There are four principles that separate people who successfully build AI systems from people who get stuck and then just give up. And I didn't learn these from a textbook, and I'm also not making them up. I learned them last week from watching dozens of people build the same system in completely different ways. So last week,

I posted a video about building a second brain without any code at all, and it got a lot of traction. But what happened next was much more interesting than the video itself, because it taught me how building actually works in 2026. People took the broad strokes, the architecture I described,

and they implemented it in lots and lots of different tools, including tools I'd never recommended. So that meant they hit lots of interesting walls. They got around those walls interesting ways. And they ended up using AI not just inside the system, as I suggested, but also to build the system. So today,

I want to dig into that story and I want to specifically call out four building principles that emerged from watching the community go through dozens and dozens of builds over the last week. And here's the frame I want you to hold on through all of this. The second brain, that's just a project.

What we learned building it is about how complex AI systems get constructed today in a world where you have AI as a collaborator and peer building communities end up functioning as pattern libraries. So let's start with the principles. First, architecture is portable. tools are not.

So when I started with this project, when I laid out the video, I mentioned a specific stack. I talked about Notion for storage and Zapier for automation and Cloud or ChatGPT for intelligence. And I mentioned the different jobs that these tools do. I talked about how you have to have something that's kind of a drop box,

something that sort of lets you put things away, something that sorts, something that acts to structure data. And I went through the different principles. Now, people were able to take those principles and build on them with any kind of tool. The tools could be very, very interchangeable.

And the architecture itself stayed very stable across all of those tools. So one community member built their system using Discord, Obsidian, and MacWhisperer. So MacWhisper is a tool that transcribes audio locally on your Mac, and Discord is really the capture point that replaces Slack.

And they set up a Discord server with like a special structure and added timed prompts. They really sort of built it out. So they used very different tools, but they came from the same principles. You can see the capture point. You can see the sorting that happens. You can see the intelligence layer.

The implementation details may be unrecognizable. Like they connected Mac Whisper to automatically process their Zoom recordings, send transcripts to Obsidian, and then they run a slash command that processes and files everything by project. Super cool. Definitely not something in the original build plan, but something that follows the arc,

the structure where you're actually trying to process from a capture point and store it and then make sure that you can retrieve it in an intelligent way later. And this just underlines how important it is to think about architecture consistently in the age of AI. You can end up in a place, if you have a good architecture,

where you have a completely different tool stack. I did not suggest using Mac Whisper and Obsidian, but it still works because the architecture holds. And so the implication here is that when you're learning to build with AI, don't necessarily memorize the tools. Learn the patterns. The tools will shift. Zapier might not be the right choice for you.

Maybe Notion might not fit your workflow. But the patterns, the idea of how the second brain is constructed, that it needs a place to drop ideas, it's clean, it needs a way to sort ideas, et cetera, those are sticky patterns. Those are steady. Once you understand them, you can implement them anywhere.

The second of the four principles that I want to call out is that when you're working with AI, principles-based guidance scales way better than rules-based guidance. And this goes for whether you're coaching or whether you're teaching or frankly, whether you're trying to learn for yourself. So one community member ended up building their system using Claude's computer use

capability combined with Obsidian and a custom TypeScript agent. And so they had a custom agent running on a VPS, which is just a virtual server in the cloud, and they had remote access so they can connect to it from anywhere. And then here's the part that caught my attention.

They wrote an architectural best practices doc that guides the coding agent through fixes when things go wrong. But instead of hard coding specific rules, they were smart. And so they said, let's write principles. Let's write use test-driven development or use dependency injection. Don't swallow errors. They essentially tried to give the agent smart software development principles so

the agent could interpret those principles rather than following rigid rules. And this matters because AI systems are very good at applying judgment in context. They're not just pattern matchers executing if-then rules. We know that, right? When you give AI a principle like don't swallow errors,

it can figure out what that means in a hundred different situations that you did not anticipate. And when you give it a rigid rules, like always log errors to this specific file, you're kind of limiting it to do only that one thing. Which sometimes you need to do, but if you're trying to build with AI,

it's often more useful to go with the principle-based approach. And so the implication for building is really interesting. When you're writing prompts to build, you're effectively creating guidelines for AI systems. And when you're designing workflows that AI will execute, it often makes sense to have AI lean toward principles that scale rather than just writing rules,

unless you have a very, very deterministic workflow you're trying to encode. So give the AI room to exercise judgment where that makes sense. You'll get more robust systems that handle edge cases you didn't think of. I also want to call out a meta layer that is really interesting.

This build required writing principles for an agent to build a second brain system that itself used AI. There's a meta build pattern here that I've seen over and over again in successful 2026 builds. People who are leveraging AI to build their AI tooling are going much, much, much faster. And it's another example of how principles can scale.

Because if you can write an agent that builds your second brain for you and your agent relies on good software design principles, like don't swallow errors, use test-driven development, separate your concerns, you're going to be able to reuse that build agent other places. Principles-based guidance scales in lots of different ways. It's a fractal principle.

It helps us remember things better. It helps us build directly better. It also helps us build agents that can build more than one thing, like a second brain and then something else. So that's the second principle. The third is that if the agent builds it, the agent can maintain it.

One of the most ambitious community builds came from someone who created what they called a meta agent framework. Instead of wiring up individual tools, they built an agent that coordinates between Cloud Code, Codex, Copilot, and Goose, which are all different AI coding assistants. They implemented what's called a writer-critic loop for reliability,

where one agent produces output and another agent validates it catching errors before they propagate. The agent can then set up a cloud infrastructure automatically. It can generate user interfaces, This is much more than just a second break. And the insight that really stuck with me is that they said their goal was to

enable the agent to set up the infrastructure rather than doing it themselves. Because if the agent sets it up, the agent understands it, which means the agent can self-correct and self-heal long after you've forgotten how the system works. This ends up being more like a second brain for an engineer who's coding a lot than

like a traditional second brain build. But by having the agent construct it, you have agent maintainability, which is going to be, I think, a big theme in 2026. So think about what this means. If I build something myself, I understand it when I am building it.

But six months later, I have to pick up a lot of context to get back into it. Anyone who has worked with engineers or built something themselves, there's something called switching cost. You have to get back into the mindset and understand the code base before you can make meaningful changes.

It is non-trivial time and it doesn't look productive because you're just trying to recover your memory. The configuration details, the edge cases you handled, the reasons you made certain choices, all of that fades away. When an agent builds something with you and you keep that conversation context, you keep the artifacts that you created together,

the agent can come back and return to debug and extend and maintain the system. And if you've prepared the memory environment correctly, you can re-instantiate the agent. You can re-invoke the agent with all of the memories associated with that particular build, and it's fresh. There's no context switching.

The documentation ends up being the build process itself, and you can reinvigorate that. You can bring it back to life like you're bringing back a vampire to life in an old B movie, and the agent will just continue to go and do its work. The implication is that when you're building with AI,

Consider involving the AI in the construction process, not just the execution. And I would go farther than that. I would say you're not just really considering in 2026. You should be defaulting to trying to involve AI in the construction process. Let it help you set things up. Keep the conversation going.

When something breaks in six months, you can come back to that conversation. The AI will help you pick up the context you've lost. Even if you're a non-coder, this has never been easier, because Claude launched Cowork this past week, and Cowork can hook into the Claude extension in Chrome, and you can just ask Claude Cowork, please work with my Chrome instance, and it will use your browser and build stuff. People were doing that as well as part of their build process for this second brain in the Substack chat. The fourth principle is that your system can be infrastructure, not just a tool.

Now, most people who built the second brain did treat it as a personal productivity tool. That was the original intent after all. You capture thoughts, you file them, you get digests, et cetera. But at least one community member saw something a bit different. They built a hybrid database approach using Postgres for structured data combined

with a vector database for semantic search. So a vector DB lets you search for meaning rather than exact keywords. So you can ask something like, what did I capture about customer onboarding? And you find relevant entries, even if they don't contain any of the exact words like customer onboarding.

Then they built an API endpoint that lets other applications query their second brain, which is super cool. And they're working toward a reusable software development kit or SDK, basically a toolkit that other developers can use so they can connect any application to their personal knowledge base.

This is someone who saw that the second brain is not really just a personal productivity tool, but it can be a piece of infrastructure that they can build other tools and systems on top of. I think that this shows the kind of higher level systems thinking that we need a

lot more of to build compounding advantage in 2026. There's another piece of the story from a different build that illustrates the same principle. Another builder went deep with Qdrent for vector search, which sounds really technical. It's a bit of a technical build here, right? They used Neo4j for graphing relationships and Postgres as the central system of records.

If that sounds complicated, it's not an easy build. It's definitely intermediate to advanced. They have multiple cloud agents coordinating with a layer they call a skills plus evidence layer where generated content comes with built-in receipts showing what sources informed each output across this entire system. This is definitely not a non-technical build. It's a technical build.

It's an engineering build. But it demonstrates that you can take the same architectural patterns and they will scale from simple to very sophisticated and the principles will not break. The other thing I will call out is the engineering skills that are added here enable you to do and realize larger pieces of work.

And so if you want to think about the second brain or any software project you're contemplating in a broader lens, if you want to say this is not just a personal productivity tool, this is a piece of infrastructure, I want to do something bigger with it. The more you're able to scale your technical skills,

the farther you're going to be able to take that vision. And this is one of the things I keep emphasizing that is counterintuitive. Technical skills have tremendous value in 2026. Don't listen to people who say that engineering is dead. Engineers over the last week have been able to do much,

much more interesting things with their projects because they can use AI to go farther because they have the technical domain knowledge to know where to push AI to help them build. Now, if you're trying to learn technical skills, it's never been easier. You can learn them with AI more easily than you ever could previously.

You can get custom AI coding instructions delivered to you every morning in ChatGPT if you set a scheduled task. I do that every morning. It's super easy. And there's lots of other ways you can do it. There's lots of demos you can be a part of.

You're not short of ways to learn technical skills if you want to. And they're worth it. But zooming back to the broader implication, if you are building something, you should be asking yourself whether it's a tool or infrastructure. A tool is going to solve a problem.

Infrastructure enables others to build on top of the solution that you've constructed. So if you design with infrastructure in mind, you create much, much more leverage than if you're just building a personal productivity tool. And I think that that kind of systems thinking is going to be more and more

valuable in 2026 as we're looking to hit another layer of abstraction as humans managing systems of agents. Like your second brain can power a daily digest and that's super helpful. But if you are using it to power the rest of your entire workflow and system, your meeting prep, your email drafts, your project planning,

if you're using it to power an entire sort of chief of staff operation workflow, well, that initial investment pays much more dividends. And yes, you're going to have to have more technical skills to go after it, but it'll be worth it. And if you're wondering if you can get more details on these builds,

I absolutely wrote up a summary of community builds that I'm going to be sticking in the Substack as well so that you can kind of dive in and understand a bit more. Now, let me add one more build that illustrates something different. One community member went super minimalist. They're using Notion's mobile export feature as their capture mechanism,

so they're not using Slack. They capture thoughts on their phone, export to Notion, and Cloud just processes their inbox on schedule and sorts everything into an outbox. The whole system is essentially just an inbox outbox with AI processing in between. No Zapier, no automation chain, just Notion and a scheduled Cloud call.

You can do another version of Simple with YAML files, which are just declarative files that developers like to use to talk about business rules. Another builder swapped out Notion entirely for local YAML files. YAML is just a simple text format for storing structured data,

and they kept Slack as the capture layer because they're in Slack all day anyway, and it's frictionless from their phone. But instead of using Zapier as the automation layer, they use Claude code, which lets you have a conversation with Claude while it reads and writes files to the computer.

The key difference is that their system is session-based rather than always on, which is what they preferred. And so when you sit down to work, you spin up a Cloud Code session that processes your inbox, runs the classification, writes to local YAML files, and then gives you a fix button in that same conversation.

Look, the point is not that you copy these builds, right? These different builds illustrate that you don't necessarily need complexity to get the benefits. You might be a non-engineer and say, oh, it's simpler to use Notion in out or an engineer and say, I just want to do YAML files.

But regardless, you can start very simple and the principles still work. The architecture supports something that simple as well as something as complex as some of the builds I discussed earlier in the video. Now, here's the pattern that emerged that I think represents something genuinely new about how building works in 2026.

And I want to spend some time on this because I think it's the most important thing I learned from watching the community. The people who got their systems working fastest were not the ones who followed my tutorial the most carefully. Instead, they were the ones who combined community knowledge with AI collaboration. They would hit an obstacle.

They would post in the Substack chat. They would get a pointer from someone who'd solved a similar problem and then work with Claude or ChatGPT to implement the fix in their own context. One community member posted that she built the entire system in two hours by just having Claude do it in the browser for her.

And she gave the architecture document to Claude and said, just solve it. She'd already read through the community discussion. She had some idea of what to do. And she just said, go do it. This is the new model. Community ends up providing a pattern library for us to understand where common obstacles emerge.

And AI ends up giving us implementation muscles so we can do other things while builds happen. Our job is to provide the context and the judgment about what applies in our specific situation. And our job is, frankly, to provide the intent to say, this is how I want to express these larger principles or this larger architecture

in a way that works for me. I want to define what works for me as a second brain, not what works for you. Everyone's individual and different. That's part of what made this such a cool project is that you get to see the wide variety of different solutions people implement.

this is why sas tools for second brains are so hard to build everyone's brain is different and the beauty of architecture is you can build your own second brain the way you like it the tutorial alone will not adapt to all of the tools that are your

favorites and it doesn't need to and frankly even the ai alone will not know what other problems people have already solved the community is going to be critical for you to be able to be successful building long term. And I don't mean my community in particular. The Substack community is wonderful.

But you find your build community that works for you. Having a community that has a pattern library is super useful because it can supercharge your ability to acquire context and strategically allocate that context with your AI to build faster. And that's what I saw this week. Now,

let me give you a quick tour of what the Zapier automation actually looks like when it's built. I got lots of requests for this, so I'm excited to show this to you. So the flow starts with a Slack trigger. When a new message is posted to your SB Inbox channel, Zapier captures it.

And then the trigger gives you access to the message text, the channel ID, all the details. The message text then goes directly to Claude via an API call. You can see that right here. The Anthropic action in Zapier sends you a classification prompt along with the raw message.

And the prompt is going to specify the JSON schema and tell Claude to classify into one of four categories and ask for confidence score, which I talked about in the last video. After Claude responds, a code by Zapier, right here in step three, cleans up the response. This step removes any markdown formatting, it parses the JSON,

and it flattens the nested structure you get from the LLM into a field that's super easy to reference in later steps. If Claude is going to return something that's malformed or incorrect, this code step can handle it gracefully. Next, we have to route. So split into paths.

A path step will check the destination field from the parsed response. If it's people, the flow goes down the people path. You get the idea. Each path does a few things. First, it creates a record. It creates a record in the appropriate Notion database based on the routing that you've done in the previous split.

So now we're in the people database and it can split and create that Notion record. Then it creates an entry in the inbox log with the original text where it was filed, the confidence score and the Slack thread. This is the receipt I talked a lot about in the original video.

Finally, it sent a Slack message back to let you know that it processed all of that. And so a Slack message might look like filed as people, Sarah Chen, confidence 0.87 reply fix if I got it wrong. Now, that's how it works for people and for projects and for ideas and for admin.

The needs review path is a little bit different. Instead of filing to a destination database, it logs the item with a needs review status over here and sends a Slack reply that asks for clarification. It says something like, I couldn't confidently classify this. Can you repost it with a prefix like person or project?

That's really the complete flow. You have one trigger at the top, up here, one AI call, and now you're starting to parse it, and then you're starting to run different parallel paths. You can do this other places. I've also seen this built out in N8N. I've seen it built out in Make.

Again, the principle is what matters, not the tool. Just as a bonus here, I want to share a couple of things that emerge from the community that are not so much relevant to the larger all AI projects in 2026 conversation, but are super relevant to how we think about these tools in the second brain work project specifically.

First, Zapier has limitations that become apparent at scale. It was designed as an easy entry point. It does that well, but it has trouble handling larger loops. It doesn't branch and reconverge for more complex workflows. And so it doesn't work if you're trying to do more advanced work. Second, the storage backend matters less than you might think.

A lot of people are using Notion, they're using Obsidian, they're using YAML files. I laid out some of that. If you have an existing system that has a storage bucket, that's probably good enough. Third, if you're building a second brain, capture interfaces really should just be where you already live.

If you're in Slack, then do it in Slack. Discord works. Google Chat works. Frankly, you can probably rig this up with text messages. The fourth one that's a lesson for second brain builders is there's an emerging split between sessions based and always on approaches. So the Zapier flow I showed is designed to be always on.

That's the way my brain works anyway. Messages get processed automatically, whether you're engaged with them or not. The cloud code approach I talked about earlier is session based. You process your inbox when you sit down to work in conversation with the AI. It doesn't pre-process for you. Both patterns are fine.

Always on is better if you're really wanting to offload the mental load. Session-based gives you much more control and the ability to ask questions during processing and shape that. Again, this is why second brain, one size fits all solutions don't work. You have to decide what works for you. Finally, and this is where things get interesting,

there's movement toward AI agent generated interfaces. Instead of building a fixed dashboard in Notion and hoping it will serve your needs forever, Some builders are generating UI on demand. I think we're going to see more of this in 2026. And so when they need a particular view of the data, they ask Claude to just create it.

I found myself doing this on a different application in Claude Cowork just this week. So I can see that this is going to be a pattern that starts to emerge. Let me close with a reflection on how this all reshapes my own thinking on second

brain building and how AI is shaping what we as a species build and do in the coming year. The second brain was just the project, but the way we built it revealed that we are changing build patterns we've held steady for a long, long time. As I shared in my previous video,

we've had the same cognitive architecture for about half a million years. We've had a few items in working memory, terrible retrieval, etc. The second brain was designed to really help with that. What's different in 2026, what I learned as I started to build with the community,

is that the combination of community plus AI is really changing the economics of building. Community creates a shared pattern library that evolves in real time. Community helps us figure out a better way to handle a confidence threshold, to get around an obstacle, and everybody benefits quickly.

And AI is what provides the implementation muscle to adapt all of these community patterns to fit our particular context. The gap between understand what someone else did and I can do the equivalent used to be where a lot of projects went to die, and now AI can help us bridge that gap.

You no longer need to be an engineer to construct sophisticated systems. And I'm beginning to think the more important thing is not to be an engineer, but to have a community and to have access to an AI you're willing to push. Because you can acquire the technical skills you need along. This community proved that principles are portable.

People can build in any tool they want. You can build in Discord, in Obsidian, wherever you are. That doesn't matter. What matters is that you are able to understand what works for you, how you want to scale build principles, and how you want to use AI to build more quickly.

And that, to me, feels like a really big extension from the original idea I had in the video. Because in the video, I was like, the second brain itself, the tool itself, is a big deal. And that's still true. The build pattern we're seeing is an even bigger deal because it means that we can

take this practice of building in community, using AI to leverage and accelerate our builds, being really flexible with the tools so they fit what we need. And we can build faster than we've ever built before. For so long, building has been top down.

Building has been you get an instruction from someone at work or you try a weekend project and you have to plan it out and you have to do every single line yourself. It's how we built the pyramids and it's how we built software for a long time.

It hasn't changed in a long time, just like our cognitive architecture. That's not true anymore. Now we have the ability to very rapidly digest, meld, and understand the best of the build communities around us. And we can implement it quickly because AI is getting good enough to provide real implementation muscles.

We have built superpowers we have never had before and that are not technically gated. And that makes 2026 a really exciting year to be a builder. So I hope you enjoyed this tour through how the community just tore through the second brain, built lots of different things. I love it. It's a great community. Just a little plug.

It's not great because of me. It's great because of the community. And I hope you go and build something fun. It doesn't have to be second brain. Use AI, find a community, doesn't have to be mine, and have fun. Go build.