Episode Book

EPISODE #5940

The Missing Middle: Content APIs and ISR for a 6,000-Episode Catalog

PUBLISHED	13 October 2026 at 18:16 UTC
DURATION	32:14
LISTEN	myweirdprompts.com/episode/content-api-isr-architecture-06199/
HOSTS	Corn, Herman, Hilbert
SCRIPT MODEL	deepseek-flash
VOICES	chatterbox-regular (chatterbox-local)
PIPELINE	V5.4
SOURCES	16 cited by the research (listed at the end)
BOOK GENERATED	19 October 2026 at 17:41 UTC

SYNOPSIS

Daniel has built a 6,000-episode podcast with no admin panel — just Claude Code and a custom admin MCP talking to Neon. But Neon's docs explicitly warn against connecting MCP agents to production databases, and Vercel's 45-minute build ceiling looms over every deploy. This episode works through the architecture question: does the catalog need a bespoke content API, or do Vercel's ISR with tag-based revalidation and Neon's Data API already provide the arbiter layer? We dig into why the current "batch of five triggers a deploy" pattern breaks, how generateStaticParams and on-demand revalidation flip the model from build-time to request-time, and the one caveat that actually matters — every new deployment starts with a cold ISR cache.

TRANSCRIPT



Corn

Here's the question that's been rattling around my head since I read this. When the catalog is too big to rebuild on every change, what actually has to change? The data? The front end? Or the thing in the middle that nobody's built yet?



Herman

That's the whole episode right there.



Corn

That's from Daniel. He's been building this podcast in public for nearly six thousand episodes now, and he wants to hash out our own architecture on air again. The stack as it stands. Vercel serves the front end, the thing that drives the feed. Neon holds the data, serverless Postgres. Modal does the text to speech generation. And the production process is a long running Python function that updates the database and then triggers a deployment every time it's got a batch of five new episodes ready.



Herman

Five at a time. That's a lot of deploys.



Corn

And he's been wanting to sit down and actually curate. Manually tag episodes, create tags that are missing, prune the ones that are superfluous, fix a few category assignments. Plus he's got a mental list of his all time favorites that he wants to turn into a Creator's Choice category on the site. Which means he needs an admin interface. And here's the part that got my attention. Everything to this point has been built with Claude Code and a custom admin MCP that he wrote specifically for the routine stuff. Deleting accidental duplicates, that kind of thing. It works. But it's not a GUI. It's Daniel talking to an AI agent that talks to the database.



Herman

Which is a remarkable run, honestly. Six thousand episodes with no admin panel.



Corn

So the question he's actually asking isn't how to build an admin tool. It's what happens to the architecture once you decide the catalog is too big to keep rebuilding from scratch. Does an API layer need to sit between the database and the front end? And can the front end get deployed from that content API instead of pulling straight from Neon?



Herman

And I'll flag the thing I found in the docs before we get deep into this, because it's the kind of detail that reframes the whole episode. That admin MCP he's been running? Neon's own documentation says don't do that. Specifically. We recommend MCP for development and testing only, not production environments. Never connect MCP agents to production databases.



Corn

Never is a strong word for something that's been working fine.



Herman

That's the tension. It's not a hypothetical, it's the thing that made the whole project possible, and the vendor is on record saying it shouldn't be pointed at a live database.



Corn

So we've got three layers in play. Data in Neon, delivery on Vercel, and a missing middle. That missing middle is the entire episode, I think. Daniel calls it a content API. Something that arbitrates.



Herman

Right. And here's the thesis I want to test over the next half hour. The instinct to build a bespoke API is correct in spirit but possibly wrong in implementation. Because Vercel's ISR with tag based on demand revalidation and Neon's Data API already give you most of what that arbiter layer needs to do.



Corn

Two products he's already paying for, doing the job he's about to build from scratch.



Herman

Roughly. Though there's a catch in each of them, and we'll get to both. But the shape of the answer is that the missing middle might already exist and just needs configuring.



Corn

Let's put the scale number on the table. Six thousand episodes. And Vercel's hard ceiling on a single build is forty five minutes. That's the wall.



Herman

Forty five minutes per deployment, every plan. That's the constraint everything else has to work around.



Corn

So the first question is why the current pattern breaks. And Vercel has actually published the number that tells you when.



Herman

They have, and it's unusually blunt for vendor documentation. Their guidance on reducing build time says a Next.js project pre-rendering thousands of static pages can hit that forty five minute maximum. And then the line that matters. The fix is to stop doing all of that work on every build.



Corn

Stop doing all of that work on every build.



Herman

That's the whole migration in one sentence, Daniel could have written it himself. And they name two drivers of long builds. The first is page count. Pre-rendering scales with the number of pages. So six thousand episodes is six thousand pages, and that's before you get to any index pages or tag pages or anything else that expands the surface.



Corn

And the second driver?



Herman

Per page work. Every page that does data fetching or image generation or any heavy computation repeats that work, and the phrase they use is that slow upstream calls multiply across the build. So it's not one slow query that hurts you. It's one slow query happening six thousand times in sequence during a single build.



Corn

Which is exactly what we're doing. Every deploy, the build reads the whole catalog out of Neon and renders the front end afresh. And the batch of five trigger means that deploy fires constantly.



Herman

Constantly is right. Every five episodes. So with the cadence this show runs at, we're not talking about a weekly build that has room to be slow. We're talking about a build that fires all day, and each one re-reads a catalog that only grows.



Corn

And the thing that grows is the expensive part.



Herman

Always. The build time isn't flat, it's a function of how many episodes exist. So the current architecture has a cost curve baked into it. Every batch of five makes the next build slightly worse than the last one. You don't feel it at a thousand episodes. You feel it creeping at five thousand.



Corn

It's a subscription to your own growth.



Herman

And Vercel's recommendation for what to do instead is right there in the same document. Pre-render a small set of popular pages at build time and generate the rest on demand. In the App Router there's a function called generateStaticParams that defines which paths get built ahead of time, and a setting called dynamicParams that controls what happens to everything else. If you return an empty array from generateStaticParams, you skip pre-rendering entirely and Vercel generates each page on first request instead.



Corn

Which flips the model. Instead of paying for sixty five hundred pages on every deploy, you pay for a page once, when someone actually asks for it.



Herman

That's the shift. Build time stops tracking catalog size and starts tracking how many pages are hot.



Corn

So that's one half of the answer. The other half is what happens when the content changes after the page exists.



Herman

That's Incremental Static Regeneration. ISR. And the way to think about it is it combines the speed of static content with the flexibility of server side rendering. It follows a pattern called stale while revalidate. A visitor gets the cached version of the page, the fast one, and in the background Vercel regenerates it so the next visitor gets the fresh one.



Corn

So nobody's waiting on a regeneration. Somebody's getting yesterday's version while today's is being built behind the curtain.



Herman

And there are two ways to trigger a regeneration. Time based, where you set a revalidate interval and the page refreshes on that schedule. Or on demand, where an API call tells Vercel the content changed.



Corn

On demand is the interesting one for us, isn't it. Because it's the difference between refreshing on a timer and refreshing because something actually happened.



Herman

It's the difference between guessing and knowing. And this is where the documentation gives the line that I think answers Daniel's question almost directly. Instead of revalidating entire paths, you tag cached content and invalidate specific tags when the underlying data changes.



Corn

Say that again, because I want to make sure I've got it.



Herman

Tag cached content and invalidate specific tags when the underlying data changes. So instead of invalidating a whole path, or worse, redeploying the whole site, you tag the things that a piece of content belongs to, and when that content changes you invalidate the tags it touches.



Corn

So episode fifty nine thirty eight isn't just a page. It's a member of a set of tags.



Herman

It's a member of whatever sets it belongs to. Say it's in a category, and it's in a tag for a guest who appeared on it, and it's in Creator's Choice. Those are three tags. If Daniel fixes that episode's category assignment in the admin tool, you invalidate the tags that episode touches, and only the pages carrying those tags regenerate.



Corn

Not the other five thousand nine hundred and ninety nine.



Herman

Not touched. That's the entire win. Build time stays flat as the content grows, because the generation is deferred. Pages generate on request or through an API call instead of during the build. The catalog can go to ten thousand episodes or twenty thousand and the deploy doesn't get any slower.



Corn

And this is a real mechanism, not a proposal. Somebody's shipped this.



Herman

It's in the product. Durable caching holds generated pages for up to thirty one days or until you revalidate them. Purges are globally consistent, so a cache purge propagates across every region within three hundred milliseconds. And there's a thing called request



Herman (continued)

collapsing where multiple requests for the same uncached path fold into one function invocation per region, so you don't get a stampede when a cold page gets popular.



Corn

Three hundred milliseconds to purge globally. That's faster than most people can decide they want the change.



Herman

Faster than this conversation, certainly.



Corn

So walk me through what happens today, so I can hold it up against the new version.



Herman

Today, a batch of five episodes finishes. The Python function on Modal writes them into Neon. Then it triggers a Vercel deploy. And that deploy reads the whole catalog out of Neon and re-renders the front end from scratch. Every episode, every tag page, every category. All of it, every time.



Corn

Five episodes of new content, six thousand episodes of work.



Herman

Every five episodes.



Corn

And the ISR version?



Herman

The admin writes go through the API. The API calls revalidateTag on the tags the write touched. Say revalidateTag for the episode, revalidateTag for the category. And only those pages regenerate. The build itself doesn't run. There isn't a build. The change propagates in the time it takes that purge to reach the edge, which is measured in hundreds of milliseconds.



Corn

So the deploy isn't the mechanism for reflecting content changes anymore. The deploy is just for shipping code.



Herman

That's the cleanest way to say it. And it's a decoupling that the current architecture doesn't have, because right now deploying and updating content are the same operation.



Corn

They're fused. Every content change is a code deploy.



Herman

Which means code that has nothing to do with episodes has to wait for a build, and content changes that have nothing to do with code are scheduled like software releases. Those are two different clocks and the current system makes them the same clock.



Corn

Hold on. What about the case where you do deploy code? A real deploy, a new feature.



Herman

That's the caveat, and it's the one that actually matters for Daniel's question. Each new deployment uses its own ISR cache and does not reuse the cache from a previous deployment.



Corn

So a real deploy still starts cold.



Herman

Every page you had cached in the previous deployment doesn't carry over. New deployment, fresh cache. The pages regenerate on request, so the site isn't down and the first visitor doesn't wait long, but you've lost the warmth. Multiply that by every deploy and you can see why the batch of five trigger is the actual problem.



Corn

Five episodes, cold cache, every time.



Herman

If every content change forces a deployment, you're not just paying the build cost, you're resetting the cache. Which is why on demand revalidation isn't a nice optimization on top of the current pattern. It's the thing that replaces the current pattern. The batch of five should write to the API and invalidate tags. It should never touch a deploy.



Corn

Because the deploy is the only thing that can cold start the cache you just built.



Herman

Right. So the architecture Daniel's describing isn't really a new layer bolted on. It's a separation of two things that got fused. Code deploys are rare. Content updates are constant. Those go through different paths now.



Corn

So the delivery side has a shape. The question I'm sitting with is what calls revalidateTag.



Herman

And that's the arbiter layer. Because revalidateTag has to be called by something that knows the content changed and knows what it touched. And in Daniel's model, that something is the content API.



Corn

So ISR solves the delivery side. But it only works if there's something in the middle that knows when to call revalidateTag, and that's the arbiter layer.



Herman

Which leads us to the bad practice claim, and I want to take it seriously because Daniel's instinct there is correct and the reason is subtle.



Corn

He said propagating changes directly to the database sounds like bad practice. I want the actual mechanism, not the vibe.



Herman

The vibe is true, but here's the mechanism. If an admin tool writes straight to Neon, you skip three things. Validation, so nothing checks that the write makes sense before it lands. Authorization, so nothing checks that the writer is allowed to do it. And cache invalidation, which is the one that actually breaks the architecture.



Corn

Because there's no single place that knows a write happened.



Herman

There's no single place that can call revalidateTag. The database has no idea the front end exists. So if your admin tool writes to Neon directly, the only way to make the change visible is to trigger a redeploy. Which is the exact pattern you're trying to escape.



Corn

So the bad practice isn't really about the database. It's about the missing seam.



Herman

It's about where the cache invalidation lives. And the answer is that it has to live somewhere the write passes through. That's what makes it an arbiter rather than just a middle layer. It's the one place that knows both that the content changed and what depends on it.



Corn

So the API layer earns its place by being the thing that knows about both sides.



Herman

And here's the part of the research that made me sit up. Neon already ships that layer. It's called the Data API.



Corn

Neon has a content API already?



Herman

It does. It's a REST endpoint, PostgREST based, enabled per branch. And the detail that matters for the arbiter role is this. The Data API has no permission layer of its own. Every request is authorized entirely by your PostgreSQL GRANT statements and Row Level Security policies.



Corn

So the authorization model isn't in the API. It's in the database.



Herman

It's in the database, and the API is just the door. Which is a completely different shape from building a service that reimplements permissions in application code. Here the policies live where the data lives, and the API just enforces them.



Corn

That's less code. That's a lot less code.



Herman

It's a lot less code, and it's a smaller surface. And the way roles work, there are three. Authenticated, anonymous, and custom roles. So you can define a role called admin, and that role maps to a Postgres role called admin, and the grants on that role are what the admin interface is allowed to do.



Corn

So the role split is the whole design. Admin role gets write grants for the admin interface. Anonymous role gets select only for the public front end.



Herman

That's the concrete configuration. The public front end authenticates as anonymous and can only read. The admin interface authenticates as admin and can write. And neither of those is enforced by application code that someone might refactor badly. They're enforced by GRANT statements and RLS policies in Postgres.



Corn

Which means if the admin UI has a bug and sends a delete, the policy still stops it.



Herman

If the policy is written correctly. Row Level Security has three states, and it's worth knowing which one you're in. Disabled, where every authenticated user sees every row. Enabled with no policies, where all access is blocked, which is the safe default. And enabled with policies, where rows get filtered, typically by matching against the user id.



Corn

So the failure mode of forgetting to write a policy is closed, not open.



Herman

That's the design intent. Enabled with no policies blocks everything. You have to actively write the policy that opens access, which is the right direction for a thing that's about to be pointed at production.



Corn

So the API layer becomes a policy surface rather than a service you build.



Herman

That's the reframe. You're not writing an API. You're writing policies and letting the Data API enforce them. And the JWT role in the request is what selects which policy applies. Tokens expire in about fifteen minutes, so there's a short window on any credential in flight.



Corn

Good. Now the part I've been waiting for. The MCP.



Herman

So the current admin tool is an MCP server that Daniel built and connected to the production database, and it works. And Neon's own docs say, and I'll read it because the wording is unambiguous, we recommend MCP for development and testing only, not production environments. Never connect MCP agents to production databases.



Corn

And this is the tool that built the project.



Herman

That's the tension, and I don't think it's a gotcha. The MCP is what made six thousand episodes possible without an admin panel. It's the reason there's something to administer. But the vendor is saying the pattern it's being used in is not the pattern it was designed for.



Corn

Why would the vendor say that, though? What's the actual risk?



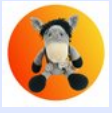
Herman

The general answer is that an MCP server is an agent with credentials, and the agent treats tool input as a thing to act on. Every input is untrusted until something decides otherwise. So if the tool passes input straight into a SQL string or a shell or a filesystem path, you've created a channel where a bad input becomes a bad write.



Corn

So it's not that the MCP is broken. It's that the MCP is a credential holder with an open input surface.



Herman

And scoping down to what the tool actually needs is the mitigation, not the elimination. Neon's server supports a read only mode, where select and schema inspection work and writes are disabled. It supports scoping to a single project. It supports filtering to specific categories of tools, so you can expose querying without exposing schema changes.



Corn

So you can narrow it a lot without turning it off.



Herman

You can narrow it a lot. And there's a version of the current setup that's defensible, which is the MCP pointed at a development branch with read only scoping, and the admin writes going through the Data API with the admin role. That gets you the same editing capability and moves the write path off the agent.



Corn

So the MCP becomes the thing you use to think, and the API becomes the thing that writes.



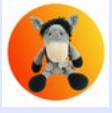
Herman

That's the split I'd draw. And it lines up with what the migration toward the content API wants anyway.



Corn

There's a general set of practices that applies to any agent that's writing to a live system, isn't there. I want that list, because Daniel asked how other people build around this stack, and this is the piece that generalizes.



Herman

It does generalize. Treat each server as a bounded context, so it does one thing. Prefer stateless and idempotent tools, so running the same operation twice doesn't produce two rows. Treat every tool input as untrusted, which means never passing input straight into a shell or a SQL string or a file path. Scope credentials down to exactly what the tool needs. And never auto approve destructive operations, so a delete always requires a human deciding to delete.



Corn

And the hints on the tools themselves?



Herman

Read only hint and destructive hint are user experience hints, not security. They tell a well behaved client what a tool does. They don't stop a badly behaved one. So if you're relying on those annotations to keep an agent from dropping a table, you're relying on the agent's manners.



Corn

You're trusting the lock because the door has a sign on it.



Herman

The sign says please. It doesn't say no.



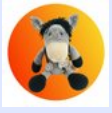
Corn

So the write path goes through the API, the API has policies in Postgres, and the policies are the actual boundary.



Herman

Which is why the arbiter layer isn't just for cache invalidation. It's also where the security model lives, and the security model is in the database rather than the service. That's the difference between building an API and enabling one.



Herman

Go ahead, because I think it's the right template and I want to see you land it.



Corn

The framing is that a headless CMS is a content API with an editing interface bolted on top. It stores content as structured data, exposes it through an API, and doesn't care what consumes it after that.



Herman

And the AWS definition is the same shape from the other direction. It separates content management and storage from content presentation. So the API is the contract and the presentation is just a consumer.



Corn

Apply that to Daniel's stack. The Data API plus ISR is the content API. The admin interface is the editing interface bolted on top. And the Vercel front end is a consumer like any other.



Herman

Which is the reframe that dissolves the question. He was asking whether to build a content API. The headless CMS pattern says the content API is the definition of the system, and the editing interface is a client of it, not the other way around.



Corn

The admin UI isn't special. It's a client with write permissions.



Herman

That has a practical consequence for the Creator's Choice problem. If Creator's Choice is just another tag, then the admin interface doesn't need a special code path to curate it. It already has the flow. Write through the API, invalidate the tag, let ISR regenerate the affected pages.



Corn

The curation feature is a row in a tag table and a page that queries for it.



Herman

The tag based invalidation means editing the Creator's Choice list only regenerates the pages that show it. Not the episode pages. Not the category pages. Just the ones carrying that tag.



Corn

Because a tag is a dependency edge you get to declare.



Herman

That's the whole point of tagging cached content instead of invalidating paths. The tag is the edge. When you invalidate it, you walk the edges to find what actually needs regenerating.



Corn

That's a lot of machinery for something that reads like a curation UI on the surface.



Herman

The curation UI is the easy part. The hard part is knowing what changes when a row changes, and the tag is the answer to that.



Corn

Let me do the concrete version, because I think this is where the episode has to be useful to somebody who isn't us.



Herman

Go.



Corn

Admin role. Write grants on the episode table, the tag table, the category assignments. RLS policies that scope what the admin role can touch. An admin interface that authenticates as admin and talks to the Data API. Anonymous role, select only, no write grants at all. The public front end authenticates as anonymous and only ever reads. And every write path funnels through the API, which knows the tags it touched and calls revalidateTag.



Herman

The reason somebody else can copy that is that none of it is bespoke. Every piece is a documented configuration on a product they're probably already using, if they're on this stack.



Corn

Which is the interesting answer to Daniel's last question. How can others building around similar stack elements create their own admin utility.



Herman

The answer is that the admin utility is mostly the part you build, and the API plus the cache invalidation is mostly the part you configure. The thing you write is the editing interface and the policy. The thing you don't write is the arbiter.



Corn

That's a strange division of labor for the phrase build an API.



Herman

It is. Most of the API is a checkbox and a policy file.



Corn

Which is either good news or a letdown, depending on whether you wanted to write a service.



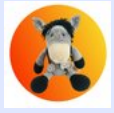
Herman

I think it's good news for a solo operator, and I say that as somebody who's written services that didn't need to exist.



Corn

There's an episode in here about the gap between building and enabling.



Herman

There probably is, but let's not write it today.



Corn

Fair. There's one more piece of the current stack I want to bring up, because it's the bit that has to survive the migration. The admin MCP.



Herman

It doesn't have to survive. Or rather, what has to survive is Daniel's ability to do routine administration, which the MCP currently provides. If those operations move to the Data API behind a thin interface, the MCP can be scoped down to a development tool and pointed at a branch.



Corn

The migration path off the MCP is the same path that leads to the content API.



Herman

They're the same move. You're taking the write path out of the agent's hands and putting it behind a policy boundary. The cache invalidation is the thing that makes it necessary, and the security model is the thing that makes it right.



Corn

The MCP that made the project possible is officially dev and test only, and the migration off it is the migration toward the content API.



Herman

That's the finding.



Corn

Let me ask you something, because I want to check whether I'm agreeing with you for the right reasons. If the Data API already exists, why has Daniel been running the MCP for all this time? Was that a mistake?



Herman

I don't think so, and I'd push back on that framing. The MCP existed because there was no interface. The Data API existed but there was no UI, and the MCP plus Claude Code was effectively the UI. It was a way to get an interface without building one. And at a certain scale that stops being true, which is the moment you need a real interface with real policies.



Corn

The MCP was the right call at the time and the wrong call now, and the reason it flipped is the cache invalidation requirement, not the security one.



Herman

The security one is real but it was always there. The cache invalidation one is what appeared as the catalog grew, because that's what made the current redeploy pattern expensive.



Corn

You sound very confident about that.



Herman

I'm confident about the shape. I'd want to be careful about whether the MCP was ever a security problem in practice. It might have been fine, because Daniel was the only user and the scope was narrow. The vendor's guidance is about the pattern, not about what happened on this particular setup.



Corn

That's more honest than most people would be.



Herman

I'd rather be right than tidy.



Corn

The delivery layer is ISR with tag based invalidation. The arbiter is the Data API with RLS roles. The editing interface is a client of the arbiter. And the MCP moves to development. That's the whole architecture.



Herman

Four sentences. Somebody should have written it down before we started.



Corn

That's the episode, and I think we've got it.



Hilbert

The thing nobody warned me about is that the cache is per deployment.



Corn

Hello, Hilbert.



Hilbert

I run a small thing. Records site. Maybe two thousand pages. Build times were fine for years, and then one week they stopped being fine. Four minutes became fifty one. I know it was fifty one because I was sitting in a car outside a laundromat timing it against the meter.



Herman

Against the parking meter?



Hilbert

The meter gave me twelve minutes and I was two runs away from finishing, so the build and the meter were the same clock for about a week. Anyway. Fifty one minutes. And the reason I know it's per deployment is I redeployed to fix a typo in a footer. A typo. And every page went cold at the same time. I sat there watching the log. The typo was not worth that.



Corn

You found the ISR cache caveat the hard way.



Hilbert

I found it at about the eight minute mark of a fifty one minute wait, which is when I started wondering whether the footer was really the problem.



Herman

The laundromat was because?



Hilbert

My washing machine is in a dispute with the building's plumbing. It's been eleven weeks. The car is my office on Tuesdays now. There's one on King George with a meter that takes the card, which matters because the other one doesn't.



Corn

Eleven weeks is a long dispute.



Hilbert

The plumber and the landlord have stopped speaking. I'm not in the middle of it. I'm just doing laundry in a car.



Herman

What did you land on as the fix, though? Because the per deployment cache is the thing we've been circling.



Hilbert

I stopped deploying. I put the writes behind the API and let it punch the tags, and the deploys went back to being about code. Which means the typo situation doesn't happen, because there's no deploy to touch.



Corn

You separated the two clocks before we did.



Hilbert

I separated the two clocks because the meter was running.



Herman

In your setup, did you put the writes behind the API because of the cache, or because of the policy?



Hilbert

Cache. The policy came later. And I'll tell you the piece that surprised me. The tag calls are three lines. That's the whole integration. Three lines in a handler. I spent a weekend planning a service and the actual change was three lines.



Corn

Three lines.



Hilbert

I've started recognizing the other regulars. There's a man who does two loads and reads the whole time, and a woman who brings her own detergent in a jar, and a third one who I haven't placed yet. I have a ranking. I'm not going to explain the ranking.



Herman

That's the closest we've gotten to the actual migration path. Three lines and no deploy.



Hilbert

The deploys stopped being the mechanism. That was the whole thing.



Corn

The thing Hilbert ran into is actually the thing we keep circling. The cache is per deployment, and that changes what the migration path looks like. If a deploy invalidates your whole cache anyway, then the fix isn't only about cache efficiency. It's about keeping the deploy off the content path entirely.



Herman

Which makes the pressure toward the content API stronger, because the API is what lets a write happen without a deploy.



Corn

The open question I want to leave with. If the arbiter layer is mostly configuration, RLS roles and three lines of tag invalidation, rather than a service somebody writes, does that change what building an API even means for a solo operator? Because the work is real but the work isn't code.



Herman

The unresolved tension worth sitting with is that the MCP which made this project possible is officially dev and test only, and the migration path off it happens to be the same migration path toward the content API. Those are the same move.



Corn

At six thousand episodes the build ceiling is a nuisance. At twenty thousand it's a wall, and the architecture decision is really about which wall you want to hit. You can hit the build wall, or you can hit the cache wall, but you can't avoid both by doing nothing.



Herman

That's where I'd leave it. If you're enjoying the show, a review helps other people find it. Corn, thank you as always.



Corn

Thank you. If this was your kind of episode, go back for episode five, Fine-Tuning ASR For Maximal Usability; episode four, If Your Voice Ages, Does Your Fine-Tune Become Useless; and episode seven, Building Custom ASR Tools. This has been My Weird Prompts. Our producer is Hilbert Flumingtop, who I'm told has a ranking system.



Herman

Send us your own prompt on Telegram at t dot me slash MWP listener bot. We'll be back soon.

SOURCES

What the research for this episode read before the script was written. Primary sources first.

1. **Vercel [primary]**
   Incremental Static Regeneration (ISR) docs
   https://vercel.com/docs/incremental-static-regeneration

2. **Vercel [primary]**
   Getting started with ISR docs
   https://vercel.com/docs/incremental-static-regeneration/quickstart

3. **Vercel [primary]**
   How do I reduce my build time with Next.js on Vercel? KB guide
   https://vercel.com/kb/guide/how-do-i-reduce-my-build-time-with-next-js-on-vercel

4. **Neon [primary]**
   Getting started with Neon Data API docs
   https://neon.com/docs/data-api/get-started

5. **Neon [primary]**
   Access control & security (Data API) docs
   https://neon.com/docs/data-api/access-control

6. **Neon [primary]**
   Neon MCP Server overview docs
   https://neon.com/docs/ai/neon-mcp-server

7. **Neon [primary]**
   Neon serverless driver docs
   https://neon.com/docs/serverless/serverless-driver

8. **Neon [primary]**
Connection pooling docs
https://neon.com/docs/connect/connection-pooling

9. **Neon [primary]**
FAQ What is the best Postgres setup for serverless APIs?
https://neon.com/faqs/best-postgres-setup-serverless-apis

10. **Model Context Protocol [primary]**
Security Best Practices (spec 2026-07-28)
https://modelcontextprotocol.io/docs/2026-07-28/tutorials/security/security_best_practices

11. **AWS**
What Is Headless CMS?
https://aws.amazon.com/what-is/headless-cms/

12. **Strapi**
Exploring Strapi's Architecture
https://strapi.io/blog/strapi-architecture

13. **Focus Reactive**
Headless CMS Architecture for Multisite and Modern Frontends
https://focusreactive.com/blog/headless-cms-architecture/

14. **The New Stack**
15 Best Practices for Building MCP Servers in Production
https://thenewstack.io/15-best-practices-for-building-mcp-servers-in-production/

15. **Tishan David**
Shipping production MCP servers without the footguns
https://news.tishandavid.com/articles/production-mcp-servers

16. **Hacker News**
Neon Serverless Postgres is generally available, 2024-04-15
https://news.ycombinator.com/item?id=40040593