Episode Book

EPISODE #5912

Building Your Own CRM with AI: Where to Start

PUBLISHED	12 October 2026 at 17:58 UTC
DURATION	22:12
LISTEN	myweirdprompts.com/episode/ai-assisted-crm-schema-first-37758/
HOSTS	Corn, Herman
SCRIPT MODEL	deepseek-flash
VOICES	chatterbox-regular (chatterbox-local)
PIPELINE	V5.4
SOURCES	13 cited by the research (listed at the end)
BOOK GENERATED	19 October 2026 at 17:41 UTC

SYNOPSIS

Rita runs a two-person landscape design practice and wants a real backend instead of spreadsheets and rented CRMs. Can AI-assisted development make that realistic? This episode digs into the backbone: the database and architecture. It covers why the schema is the real moat, how to map a garden design practice onto the canonical CRM entities, and why a rigid schema with fast migrations beats a flexible one. Then it walks through the boring-but-correct stack — Cloud Run, Cloud SQL Postgres, Terraform — and the three things that actually break: connection pooling arithmetic, migration races at container start, and secrets management. Part one of a three-part series on building your own tools.

TRANSCRIPT



Corn

Rita's question is worth taking slowly, which is convenient, because it's the only speed I have.



Herman

You've been saving that one.



Corn

Six weeks. So here's where we are. Rita and one other designer run a landscape architecture and garden design practice. Google Workspace for everything, spreadsheets and folders and the usual manual duct tape holding a small business together. She wants a real backend. Clients, leads, projects, suppliers. And the actual question underneath it is whether AI-assisted development has finally made it realistic for two people to build and maintain their own software instead of renting a CRM. She's not asking for a weekend toy. She's asking for something durable.



Herman

And when this came up before, the recommendation was Postgres, Python and FastAPI on Cloud Run, integrations into Workspace. Explicitly not Firestore, because her data is relational. Clients have projects, projects have suppliers, suppliers have materials. That's a graph whether you admit it or not.



Corn

Daniel's split it into three episodes. This one is the backbone. The database, the architecture, what she builds first. Part two is the Workspace integration, part three is external services.



Herman

Right. So let's start with the part everybody skips, which is whether she should build at all.



Corn

The build versus buy question.



Herman

Retool surveyed eight hundred and seventeen builders for their twenty twenty-six report, and thirty-five percent of teams have already replaced at least one SaaS tool with something they built themselves. Seventy-eight percent expect to build more custom tools this year. CRMs and form builders are the categories under the most replacement pressure, at twenty-five percent.



Corn

That's teams though. A team has engineers.



Herman

Some of them do. And here's the number that matters for Rita. Only thirty-one percent of those builders prompt their way to a complete app. Seventy-two percent use AI to write discrete pieces of code that they integrate and test themselves. Eight percent use AI-generated code without changes. Forty-four percent test thoroughly before anything goes near production.



Corn

So the fantasy is prompt, receive CRM, put it in the cloud.



Herman

The fantasy is real for demos and dies at the first edge case.



Corn

I've watched you break your own setup with every tool that ships. You are the edge case.



Herman

I am an enthusiastic edge case, and I test in production, which is why nobody should copy me. But the point stands. The reason the build case works now isn't that AI writes the app for you. It's that it removes the friction on the parts that used to eat the time. Boilerplate, wiring, the tedious glue. You still own the design and the testing.



Corn

What's the price comparison look like?



Herman

Traditional route, agency or in-house custom CRM, thirty to a hundred and twenty thousand dollars over three to nine months. AI-assisted equivalents run two hundred to twenty-five thousand, one-time. Timelines compressed by sixty to eighty percent on the high end.



Corn

So the low end of that is a weekend and the high end is a used car.



Herman

And the broad middle is where Rita actually lives, which is she does some of it herself and pays for nothing except her own time.



Corn

Here's my problem with the entire build-versus-buy framing, though. The argument for building is never the code. It's the shape of the data.



Herman

Say more.



Corn

Buying HubSpot means buying HubSpot's idea of what a customer is. Deals, accounts, pipelines, stages. Rita doesn't have deals. She has a garden. She has a client who wants a courtyard redesigned and a supplier who can get the right stone in three weeks. That's not a sales motion, that's a project.



Herman

That's the strongest argument, and it comes from the schema side. A custom CRM's real value is a schema designed around your actual work. Deals and accounts for a sales team. Projects and suppliers for a design practice. And that's the part AI can't guess for her, because it doesn't know how her business runs. It'll happily give you the Salesforce schema back.



Corn

The schema is the moat, the code is the drawbridge.



Herman

I'm stealing that.



Corn

You're welcome. So what does the schema actually look like?



Herman

Start with the canonical CRM. Six entities. Lead, Contact, Account, Deal, Activity, User. Every commercial CRM is built on those six, from Salesforce down. For Rita, Deal maps to Project and Account maps to Client, and the rest transfer directly.



Corn

So she's not inventing a new model.



Herman

She's relabelling a well-worn one. And the design order matters. List entities first, fields second. People jump straight to fields and end up with a table that can't express anything.



Corn

Guilty. Every spreadsheet I've ever built started with a column nobody ever used.



Herman

Then define your primary keys explicitly. UUIDs if you're ever going to sync across systems, plain integers are fine if it's single-tenant, which it is for Rita. Map your foreign keys before you touch fields, because that's where the relationships live. Then trim fields ruthlessly. Every field you add is a field somebody maintains forever.



Corn

And then the thing nobody does.



Herman

Add created_at, updated_at and deleted_at on day one. Day one. Not when you need them.



Corn

Because if you need them later, you need them retroactively, and you don't have them retroactively.



Herman

Right. And test with fifty to a hundred sample records before you build a single workflow. If the schema can't hold a hundred fake gardens, it won't hold a real one.



Corn

Is there something Rita can copy that isn't a blank page?



Herman

There's a boilerplate that's almost eerily close. It's a project management schema written for web design studios. Clients to projects to tasks to time entries to invoices. And the project status field is a check constraint with a fixed list. Discovery, proposal, active, review, delivered, archived.



Corn

That's a design consultancy. That's the whole shape of her business.



Herman

Swap in the actual values Rita uses, and the tables don't change. A web design studio and a garden design studio have the same bones. Client, project, deliverable, invoice.



Corn

Where do suppliers fit? Because that's the part that isn't in the web design template.



Herman

A simple suppliers table, plus a junction to projects and materials. There's a field report schema for electricians that does this well. Materials carry a supplier field and a unit cost and a total cost. So a project points at materials, materials point at a supplier, and you can answer the question that matters. What did this garden cost me, and who did I buy it from?



Corn

That's the query Rita actually wants. Not a dashboard. A number.



Herman

And that's the natural stopping point for the schema. Because the temptation is to build a flexible attribute system. Custom fields, definitions, a JSONB column that can hold anything. The Attio and Notion pattern.



Corn

Resist.



Herman

Resist hard. That's a schema for a product with customers who all want different things. Rita has two users. There's exactly one sales motion in the building. If she needs a new field, she adds a column on a Tuesday.



Corn

You add a column. She watches you add a column.



Herman

I add the column, the migration runs, and we've spent four minutes. The flexible system costs you a year of complexity to avoid four minutes.



Corn

Okay, but let me push on that, because I've heard the other side of this argument and it's not stupid. What if the practice grows? What if she hires a third designer, or starts doing commercial work with a completely different project lifecycle, or wants to track plant warranties separately from hardscape warranties? Doesn't the rigid schema become the thing that traps her?



Herman

It becomes the thing she changes. And changing it is the four-minute migration. The trap isn't the rigid schema, it's the belief that you can predict which fields you'll need in three years. You can't. Nobody can. So you optimize for the cost of change instead of trying to



Herman (continued)

avoid change. A rigid schema with a fast migration process is more flexible in practice than a flexible schema, because the flexible schema makes every query harder forever in exchange for a change you might never make.



Corn

So the flexibility is a liability you're paying interest on.



Herman

Every single day. And here's the thing about the JSONB column specifically. The moment you put data in there, you've lost your types, you've lost your constraints, you've lost the database's ability to tell you when you've done something wrong. You've traded a schema for a convention, and conventions get forgotten.



Corn

So the schema is the spec.



Herman

The schema is the spec. Everything else flows from it. Routes, forms, views. If you get the tables right, the rest is typing.



Corn

And that gets us to where the tables live, and I want to preface this by saying I know where this is going and I still don't like it.



Herman

The architecture question. The boring default, and I mean that as a compliment. Cloud Run for the application. Cloud SQL running Postgres sixteen. The Cloud SQL Auth Proxy as a sidecar, IAM-authenticated, no public IP. Secret Manager for credentials. Cloud Build and Artifact Registry for the images. Terraform to describe all of it in code. That's the stack for anything under about a thousand requests a second.



Corn

That's a lot of nouns for two people who design gardens.



Herman

It's four services and a proxy. The frame it's sitting in is Terraform, and that's one file to start.



Corn

Fine. What breaks?



Herman

Connection pooling. This is the classic one. Cloud SQL's max connections is fixed by the instance tier. Small instance, about a hundred connections. Cloud Run scales horizontally under load. If it spins up a hundred instances and each one opens a pool of ten connections, you're at a thousand and the database refuses them all.



Corn

And the app interprets that as the database being down.



Herman

Because from the app's perspective, it is. The fix is arithmetic. Size the connection pool in your application to the max connections divided by the max instances. Usually that's two to five, not ten. If you need more headroom, put PgBouncer in transaction mode in front of it.



Corn

That's a second thing to run.



Herman

It is. And one caveat. Prepared statements don't survive PgBouncer's transaction mode. So if you add it later, that's a code change, not just a config change.



Corn

Noted. What else breaks?



Herman

Migrations. Never, ever run migrations at container start.



Corn

Why?



Herman

Because N containers start at once and all of them race to alter the same table. Some of them will be talking to a schema that half-exists. It's a coin flip with your data on it.



Corn

So where does it go?



Herman

A Cloud Run Job. It runs the migration command once, completes, and exits. You deploy that job first, wait for it to finish, then deploy the service. Same pattern works in CI with Sqitch or Flyway. And use expand-contract migrations. Add the new column and backfill it in one deploy, switch the code over in the next, drop the old column in the third. Never combine a schema change and a code change in one pull request.



Corn

Because if the deploy halfway fails, you've got code expecting a column that doesn't exist.



Herman

Or a column nothing writes to. Both are worse than a failed deploy.



Corn

And secrets?



Herman

Secret Manager, referenced from the service config. Never baked into an image, never committed to Git. That's non-negotiable and it's also the easiest thing in the whole stack to get right.



Corn

So this is all reasonable. It's also, on the face of it, about fifty dollars a month before Rita has a single client.



Herman

That's the interesting part. And this is where the two halves of the bill diverge completely.



Corn

Cloud Run is serverless.



Herman

Cloud Run scales to zero. If nobody's using it, there are no instances and you're not billed. There's a free tier that covers two million requests a month, a hundred and eighty thousand vCPU-seconds, three hundred and sixty thousand GiB-seconds. In practice about nine hundred thousand requests at normal memory and latency.



Corn

And Cloud SQL?



Herman

Cloud SQL has no free tier and it never scales to zero. It bills by the hour whether it's serving a million queries or nothing at all. The smallest useful instance, one vCPU, three point seven five gigs of RAM, ten gigs of SSD, is about fifty dollars and fifty-one cents a month.



Corn

Whether she uses it or not.



Herman

High availability doubles the compute and memory, so about ninety-nine eighty-two.



Corn

So if I understand this correctly, and I want to be wrong, the app costs nothing and the database costs everything.



Herman

Cloud Run serving a million requests is about forty-eight cents. The database sitting next to it is about fifty-one dollars. That's roughly a hundred to one.



Corn

The gardening is free. The filing cabinet costs fifty dollars a month.



Herman

And that's the correct way to think about it. The cost is not in serving traffic. It's in the things that run continuously whether or not anyone is using them.



Corn

So the obvious question. Is Cloud SQL the wrong starting point?



Herman

There's a real alternative and it's worth taking seriously. Neon.



Corn

The serverless Postgres.



Herman

Neon's free plan is free, permanently, not a trial. No credit card. A hundred projects, one gigabyte of storage per project, a hundred compute unit hours per project per month, autoscaling up to two compute units, and it scales to zero after five minutes of inactivity. Five gigabytes of egress.



Corn

So it does the thing Cloud SQL won't do.



Herman

It scales to zero. If Rita's asleep, or if Rita's asleep and I mean that in the sloth sense, the database costs nothing.



Corn

I object to being the unit of measurement.



Herman

The paid tier, Neon Launch, is pay as you go with no monthly minimum. About eleven cents per compute unit hour, thirty-five cents per gigabyte month of storage, autoscaling up to sixteen compute units, five hundred gigabytes of egress included. Invoices under fifty cents aren't even collected. And upgrading from free to paid requires no data migration, same architecture, same connection strings.



Corn

What's the catch?



Herman

The reception is split, and this is where I'd be careful with Rita. There are people who love it. One developer called it by far the best serverless database he's used. Another migrated thousands of databases onto it and said it just worked.



Corn

And the other half.



Herman

Glitchy. Slow. Laggy. One user said the pricing was too high. Another said it was so unreliable that Neon removed the percentages from their status page.



Corn

That's a striking thing for a company to do.



Herman

They admitted it. Neon's own engineers said the status page calculation wasn't right, and pointed at twelve significant outages in a two and a half month stretch back in late twenty twenty-three. Their response since is that they now measure projects under five minutes of downtime across seven hundred thousand databases.



Corn

So the argument isn't 'Neon is bad.' It's 'Neon had a bad stretch and has been loud about fixing it.'



Herman

And it's still a young company with a shorter track record than Google's managed Postgres. Cold starts are also real. If the database has scaled to zero and Rita opens the app at seven in the morning, something has to wake up.



Corn

How long?



Herman

I don't have a good number for that and I don't want to invent one. It's noticeable enough that people write about it. What I'd say is that if the app is going to be open all day, the database stays warm and the cold start never happens. It's the first query of the morning that stings.



Corn

And that's a two-person practice. The first query of the morning is Rita opening her laptop with coffee.



Herman

Which is exactly the moment you don't want to wait.



Corn

So what's the actual decision?



Herman

For a business whose data is its livelihood, cheap but flaky versus expensive but boring is the real question. And I'd answer it with a sequence rather than a verdict. Start on Neon Free. It costs nothing, it's plain Postgres, she'll learn the schema and the queries against it, and if it annoys her, she moves.



Corn

And move to what?



Herman

Cloud SQL once there's revenue. Once fifty dollars a month is a rounding error against what the practice bills, boring wins, because boring means a managed Postgres with point-in-time recovery that takes minutes instead of hours.



Corn

And the migration itself?



Herman

Both are plain Postgres. The schema moves. The SQL moves. A dump and restore and an afternoon of testing connection strings. The abstraction to actually avoid here is Firebase Data Connect, which does offer Postgres but wraps it in another layer Rita doesn't need.



Corn

That was the original advice. Don't fix what isn't broken.



Herman

Don't fix what isn't broken with a wrapper.



Corn

So say she does all this. Say the schema is right and the database is live. How does it not become another folder of fragile scripts?



Herman

Four things and none of them are architecture. Git, with everything in it, including the infrastructure definition. Migrations under version control and run by a job, not by a person who remembers. Secret Manager instead of a text file on a laptop. And an actual backup you have restored at least once.



Corn

Say that last one again.



Herman

An untested backup is not a backup. It's a hope. Everybody agrees with that sentence and almost nobody does the restore test.



Corn

I want to put a number on that, because I think people hear it and nod and then do nothing. How long does a restore test actually take?



Herman

For a database this size, under an hour. You spin up a scratch instance, you restore the dump, you run three queries, you confirm the row counts match, you tear it down. That's it. Once a quarter. And the first time you do it, you'll find something wrong. A permission, a missing extension, a connection string that only exists on one machine. Better to find that on a Tuesday afternoon than during an actual incident.



Corn

And the portability question. She's a small business, she's allowed to be nervous about lock-in.



Herman

This is the good news. Postgres is Postgres. The schema, the queries, the data, all of it moves between Neon, Cloud SQL and a box under her desk. What she shouldn't do is write the app against a proprietary data API and then discover she's married to it.



Corn

And if it becomes a real application with a real bill, what keeps the bill from running away?



Herman

A few guardrails. Set minimum instances to zero on anything that isn't latency critical. One always-on instance is about six dollars and fifty cents a month in CPU alone, and that's the number that creeps. Cap the maximum instances on Cloud Run, which is the single most effective cost control there is. Don't enable high availability until there are paying customers. Stop non-production instances when nobody's using them.



Corn

And watch storage.



Herman

Watch storage autogrow, because it can't be shrunk without recreating the instance. That's the one that surprises people a year in.



Corn

Which lands us on the practical question. What does Rita build first?



Herman

The schema. Nothing else. Get the tables and the relationships on paper, with the status constraints filled in, and walk through a hundred fictional jobs in her head. If the schema holds a hundred imaginary gardens, it'll hold hers.



Corn

And then?



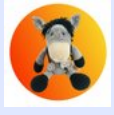
Herman

Then a minimal FastAPI service with two or three endpoints. Create a client. Create a project. List the projects for a client. A Neon Free database. A migration job that runs the schema into it. That's the whole build. It should take a weekend, and when it's done she has a real service talking to a real database.



Corn

What does she defer?



Herman

Authentication complexity, for one. Two users, one shared login, done. The frontend, for another, because a JSON response in a browser tab is a perfectly good interface at this stage. And the entire Workspace integration, which is part two of this.



Corn

And the recurring bill for two users?



Herman

Essentially nothing. Neon Free and the Cloud Run free tier will carry a two-person practice until something changes the math. The first real dollar probably comes from storage, not traffic.



Corn

Which is a strange place to end up. The database everyone warns you about is the one that costs nothing, and the database everyone recommends is the one that costs fifty dollars a month to sit idle.



Herman

That's the whole tension of this episode in one sentence. Boring is expensive because boring is always on. Cheap is risky because cheap is sometimes off. And the correct answer for Rita is to start cheap and buy boring when the practice can afford it.



Corn

The schema is the moat.



Herman

The code is the drawbridge.



Corn

The filing cabinet is the rent.

SOURCES

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

1. **Neon, Neon Pricing Plans , current as of 2026. [primary]**
   https://neon.com/pricing.md

2. **Google Cloud, Cloud Run pricing . [primary]**
   https://cloud.google.com/run/pricing

3. **Google Cloud, Cloud SQL pricing . [primary]**
   https://cloud.google.com/sql/pricing

4. **Nightsong, Cheapest Way to Run a Database on GCP, last reviewed 2026-09-19.**
   https://www.nightsongcloud.com/gcp-cheap-database

5. **Nightsong, What a Small App Actually Costs on Google Cloud, last reviewed 2026-09-19.**
   https://www.nightsongcloud.com/gcp-what-it-actually-costs

6. **Neon, 100 projects for free, 1 GB each, 2026-02.**
   https://neon.com/blog/neon-free-plan-1-gb-per-project

7. **Retool, The build vs. buy shift, 2026 (survey late 2025).**
   https://retool.com/blog/ai-build-vs-buy-report-2026

8. **Islam Gamal, Cloud Run + Cloud SQL: The Boring Blueprint, 2026-03-14.**
   https://islamgamal.com/blog/cloud-run-and-cloud-sql-blueprint

9. **Fuzen, CRM Database Examples: 7 Schemas, published 2026-04-07, updated 2026-06-15.**
   https://www.fuzen.io/posts/internal-crm-database-examples

10. **Flask Vibe, Free PostgreSQL Schema Boilerplates for Vibe Coding.**
    https://www.flaskvibe.com/tools/postgres-schema-boilerplates

11. **Hacker News, Neon Serverless Postgres is generally available, 2024-04-15 (217 comments).**
    https://news.ycombinator.com/item?id=40040593

12. **toolchew, Neon in 2026**
    serverless Postgres one year in review.
    https://toolchew.com/en/review-neon-2026/

13. **devshot, Neon Review 2026, updated 2026-09-25.**
    https://devshot.co/reviews/neon-review