Matt Pocock's Claude Code Skills Beat Superpowers Now

Eric Tech | 24:17 | transcript saved 23 Aug 2026
https://www.youtube.com/watch?v=8D8ewFBJfFM

Why Matt Pocock Built These Skills

Have you ever wanted to build something with AI, but AI never builds you what you want? Part of the reason is that AI doesn't have the full context, the full story of what you want to build. If you were to give it everything you want — typography, colors, functionality, everything — and embed that into AI's brain, there's no way AI wouldn't build applications with the highest accuracy. But the process of collecting that context is really challenging, because AI doesn't know what context to collect.

Luckily, there's actually a skill for that. It's called the Grooming skill, and it's currently the second most downloaded skill of all time on the skill leaderboard. Essentially, what this skill does is interview you relentlessly to collect that context, then embed it into AI's brain so it can build applications with the highest accuracy.

That skill was created by Matt Pocock, an expert in the web development space. Most people know him because of the Grooming skill, but he's actually created over 51 skills, with over 13 million downloads total. So in this video, we're going to go over exactly what all those skills are, how we can use them to build applications with the highest accuracy, how they work behind the scenes, and how they differ from skills we've already covered on this channel, like G-Stack, Superpowers, and GSD.

With that said, let's get straight into the video.

The Motivation Behind the Skills

First, I want to talk about why he actually built these skills — what's the reason, what's the motivation? He thinks that yes, nowadays AI can build things really fast, but the accuracy it generates is really random. He thinks of AI as kind of a black box: you give it a prompt, give it instructions — let's say, "build me a checkout page" — and sometimes it gives you what you want, sometimes it doesn't, and oftentimes it gives you different styles and functionality than you initially imagined. That's why he built these skills: to control the randomness that AI generates — things like the Grooming skill, "to specs," or "to tickets," basically to harness the agent and rein in that output randomness.

Obviously, he's not the first one to build skills like this. We've already covered skills like G-Stack, Superpowers, and GSD on the channel, and those spec-driven development skills are really popular in the AI space. If you want a full breakdown of how those skills differ, you can check out that video — I explain it there.

How His Approach Differs

Essentially, what Matt Pocock thinks is that he doesn't really like those other skills, because they tend to take over the entire pipeline — every one of them wants you to follow the whole process start to finish. For example, G-Stack has a brainstorming session, an auto-plan skill, an office-hour skill; GSD has a skill that breaks the work into different phases. And these skills are all chained to each other — you have to run one after the other, so there's a dependency between them.

So let's say you trigger skill number two, and it goes wrong, and you don't realize it until skill number six.

If you then want to change direction on the project's development, you have to rerun the entire pipeline again. There's no way to just trigger step two and rerun the rest — the whole process is chained together and you can't break it apart.

That's why Matt Pocock doesn't like that approach. Instead, he thinks skills should be broken into modular pieces that can be triggered in any order. Rather than one big framework where changing one thing breaks the entire process, you build smaller, modular, reusable skills. So if I finish the implementation skill, I can still trigger the "to specs" skill, and after that, I can still trigger the Grooming skill — in whatever order I want.

That's how Matt Pocock's approach differs from a lot of traditional spec-driven development skills.

The Grooming Skill

Now let's take a look at his most popular skill, the Grooming skill — the second most downloaded skill ever on the skill leaderboard. To put it in perspective, the skill itself is very short, just a couple of lines — that's the entire skill. I've summarized it into five rules.

The first rule is relentless: while the AI agent and you haven't reached a shared understanding, it will relentlessly — continuously — keep asking questions until both of you reach that shared understanding.

The second rule is the decision tree: every time it asks a question, it tackles only one branch at a time. So if it's building your application, it's not going to ask you one question about the checkout page, one about the about page, and one about some other page all at once — jumping all over the place. It focuses on one branch, one decision, at a time — say, the checkout page — and once that's fully cleared, it moves on to another page or decision.

The third rule is one at a time: it waits for your answer before asking another question.

The fourth rule is recommend one: it never just hands you a blank question — it gives you recommendations and options to choose from.

The fifth rule is do not act: it has to wait for your confirmation before it acts.

That's the five rules the skill offers. The whole process works like a decision machine: the model supplies you with options, and you make the decisions and stack them on top. Every decision it makes leads it to keep asking more questions until there are no more questions left in the decision tree. That's essentially the slash-Grooming skill.

About This Channel

Before we continue — if you've watched this far, you probably want to build something with AI. Quick context: my name is Eric, and I used to be a senior software engineer at companies like Amazon and Microsoft. On this channel, I help you master how to build AI agents, automation workflows, and real SaaS products. If that interests you, like this video and consider subscribing for more content like this.

Lastly, if you want to take your skills to the next level, I have a Skool community where I hold you

Matt Pocock's Claude Code Skills Beat Superpowers Now

Eric Tech | 24:17 | transcript saved 23 Aug 2026
https://www.youtube.com/watch?v=8D8ewFBJfFM

accountable. In that community, we have tons of classes you can use to learn about different topics in AI. If you have questions, you can post them there, and we also hold weekly live calls where I hold you accountable and answer any questions you have about the AI space.

Two-Spec Skill

If you're interested in that, make sure to check it out in the description below. But with that being said, let's get back to the video. Like I said earlier, Gloomy Skill is just one of the skills that Matt Pocock created, and there are actually a lot more skills he created that are also really popular — among them are skills that can help us build applications with the highest accuracy. Let's take a look at this one, which is the two-spec skill.

Essentially, what the two-spec skill does is that after you build a consensus between the agents and yourself, the next thing we're going to do is actually write it down. Because if you don't do that, and if you just close the conversation, everything you talked to the agent about is just going to be gone — all the agent's questions, all the grilling session you've done previously, everything's gone if you don't write it down. So that's why two-spec is going to freeze that, or in this case convert that into a spec, or an overall design plan on exactly how we're going to implement this.

The way this skill differs compared to any other two-spec skills from other spec-driven development frameworks is that Matt enforced this skill to never put any code block in the spec file itself. There could be really highly technical stuff, but he never wants to put code in the spec, because if you put it in the spec, the agent is going to follow that code. But what if you have the agent follow this code, and this code is outdated by the time you actually start building it? You can clearly see that would make a huge difference. So the spec without code is going to be a lot cleaner, and this will actually force the AI agent to look at the code, cross-reference the code, and make better decisions on what's right. That's why it doesn't make sense to put code in the spec.

Two-Tickets Skill

Now that we know exactly how the Gloomy skill and two-spec work, let's move on to the next skill, which is two-tickets. Two-tickets is exactly like what the name says — converting what we have in the spec into actual tickets that the AI agent can form.

The way it groups those tickets, or breaks the spec into different tickets, is also very different compared to other spec-driven development frameworks. Instead of grouping them by layer, which is what most traditional spec-driven development does — for example UI or API, where I could have a phase or ticket on the database, another ticket on the API, and another ticket on the UI side — what happens there is that after one ticket or one phase is finished, I only have one thing: my database. And if I have API or UI builds, I have to wait until all the tickets are complete in order to test the application.

What Matt Pocock proposed here is that instead of grouping by layers, we should group by features. So I can have ticket one just building out the login page, and I can test that entire feature end-to-end because the feature is fully functional. This makes it easier to test and makes the application very modular, so I

can change requirements or do other things going forward. That's the whole point Matt Pocock proposed: slice by feature, not by layer.

Implementation and Test-Driven Development

Once we've converted our spec into different tickets, what we do next is start implementation. For that, Matt Pocock also created a skill called slash-implement, which helps us write the code. If you look at the implementation skill, you'll see it's also very short. Essentially, what the skill does is trigger slash-TDD, which is test-driven development, whenever possible.

Test-driven development basically means writing tests first, before writing code — something that can actually fail first, before you run the implementation. For example, let's say we have Claude building a checkout page. We have it write tests first to define the requirements we're going to set. Initially it's going to be zero passing, because we haven't had any implementation yet. Once Claude finishes the implementation, it's going to run the tests and make sure everything passes based on the initial requirements set in the tests.

If we were to do it the other way around — write the code first — the test would actually be based on whatever the code writes. So if there's a bug in the code Claude writes, the test is also going to be satisfied; it's going to say everything succeeded, but there are actually bugs inside that we didn't catch. But if we write the test first, the code is shaped based on the test we write. This way, we can prevent bugs from being introduced in the code implementation itself.

Code Review Skill

After we have the full test-driven development complete, the next thing I'll show you is the skill called code-review, to review the work that's been done. Unlike other traditional code review skills we've seen before, where it's just checking for bugs in the same session it was written in, this time is going to be very different, because we're actually going to start a fresh context window and follow a code review checklist that Matt Pocock created, walking through the items step by step to make sure the code is fully checked.

That checklist is only 12 vocabulary words coming from the book *Refactoring* by Martin Fowler. Matt extracted 12 keywords from that book and embedded them completely into that skill, using those vocabulary terms for Claude to check against.

Just to give you an example of the keywords in the checklist: one is called "shotgun surgery," which means that if you were to change a color, a requirement, or a feature of your application — for example, a button color — and you have to change it in multiple places, that's going to be a problem, because if you miss one, that process becomes inconsistent. Essentially, shotgun surgery checks to see if anything like this happens in our codebase.

Another example is called "feature envy." What feature envy does is make sure that logic belongs in the right file. So if logic is in orders.ts but should actually be in inventory.ts, then that logic is living in the wrong place. We want to make sure logic is in the right place — that's what the feature envy check looks for.

Matt Pocock's Claude Code Skills Beat Superpowers Now

Eric Tech | 24:17 | transcript saved 23 Aug 2026
https://www.youtube.com/watch?v=8D8ewFBJfFM

The last one is called "data clumps," meaning that if there are many data types being referenced by many functions, maybe we should group those data types into one single type that's much easier to reference across those functions.

The Skill Uses Vocabulary from Books on Best Practices

You can see that smaller things like these are what they check for in the code review skill. Now, if you're not really technical and you still don't understand what those terms are, that's completely fine. Just know that those terms, those twelve vocabulary words, are coming from books that lay out best practices on how to write better code. What that's saying is that simply by referencing those terms from that book, Claude here is going to follow clear instructions referenced from that book on how to clean up code much better.

That's essentially what it is: referencing those vocabulary words rather than writing step-by-step instructions on how to clean the code in the skill itself.

Matt Pocock's Skills Are Short and Concise

If you actually look carefully at the skills that Matt Pocock created — for example, this code review skill — you can see that the skill is really, really short. It doesn't matter what skill you look at, it's always short, concise, and doesn't have any useless words inside it. That's because he believes every extra word you put in the skill is a distraction that can cause the AI model to hallucinate.

That's why he created a skill called "writing for agents," which is about writing documentation for agents whenever you're using it to create or edit skills. The first thing that skill helps you do, to make it more like Matt Pocock's style, is help you prune the skill. Let's say a skill contains 1,000 words — it helps you prune or compress that down to fewer words, so it's much easier to write skills that are concise. Instead of something like "I would really appreciate it if you would do this," it becomes more like "just go direct with what you're trying to do" — "interview me relentlessly," and so on.

The second thing the skill does is use phrases and vocabulary that carry deeper meaning — things like shotgun surgery, data clumps, or feature envy. We already talked about what those mean, and essentially we can reference words with deeper meaning so that whenever Claude looks at it, it knows exactly what's meant. You don't have to explain "do this first, do this second" — Claude sees those vocabulary words and already knows what they mean. So we keep the prompt short and concise, which makes it much easier for Claude to follow in the skill itself.

So by using vocabulary with deeper meaning, and by being able to prune the skill to be shorter, that's how you can use the "writing for agents" skill to write skills similar to Matt Pocock's style.

The Problem of Scattered Logic

By this point, if you follow the Matt Pocock skills we've talked about in this video, you can pretty much make your AI a senior software developer. Pretty good. But there's one big problem we haven't talked about: AI sometimes writes scattered logic or code.

Matt Pocock's Claude Code Skills Beat Superpowers Now

Eric Tech | 24:17 | transcript saved 23 Aug 2026
https://www.youtube.com/watch?v=8D8ewFBJfFM

What do I mean by this? Let's say we have a main program that processes payments. This payment-processing program calls different functions, like calculating discounts or validating cards. If we want AI to process, or in this case learn about, this main function, it's going to jump through a lot of hoops — it has to look at the entire function and every function that this function calls, in order to learn about the program. By the time it jumps through all those hoops, the context window is probably already filled up.

Don't get me wrong — this approach has its own benefit, because you can unit test each individual module or function by itself. That's good. But the problem is that it wastes a lot of tokens during the process of AI trying to learn about that function.

Encapsulation Saves Tokens

Imagine AI wants to learn how this program works by looking at every single function it calls — that wastes a lot of tokens. So what he's proposing is: since AI can only look at a limited amount of stuff at once, we can have AI just look at one thing. This one function encapsulates all the other functions, so AI can save more tokens by only looking at this one function rather than dozens of others that it calls.

Essentially, we encapsulate things so AI doesn't have to look elsewhere and waste tokens trying to understand things. Maybe it would take 10,000 tokens to understand this function before, but now it might only take 1,000 tokens, which makes the process a lot easier for AI to understand.

Now, don't get me wrong — one misconception here is that people think this just means stuffing all the smaller functions into one single file, resulting in something like 10,000 lines of code in one file. That's not the purpose. What we're trying to do is have each of those functions still live in its own file, but create one single "door" for AI and the program to access. Before, there were so many doors, and AI had to jump through so many of them to understand exactly how those functions call each other — this function calls that function, and that function calls back — and all that back-and-forth wastes a lot of time and tokens for AI. That's what we're trying to reduce.

Deletion Testing

Yes, the book does have a skill for this, which we'll talk about in a slide. That skill also has functionality to do deletion testing. Essentially, let's say we have a main program that calls a function in question. The deletion test tries to see if this function is a "fragment" — whether it's really necessary to exist, whether it has no use. If it has no use, maybe we should remove it.

Let's say we try to remove it and see if all the tests still pass — does all the functionality still work? If the test doesn't pass because that function was deleted, then we should keep it, because it's a real dependency of the main program. But if we remove this function and the whole program still works the same, then there's no point keeping it — we should delete it.

That's the whole point of running a deletion test on any functions that the main program depends on.

The "Improve Codebase Architecture" Skill

Matt Pocock's Claude Code Skills Beat Superpowers Now

Eric Tech | 24:17 | transcript saved 23 Aug 2026
https://www.youtube.com/watch?v=8D8ewFBJfFM

Now let's get to the skill he introduced called "improve codebase architecture." Essentially, this skill scans the entire Git log, walks through the hot files, runs deletion tests — everything we just discussed — and gives you an architecture review page that tells you exactly what things you could delete or improve in the architecture.

I've already run mine, and here's my architecture review page that it generated. For example, for one of my projects, here's one thing it recommends: the renderHTML function has no callers — no function calls it. You can see these are what look like parent functions, but in this case they're not really parents because they don't actually call it. So this stuff here is redundant.

Cleaning Up Modules and Duplicate Functions

We should delete both files here because those are still referenced. And then we also have things like this — for example, two shallow modules here. So these two functions are being called by two different functions, and those two functions are identical, which means we actually created both of them twice. If you were to refactor this, we'd be able to unify this function and this function into one, so it's much easier for us to refactor.

There's also another example here: one module owns the posting file format, and you can see what it looks like after refactoring on the right. Overall, this makes the entire coding process — the entire architecture — a lot cleaner, and also makes it more token efficient.

Wrapping Up the Skills Overview

We've now gone over pretty much all the most popular skills that Matt Pocock created — Ruby's, two-spec, and so many more — in this video so far. But obviously there are many other skills he's created that we haven't mentioned. If you want to go deeper on all the other skills I've talked about, as well as some deeper use cases on how we can use those skills in practical situations, check out our ASBuilder in our school community — in lesson four, we dive deep into what those skills are and how to apply them to building real applications.

My Opinion on These Skills

What do I think about the Ruby skills, all the skills we went over in this video? Like I said earlier, the reason Matt Pocock created skills is because all those other frameworks try to enforce you to follow their entire process start to finish, and there's no way for you to break out in the middle of that process. That's what he's trying to solve with his own skills.

I also think this mindset of reducing dependency on skills — making them modular — is the way to go, because nowadays models like GPT-5.6, Fable 5, or Opus 5 are really, really smart. You don't have to make those skills tied together. Now, for a weaker model like Sonic 5, or models like DeepSeek, or other models that aren't frontier, you'd probably want to use something like Superpowers, because those models are weaker — they don't have that kind of knowledge built in, they're lightweight, and they don't have that kind of framework in place.

Matt Pocock's Claude Code Skills Beat Superpowers Now

Eric Tech | 24:17 | transcript saved 23 Aug 2026
https://www.youtube.com/watch?v=8D8ewFBJfFM

So for those, using Superpowers is probably ideal. But for a frontier model, it's better to follow skills similar to Matt Pocock's — making it modular, mostly relying on the AI to make the best decisions rather than leaning on the skill itself. The skill here is really just a harness — trying to guide the AI as minimally as possible, pointing it in the right direction, and that's all.

Models Are Getting Smarter, So Skills Need Less Hand-Holding

That reminds me of a video I made. In that video, I talked about how, as models get smarter and smarter — you can see the orange line here is Opus 5 and the blue one is Opus 4.8 — as models get smarter and cheaper, companies, even Anthropic themselves, have deleted over 80% of their system prompt for newer models like Opus 5. That's because models are getting stronger and don't need as much harness — they have that harness or those skills embedded into the model itself. So if you use old-style skills like Superpowers that really hand-hold the model and don't let it think creatively, it's actually going to hold back the results you get.

That's why, after studying all the skills Matt Pocock created, I realized we shouldn't put too many guardrails on skills — too many instructions that hold back model performance. We should write skills similar to what Matt Pocock created, using the vocabulary and architecture covered in this video to prune the skill down without stopping the AI from thinking creatively about any kind of problem. That's really what I believe is the trend going forward: before, we had so many instructions embedded in skills, but now we don't need as many, because the models are getting smarter.

Let Me Know What You Think

I'm curious what you all think. Do you think we should keep more guardrails, like what we've seen before with Superpowers, GStocks, GSD, baked into the skill itself? Or do you think we should go the other way and make skills closer to what Matt Pocock did? Comment down below — and if you think there are other skills from Matt Pocock's repositories worth highlighting, comment those too. Let me know what you think; I'll make sure to read every comment and answer all of them.

That's it for this video. If you found value in it, please like the video, and subscribe to the channel if you want to see more content like this. With that being said, I'll see you in the next video.