only failed in the exact edge case it was designed to handle.

The AI learning: The AI wrote the original broken version. It looked correct. It compiled. It ran. It passed normal testing. The bug only appeared under specific conditions that rarely occur. This is an important reminder: AI-generated code needs the same scrutiny as code written by a person. "It works in testing" is not the same as "it is correct."

Wave 9: When the Transcription Engine Has Its Own Ideas

This wave is about a feature Wispr Flow does not have -- and did not need, because Wispr Flow's AI reformatting handles it at a higher level.

Voice recognition engines do not just transcribe your words. They add their own punctuation, capitalisation, and spacing. Most of the time this is helpful. But when you are dictating something specific -- a phrase you want formatted a certain way, or a verbal shortcut for a symbol or line break -- the engine's "helpful" formatting gets in the way.

Wispr Flow solves this class of problem with AI-powered reformatting -- the entire output is rewritten by a language model that understands what you meant. That is powerful but requires cloud processing and is part of what you pay the subscription for.

I took a simpler, local approach: text replacements. A list of find-and-replace rules that run after transcription but before the text hits the clipboard. You define verbal shortcuts -- say a phrase, get specific text. "New paragraph" becomes a line break, and so on.

But a simple find-and-replace does not work because the engine inserts its own punctuation between your words. If you say a multi-word trigger phrase, the transcription might contain commas, periods, or capitalisation that your replacement rule does not expect. The solution: flexible matching that tolerates any punctuation or spacing between the words in your trigger phrase.

That created a second problem: replacement rules that insert punctuation (like a period) can collide with punctuation the engine already added, producing doubles. The system needed three different categories of replacement rule, each handling punctuation differently, plus a cleanup pass at the end.

It is not as elegant as Wispr Flow's AI reformatting. But it runs entirely on my machine, adds zero latency, and handles the specific corrections I need. For a tool I built for myself, that is enough.

The learning: When you build on top of another system, you need to handle that system's quirks. The engine adding punctuation between words is not a bug -- it is correct behaviour for general transcription. The replacement system is a translation layer between the engine's output and the user's intent.

The AI learning: The three-category solution emerged from collaboration. I described the symptom ("replacements create double punctuation") and the AI designed the categorisation. Neither of us would have arrived at the approach alone. I would not have thought of it. The AI would not have known it was needed without hearing about the specific failure. The best results came from describing what went wrong and letting the AI architect the fix.

Wave 10: What It All Adds Up To

Looking at the list of fixes, none of them is individually dramatic:

- Use a fresh microphone system each time (prevents silent failures)
- Shut down in the right order (prevents truncated results)
- Warm up the AI engine during loading (prevents surprise delays)
- Wait 50ms between key events (prevents dropped pastes)
- Wait 500ms before pasting (prevents pasting into the wrong app)
- Use a stable signing certificate (prevents permission loss)
- Use a proper timeout mechanism (prevents infinite hangs)
- Use flexible text matching (prevents garbled replacements)
- Verify clipboard writes (prevents silent data loss)

Remove any one of these and the app still works most of the time. But "most of the time" is not good enough for something you use dozens of times a day. Each fix addresses a failure that happens somewhere between 1 in 10 and 1 in 100 uses. Together, they are the difference between a demo and a tool.

This is what $12 a month buys you with Wispr Flow -- a team that has already found and fixed all these edge cases. Building your own means finding them yourself, one

at a time, through real use. It took about three weeks from first idea to stable daily-use tool.

Was it worth it? For me, yes -- but not primarily for the money saved. I wanted something that runs entirely on my machine with no data leaving it. I wanted to understand how voice-to-text actually works at every level. And I wanted to learn how to build real software with AI.

I got all three.

The Meta-Learning

About building: Reliability is not a feature you add. It is the compound effect of eliminating every failure mode you find, one at a time, across multiple sessions. No single session produces a reliable app. Reliability accumulates.

About choosing what to build vs buy: Having a reference product (Wispr Flow) made scope decisions concrete. Instead of imagining features in the abstract, I could point at something real and say "that one yes, that one no." The features I kept are the ones I use daily. The features I left out are the ones I was paying for but never touching. The features I added (fully local processing, text replacements, model choice) are the ones the subscription product cannot offer because of its architecture.

About using AI to build: Three things made AI-assisted development work for this project:

1. **Plan before you code.** 2.5 hours of planning produced 20 minutes of implementation. The AI is dramatically better at turning decisions into code than it is at making decisions while coding. Do the thinking yourself. Let the AI do the typing.
2. **Give each task a fresh start.** The GSD system gives each coding task a brand new AI session with a clean slate. This prevents the quality problems that creep in during long conversations. The seventh task is as good as the first.
3. **Write everything down.** Every discovery, every decision, every debugging insight was written to files that persist across sessions. The 50ms timing, the signing problem, the timeout bug -- all found in different sessions, all available in every future session. Without this, each new conversation starts from zero and you rediscover the same problems.

The AI model itself upgraded partway through the project (from Opus 4.5 to 4.6). The transition was seamless because the knowledge lived in the files, not in any single conversation. The AI is interchangeable. The documented decisions are not.

Written 22 February 2026