

[Generative AI \(/tags/generative-ai\)](#) [Claude Code \(/tags/clause-code\)](#) [Anthropic \(/tags/anthropic\)](#)
[AI-assisted Coding \(/tags/ai-assisted-coding\)](#)

Building an Event Management System with Claude Code: Part 5 - Database Schema and Natural Language Queries

Posted by nielsb on Sunday, January 4, 2026

Welcome back to the **Building an Event Management System with Claude Code** series! In Part 4, we built the infrastructure foundation: a properly structured GitHub repository, a production PostgreSQL database on Neon, and Postgres MCP Pro configured for Claude Desktop. Today, we bring that database to life through conversational schema design and our first natural language queries—using both Claude Code for development and Claude Desktop for querying.

- To see all posts in the series, go to: **Building an Event Management System with Claude Code** (/contact-event-clause-code).

If you've been following this series from the beginning, you know we've already made one major pivot. In Parts 1 and 2, I planned to build a traditional web application with forms and dashboards, using Claude Code to write the code faster. Then came Part 3's revelation: why make yet another CRUD application when we could create something fundamentally different? An AI-native system where the primary interface is conversation, not forms.

Another Plot Twist

Today, we're taking another evolutionary step in the same direction.

The traditional approach, even in AI-assisted development, would be to:

1. Ask Claude Code to generate SQL `CREATE TABLE` statements
2. Copy those statements to a file
3. Manually run the file against the database using `psql` or a SQL client
4. Ask Claude Code to generate `INSERT` statements for test data
5. Copy and run those too
6. Write `SELECT` queries to verify everything worked

But here's what I realized while preparing this post: **if we're building an AI-native system, why is the database setup process still manual?** Why am I copy-pasting SQL like it's 2015?

So today, we're not just using Claude Code to **generate** SQL scripts. We're using Claude Code to **design the schema through conversation, create the tables directly in Neon via MCP, generate realistic test data, and query everything using natural language**. No copy-paste. No manual execution. Pure conversation.

Here's the key insight: we now have **two conversational interfaces** working together:

- **Claude Code** (development tool): Designs, creates, and populates the database via MCP
- **Claude Desktop** (query interface): Lets us query and analyze the data in natural language

This is the continuation of Part 3's paradigm shift, but applied to database development itself. Just as we moved from "web forms" to "conversational queries" for the end-user experience, we're now moving from "SQL scripts" to "conversational schema design" for the development experience.

Let's see what that actually looks like in practice.

The Paradigm Shift

In Parts three and four, I mentioned we were building an AI-native system. Today, you'll see what that actually means in practice—and how Claude Code and Claude Desktop work together to create a completely new development and usage workflow.

What we're NOT doing:

- Write SQL scripts in a text editor
- Manually run those scripts against the database
- Write separate scripts for test data
- Hand-craft queries to verify everything worked

Instead, we're using a dual-interface approach:

Claude Code (our AI development partner) will:

- Design the database schema based on our requirements

- CATALOG

- [Another Plot Twist](#)
- [The Paradigm Shift](#)
- [What We'll Accomplish T...](#)
- [Connecting Claude Code...](#)
- [Understanding the Confi...](#)
- [Claude Code Configurati...](#)
- [System Environment Vari...](#)
- [Testing the MCP Connect...](#)
- [Conversational Schema ...](#)
- [The Design Conversation](#)
- [Creating the Schema via ...](#)
- [The Creation Process](#)
- [Generating Migration Scr...](#)
- [Reflecting on the Experie...](#)
- [Generating Test Data](#)
- [The Data Generation Con...](#)
- [Loading the Test Data](#)
- [Verifying the Test Data](#)
- [The Dual Interface Advan...](#)
- [Claude Code: The Develo...](#)
- [Claude Desktop: The Qu...](#)
- [Natural Language Querie...](#)
- [Simple Queries](#)
- [Complex Analytical Quer...](#)
- [Cohort Retention Analysis](#)
- [What We've Accomplished](#)
- [Infrastructure Built](#)
- [Data Generated](#)
- [Capabilities Demonstrated](#)
- [Time Investment](#)
- [The Paradigm Shift Reali...](#)
- [What's Next: Part 6 Prev...](#)
- [What We'll Build](#)
- [Example of What's Coming](#)
- [Summary: From SQL to ...](#)
- [~ Finally](#)
- [Your Turn](#)

- Create the tables directly in our Neon database via MCP
- Generate and insert realistic test data
- Generate migration scripts for version control
- Create comprehensive documentation

Claude Desktop (our AI query interface) will:

- Query the database using natural language
- Perform complex analytics through conversation
- Provide insights without writing SQL
- Verify the schema and data Claude Code created

This is the transformation from **coding** to **conversing**, applied to both development (Claude Code) and usage (Claude Desktop). Together, they form a complete AI-native data workflow.

What We'll Accomplish Today

By the end of this post, you'll have:

- ✓ Claude Code configured to use Postgres MCP Pro for database operations
- ✓ A complete database schema created conversationally by Claude Code
- ✓ Realistic test data for 3 years of events, generated and inserted by Claude Code
- ✓ Migration scripts and documentation automatically generated
- ✓ Natural language database queries working in Claude Desktop
- ✓ A complete AI-native workflow from development to usage
- ✓ Documentation generated automatically

Let's begin.

Connecting Claude Code to Postgres MCP Pro

Understanding the Configuration Difference

In Part 4, we configured Postgres MCP Pro for **Claude Desktop** (the GUI application). This allows us to query our database through natural language conversations. But now we need to set up **Claude Code** to use the same MCP server—this will enable Claude Code to actually create tables, insert data, and manage the database through conversation.

Why do we need both?

- **Claude Desktop**: Perfect for querying and analyzing data (what business users do)
- **Claude Code**: Perfect for creating schemas and managing the database (what developers do)

Both use the same MCP server but serve different purposes in our AI-native workflow.

Claude Desktop Configuration:

- Location: %APPDATA%\Roaming\Claude\claude_desktop_config.json (Windows)
- Purpose: Interactive UI conversations
- Status: ✓ Already configured in Part 4

So, how do we configure Claude Code?

Claude Code Configuration - The Conversational Approach

Above, I (rhetorically) asked how to configure Claude Code to use Postgres MCP Pro. By now, you should know the answer: **we simply ask Claude Code to help us configure it**. This is AI-native development, even the setup is conversational.

Let's open Claude Code:

```
1 $ cd event-contact-management-2
2 $ claude --continue
```

Code Snippet 1:: Opening Claude Code with --continue Flag

Since I had closed down Claude Code after Part 4, I reopened it in the project directory with the `--continue` flag, as you see in *Code Snippet 1*. This tells Claude Code to resume our previous conversation in this directory, preserving all project context and “memory” of what we've been building.

With Claude Code running, I asked it about configuring MCP:

```
Up until now, we have assumed that you will create the scripts for the database tables, test data.
For that, we need to configure the PostgreSQL MCP server so you can use it. What options are avai...
```

Code Snippet 2:: Claude Code's Prompt re. MCP Server

Notice what's happening here: I'm not googling “how to configure MCP for Claude Code” or reading documentation. I'm **asking Claude Code itself** to explain the configuration options and help me set it up. This is the AI-native development workflow—your development tool is also your technical advisor.

Based on the prompt in *Code Snippet 2*, Claude Code responded with a detailed explanation of the configuration locations and the actual configuration needed for our project:

• Great thinking! Yes, if I can access the Postgres MCP Pro server directly, I can create the database schema.

Let me check the official documentation on how to configure MCP servers for Claude Code:

[snipped for brevity]

• Perfect question! Here are your options for configuring MCP servers for Claude Code:

MCP Server Configuration Locations

1. Project Scope (Recommended for You) ★

Location: .mcp.json in your project root
Example: W:\OneDrive\repos\niels\event-contact-management-2\.mcp.json

Advantages:

- ✓ Version controlled (committed to Git)
- ✓ Shared with team members
- ✓ Project-specific configuration
- ✓ Uses environment variables for credentials (keeps secrets out of Git)

[snipped for brevity]

2. Local Scope (Personal Override)

[snipped for brevity]

3. User Scope (Global)

[snipped for brevity]

Recommended Approach for Your Project

Create a project-scoped .mcp.json file that:

1. ✓ Can be committed to Git (version controlled)
2. ✓ Uses environment variables for credentials (secure)
3. ✓ Works for both you and future team members

Option A: Simple Approach (Credentials in File)

Create W:\OneDrive\repos\niels\event-contact-management-2\.mcp.json:

[snipped for brevity]

⚠️ Important: Add .mcp.json to .gitignore if using this approach!

Option B: Secure Approach (Environment Variables) ★ Recommended

[snipped for brevity]

Safe to commit to Git! ✓

Then set environment variable:
Windows (PowerShell)
\$env:NEON_DATABASE_URL = "postgresql://neondb_owner:*****@ep-raspy-sun-a9vr1wf1-pooler.gu...

Or add to your system environment variables permanently

How This Will Work

Once configured:

[snipped for brevity]

Next Steps

Would you like me to:

1. Create the .mcp.json configuration file for the project?
2. Use Option A (simple) or Option B (secure with env vars)?
3. Then test the connection to verify I can access your Neon database?

Let me know which approach you prefer, and I'll set it up!

Code Snippet 3:: Claude Code Explaining MCP Configuration Options

As you can see in *Code Snippet 3*, Claude Code didn't just tell me where to put a config file, it:

- Explained the three configuration scopes available
- Recommended the best approach for our use case (project-scoped with env vars)
- Explained the security implications of each option
- Offered to create the configuration file for us
- Asked clarifying questions about our preference

This is the power of conversational development: the tool doesn't just execute commands, it advises, explains, and guides you through decisions.

I asked Claude Code to create a project-scoped configuration file using the secure approach with environment variables. Claude Code then created the .mcp.json file, configured it properly, and verified the connection to our Neon database, all through conversation. It also also created an MCP-SETUP.md file documenting the steps taken (another great practice for future reference). However:

⚠️ WARNING

The MCP-SETUP.md should be committed to version control, and Claude Code wanted to do that. Just by pure chance I had a look in the file before it committed, and I noticed it contained the actual database URL with username and password in plain text. The warning here is to always review auto-generated files before committing them to version control, especially when they relate to configuration or secrets.

I asked Claude Code to remove sensitive information from `MCP-SETUP.md` before committing, and it did so without issue.

System Environment Variable

I told Claude Code that I wanted the secure approach with environment variables. As mentioned, Claude Code documented everything in `MCP-SETUP.md`, including how to set the system environment variable for the database URL. However, I do not want to do it manually; I want Claude Code to do it. So I asked it to do it. The response was:

```

• I'll set the system environment variable for you using PowerShell. This will set it permanently
[snipped for brevity]

• Bash(powershell.exe -Command "[System.Environment]::SetEnvironmentVariable('NEON_DATABASE_URL',
    'postgresql://neondb_owner:*****@ep-raspy-sun-a9vrlwf1-poo...')
    L (No content)

• Perfect! Now let me verify it was set correctly:
[snipped for brevity]

• Bash(powershell.exe -Command "[System.Environment]::GetEnvironmentVariable('NEON_DATABASE_URL',
    L postgresql://neondb_owner:*****@ep-raspy-sun-a9vrlwf1-pooler.gwc.azure.neon.tech/ev
    lmode=require

• Excellent! The environment variable has been set successfully. I can confirm it's stored:
[snipped for brevity]

According to the MCP-SETUP.md documentation:

1. Restart Required: You need to restart your terminal/IDE for the environment variable to be av
    - Close all terminal windows
    - Close Claude Code (if running)
    - Reopen your terminal/IDE
    - The environment variable will now be available to Claude Code
2. Verify Docker is Running: Make sure Docker Desktop is running since Postgres MCP Pro runs via
3. Test the Connection: Once you've restarted, we can test that Claude Code can access the databa

The environment variable is now permanently set for your user account and will persist across re

```

Code Snippet 4:: Claude Code Setting System Environment Variable

Claude Code successfully set the system environment variable for the database URL. Not that I don't trust Claude Code, but I always like to verify these things myself. So I "spun up" a new PowerShell window and ran:

```
1 PS> [System.Environment]::GetEnvironmentVariable('NEON_DATABASE_URL', 'User')
```

Code Snippet 5:: Verifying System Environment Variable in PowerShell

The output confirmed that the environment variable was set correctly.

Testing the MCP Connection

With the MCP configuration in place and the environment variable set, I asked Claude Code to test the connection to the Neon database, and it responded:

```

• I'll test the connection to your Neon PostgreSQL database through the Postgres MCP Pro server. I
[snipped for brevity]

• Excellent! The connection is successful! I can confirm:
    ✓ Connected to PostgreSQL 17.7 on your Neon database
    ✓ SSL connection working (sslmode=require)
    ✓ Credentials are correct

Now let me check what tables currently exist in your database:
[snipped for brevity]

• Perfect! The connection test is successful:
Connection Test Results:
- ✓ Database: PostgreSQL 17.7 on Neon cloud
- ✓ Connection: Working with SSL
- ✓ Authentication: Successful
- ✓ Current state: Empty database (no tables yet)

The database is ready and waiting for the schema to be created. According to the project plan, t
- contacts - Unified person records (attendees + speakers)
- events - Event definitions
- participation - Registration and attendance tracking
- speaker_sessions - Speaker presentations
- speaker_ratings - Session ratings
- email_log - Communication history

Would you like me to create the database schema now?

```

Code Snippet 6:: Claude Code Successfully Testing MCP Connection

Cool, it looks like we are set now.

Conversational Schema Design with Claude Code

Now comes the exciting part. Instead of writing SQL CREATE TABLE statements manually, we're going to **design the database schema through conversation** with Claude Code.

The Design Conversation

Here's what traditional development looks like:

1. Open a SQL editor or text file
2. Write CREATE TABLE statements
3. Think through relationships and constraints
4. Add indexes manually
5. Document the schema separately
6. Save to a .sql file

Here's what AI-native development with Claude Code looks like:

1. Describe what you need in plain English
2. Claude Code designs the schema
3. Review and refine through conversation
4. Claude Code creates the tables via MCP
5. Claude Code generates documentation automatically

Let's see it in action. I started by describing our requirements to Claude Code:

```
Based on our architecture plan (in CLAUDE.md and plan.md), you do know the high-level requirements:
- Typically, we'd uniquely identify a contact based on email address. However, we are capturing contacts
- We will have multiple events per year.
- For participants, one person may register for multiple tickets (e.g., one for themselves and one for their colleague).
- We rate speakers on the topic(s) and the presentation separately.
- A speaker can both be an attendant and a speaker at the same event.

Think hard about the relationships, indices, and constraints we'll need. However, try to keep it simple.
```

Code Snippet 7:: Claude Code's Prompt for Schema Design

Notice in *Code Snippet 7* how I use "Think hard" to trigger Claude Code's deeper reasoning mode (as we discussed in Part 2).

Notice what I'm NOT doing:

- ✗ Writing SQL
- ✗ Googling best practices for database design
- ✗ Manually planning foreign keys
- ✗ Deciding on data types

Instead, I'm **describing the business requirements** and letting Claude Code translate them into a technical schema design.

Why this matters:

- **Domain experts** can participate in schema design without knowing SQL
- **Design decisions** are explained and documented automatically
- **Best practices** are applied consistently (Claude Code knows PostgreSQL patterns)
- **Iteration** is easy, just have a conversation to refine the design

Claude Code went off to think:

```
I'll design the database schema based on the requirements. Let me think through the relationships and constraints.

Database Schema Design

Core Entities & Relationships

Key Design Decisions:
1. Email is NOT unique - Use auto-increment ID as primary identifier
2. Ticket-level tracking - Separate purchaser from attendee to handle "register for colleague"
3. Dual roles supported - Same contact can be both attendee (via ticket) and speaker at same event
4. Simple rating model - Two numeric ratings per session (topic + presentation)

[snipped for brevity proposed tables, etc]

Constraints & Business Rules

✓ Email NOT unique (family email sharing)
✓ Ticket number IS unique (Quicket requirement)
✓ One person can buy multiple tickets
✓ Ticket purchaser ≠ attendee (register for colleague)
✓ Speaker can also have ticket (dual role)
✓ Multiple sessions per speaker per event
✓ Multiple ratings per session (different sources)
✓ Topic and presentation rated separately

---
What do you think? Any adjustments needed before I create this schema?
```

Code Snippet 8:: Claude Code Thinking Through Schema Design

What you see in *Code Snippet 8* is a very brief excerpt of Claude Code's deep thinking process. It analyzed the requirements, identified core entities and relationships, and outlined key design decisions and constraints. I had a look at the proposal and saw some flaws, which I pointed out.

```
This looks almost perfect. However, I realised that on the day of the event, we will have walk-ins.
```

Code Snippet 9:: User Requesting Schema Adjustment

Claude Code went off and came back with a suggestion, which I didn't like. So we had another round of back-and-forths around the ticket number, plus other minor adjustments:

Me: No, I also realised that the ticket number may not be globally unique, but only unique per event.

Claude Code responds.

Me: That looks good, We also need to capture what type a contact is: attendee, speaker, sponsor, etc.

Claude Code responds: Great point! A contact can have multiple roles. Let me present two options:

- Option 1: Array column (Simple & Practical) ★ Recommended
 - Some pro's and con's of option 1
- Option 2: Junction table (Event-specific roles)
 - Some pro's and con's of option 2

Me: I hear what you are saying but I prefer option 2, a junction table with one to many relationships.

Claude Code responds: Perfect! Option 2 it is. Here's the complete contact_roles table design:

- Should I now create the complete SQL schema file that we can review and then execute against the database?

Me: Can you please create the complete SQL schema file for us to review. Do NOT execute against the database.

Code Snippet 10:: Refining Schema Design Through Conversation

When you look at *Code Snippet 10*, you see how the schema design evolved through conversation. I identified issues, Claude Code proposed solutions, and we iterated until we had a solid design. This back-and-forth is crucial to ensure the schema meets all requirements before creation.

After my “Can you please create the complete SQL schema ...” prompt, Claude Code responded with the full schema design and asked whether I wanted it to generate the actual SQL file next. I had a look at the proposed schema and told Claude Code to proceed with generating the SQL file. Claude Code responded:

- I'll create the complete SQL schema file for review.
- Write(database\schema.sql)

[snipped for brevity]

- Perfect! I've created the complete SQL schema file at database/schema.sql.

Schema Summary

7 Tables Created:

1. contacts - Person records with email (NOT unique)
2. events - Event definitions (multiple per year supported)
3. tickets - Ticket-level tracking (nullable ticket_number, defaults to -1)
4. speaker_sessions - Speaker presentations
5. speaker_ratings - Separate topic and presentation ratings
6. email_log - Email send history
7. contact_roles - Many-to-many contact role assignments

Key Features Implemented:

- ✓ Email NOT unique (shared family emails)
- ✓ Ticket-level tracking with walk-in support
- ✓ ticket_number defaults to -1, unique per (event, attendee)
- ✓ Dual roles (same person can be attendee and speaker)
- ✓ Event-specific and global contact roles
- ✓ Separate topic/presentation ratings
- ✓ All indexes and constraints as discussed
- ✓ Comprehensive comments and documentation

The file includes:

- DROP statements (commented out for safety)
- All table definitions in correct dependency order
- All indexes for performance
- All constraints for data integrity
- Extensive comments explaining design decisions

Please review the schema file. Once you're satisfied, I can execute it against your Neon database.

Code Snippet 11:: Claude Code Generating Complete SQL Schema File

The schema file was created at database/schema.sql . I have published the complete schema design Claude Code presented to me for review as a gist here (<https://gist.github.com/nielsberglund/87983d9d642c00909c597bc0018638b5>).

Creating the Schema via MCP

Now for the moment of truth.

The Creation Process

Instead of copying this SQL and running it manually, I'll ask Claude Code to execute it via MCP:

This SQL looks perfect. Now create these tables in our Neon database via the Postgres MCP server.

Code Snippet 12:: User Requesting Table Creation via MCP

Claude Code responded:

- I'll create the tables one at a time using the PostgreSQL client via Docker, verifying each step along the way.

Step 1: Creating contacts table

[snipped for brevity]

- ✓ contacts table created successfully!

Step 2: Creating events table

[snipped for brevity]

- ✓ events table created successfully!

Step 3: Creating tickets table

[snipped for brevity]

- ✓ tickets table created successfully!

Step 4: Creating speaker_sessions table

[snipped for brevity]

- ✓ speaker_sessions table created successfully!

Step 5: Creating speaker_ratings table

[snipped for brevity]

- ✓ speaker_ratings table created successfully!

Step 6: Creating email_log table

[snipped for brevity]

- ✓ email_log table created successfully!

Step 7: Creating contact_roles table

[snipped for brevity]

- ✓ contact_roles table created successfully!

Generating Final Summary

Let me gather comprehensive information about all created database objects:

[snipped for brevity]

- ✓ Database Schema Creation Complete!

All 7 tables have been successfully created in your Neon PostgreSQL 17.7 database. Here's the complete list of objects:

📊 Tables Created (7)

[snipped for brevity]

📈 Summary Statistics

- Total Tables: 7
- Total Indexes: 35 (including 7 primary keys, 2 unique indexes)
- Total Foreign Keys: 11
- Total Check Constraints: 16

🔑 Key Design Features

- ✓ Email NOT unique - Supports shared family email addresses
- ✓ Ticket-level tracking - Walk-ins default to ticket_number = -1
- ✓ Unique constraint - (ticket_number, event_id, attendee_contact_id)
- ✓ Dual roles - Same person can be both attendee and speaker
- ✓ Event-specific & global roles - Flexible role assignments
- ✓ Separate ratings - Topic (content) and presentation (delivery)
- ✓ Comprehensive indexing - Optimized for common queries
- ✓ Data integrity - Foreign keys with appropriate DELETE actions

🌟 Database is Ready!

Your Neon PostgreSQL database is now fully configured and ready for:

- Contact management
- Event tracking
- Ticket registration and check-in
- Speaker session management
- Rating collection
- Email campaign logging
- Flexible role assignments

Next steps: Load sample data for testing! 🎉

Code Snippet 12:: Claude Code Creating Tables via MCP

Not that I don't trust Claude Code. But, just for sh*t and giggles, let's verify it ourselves.

Obviously, I could have used some PostgreSQL client to connect to the database and check, but that is so not **AI-Native** development. Instead, I asked the Claude Desktop application that we set up in Part 4 to connect to the database:

The screenshot shows the Claude Desktop interface. At the top, a red box highlights the question "What tables are in the database". Below it, a sidebar shows "3 steps" and "list_objects". A main panel displays the response: "Your PostgreSQL database has 7 tables in the `public` schema:" followed by a numbered list of tables: 1. `contact_roles`, 2. `contacts`, 3. `email_log`, 4. `events`, 5. `speaker_ratings`, 6. `speaker_sessions`, 7. `tickets`. A yellow box highlights the text "This looks like a database for managing events, possibly conferences or similar gatherings, with contacts, speakers, sessions, and ticketing." At the bottom, there's a message asking if detailed table structure should be shown, and a row of social sharing icons.

Figure 1: Claude Desktop Verifying Tables Created

As shown in **Figure 1**, Claude Desktop confirmed that all tables were created successfully. What is interesting is that Claude Desktop inferred what this database was for based on the table names and structure, even though we never told it (outlined in yellow).

Generating Migration Scripts

Best practice dictates we should have SQL migration scripts in version control, even though we created the tables conversationally. Let's have Claude Code generate those:

Before we go on to generating sample data, best practice says we should have the sql scripts in version control. Let's generate them now.

Code Snippet 13:: User Requesting Migration Scripts and Documentation

In addition to the migration scripts, I also asked Claude Code to generate comprehensive documentation, including an ER diagram in Mermaid format, table descriptions, relationships, indexes, constraints, and design decisions:

```

• Excellent idea! I'll create proper migration scripts and comprehensive documentation with an ER
[snipped for brevity]

• Perfect! I've created comprehensive migration scripts and documentation. Let me commit these to
[snipped for brevity]

•  Database Migrations and Documentation Complete!

I've created comprehensive migration scripts and documentation, all committed to version control.

---
📁 Files Created

Migration Scripts (database/migrations/)

[snipped for brevity]

Database Documentation (database/DATABASE.md)

Comprehensive 600+ line documentation including:

[snipped for brevity]

---
📊 Documentation Statistics

- Total Lines: ~1,400 lines of SQL and documentation
- Tables: 7 fully documented
- Columns: 69 with descriptions
- Indexes: 35 documented
- Foreign Keys: 11 mapped
- Check Constraints: 16 explained
- SQL Examples: 8 common query patterns
- ER Diagram: Mermaid format (GitHub-compatible)

---
🕒 Version Control Status

 Committed to Git:
[snipped for brevity]

 Pushed to GitHub: All files successfully pushed

---
⌚ Next Steps

Your database is now:
1.  Created - All tables in Neon PostgreSQL 17.7
2.  Documented - Comprehensive ER diagram and table docs
3.  Version Controlled - Migration scripts in Git
4.  Team-Ready - Documentation for collaborators

Ready for: Sample data generation and testing! 🚀

---
💻 How to Use

View ER Diagram:
Open database/DATABASE.md on GitHub - the Mermaid diagram will render automatically

Apply Migration to New Database:
psql "your-connection-string" -f database/migrations/001_create_core_tables_up.sql

Rollback if Needed:
psql "your-connection-string" -f database/migrations/001_create_core_tables_down.sql

Read Documentation:
- database/DATABASE.md - Full reference documentation
- database/migrations/README.md - Migration management guide

All documentation is written in markdown and optimized for GitHub viewing!

```

Code Snippet 14:: Claude Code Generating Migration Scripts and Documentation

In *Code Snippet 14*, you see an excerpt of Claude Code's detailed response outlining the files created, documentation statistics, version control status, and next steps!

The database documentation generated by Claude Code is available as a `gist` here (<https://gist.github.com/niebsberglund/a293684bebae08beaf16ea4f5152c987>).

Reflecting on the Experience

This is the power of AI-native development. I didn't:

- Open a SQL client
- Copy-paste SQL
- Manually run scripts
- Check if it worked
- Create documentation by hand

I just **asked** Claude Code to do it, and it:

- Executed each CREATE TABLE statement
- Verified each table was created correctly
- Checked all constraints and indexes
- Provided a comprehensive summary

This is a massive productivity boost and a glimpse into the future of software development.

Generating Test Data

Cool, I now have a database with tables created conversationally. Next, we need realistic test data to work with.

The Data Generation Conversation

With our schema in place, we need realistic test data. In traditional development, this is tedious, writing `INSERT` statements or creating seed scripts. With Claude Code, we just **describe** what we want.

```
Now generate realistic test data for our Data & AI Community Day Durban events. I want:  
  
- I want events:  
  * 2 events in 2023  
  * 3 events in 2024  
  * 5 events in 2025  
  
- 60 contacts total:  
  * Use realistic South African names (mix of cultures: Zulu, Xhosa, Afrikaans, English)  
  * Mix of genders  
  * Companies from South Africa (real and fictional tech companies)  
  * 10 people who are both speakers and attendees  
  * 30 people who only attended  
  * 20 people who only spoke  
  
- Participation records showing realistic patterns:  
  * Realistic check-in rates (90-95% of registered attendees actually attended)  
  * 5-10 people attended all 3 years (core community)  
  * 10-15 people attended 2 years  
  * Remaining people attended 1 year (first-timers)  
  
- Speaker sessions:  
  * 8-10 sessions per event  
  * Topics: AI, Machine Learning, Data Engineering, Azure, Cloud, Analytics, etc.  
  * Some speakers present at multiple events  
  * Varied tracks: Technical, Business, Ethics  
  
- Session ratings:  
  * 3-5 ratings per session (realistic attendance)  
  * Ratings mostly between 5-7 (generally positive feedback)  
  * Some variation (not all perfect 7s)  
  * Occasional written comments
```

Make the data feel real - varied but believable for a South African tech conference.

Code Snippet 15:: User Requesting Realistic Test Data Generation

Let us see what Claude Code does with this:

• I'll create realistic test data for your South African tech conference. Let me generate comprehensive seed data for you.

[snipped for brevity]

• Now let me create a README and a helper script for loading the seed data:

[snipped for brevity]

• Excellent! Now let me commit the seed data to version control:

[snipped for brevity]

• Realistic Test Data Created!

I've generated comprehensive, realistic seed data for your South African Data & AI Community Day.

Seed Files Created

database/seeds/001_sample_data.sql (550+ lines)

- 10 Events across 2023-2025
- 60 Contacts with authentic South African names
- ~90 Speaker Sessions with realistic topics

database/seeds/002_tickets_and_ratings.sql (280+ lines)

- ~200 Ticket Records with realistic attendance patterns
- ~60 Session Ratings (sample showing pattern)
- Contact Roles (organisers, sponsors, speakers)
- 10 Email Log Entries

database/seeds/README.md (Comprehensive guide)

- How to load seed data
- Data verification queries
- Customization instructions

[snipped for brevity]

🚀 Ready to Load Data!

Would you like me to load this test data into your Neon database now?

What I'll do:

1. Load 001_sample_data.sql (events, contacts, sessions)
2. Load 002_tickets_and_ratings.sql (tickets, ratings, roles)
3. Verify data loaded correctly
4. Run sample queries to show the data in action

Just say "Yes, load the seed data" and I'll execute it!

✍ What You Can Test With This Data

- Registration Queries: "Is Thabo Mthembu registered for Event 10?"
- Attendance Tracking: "How many checked in at March 2024?"
- Speaker History: "How many times has Sarah presented?"
- Ratings Analysis: "Top 5 rated speakers"
- Email Campaigns: "Send email to all 2024 attendees"
- Core Community: "Who attended all 3 years?"
- Walk-in Registration: Test walk-in flow
- Duplicate Detection: Find shared email addresses
- Attendance Trends: Compare attendance rates across events
- Speaker Rankings: Average ratings by speaker

🎯 Next Steps

Option 1: Load the seed data now (I can do it for you!) 

Option 2: Review the SQL files first, then load manually

Option 3: Customize the data before loading

Let me know how you'd like to proceed! 

Code Snippet 16:: Claude Code Generating Realistic Test Data

This is remarkable. Claude Code:

- Generated 434 total records (10 events + 60 contacts + 118 participation + 90 sessions + 44 ratings + 102 contact roles + 10 email log)
- Created realistic South African names and companies
- Built intentional patterns (retention cohorts, speaker loyalty)
- Varied the data appropriately
- Explained its strategy and results

All through **conversation**, not manual SQL writing.

The `README.md` file mentioned in *Code Snippet 16* is available as a [gist here](#) (<https://gist.github.com/nielsberglund/6385a269768e176f3e00dc5970600cc3>).

Loading the Test Data

When you look at the `database/seeds/README.md` (<https://gist.github.com/nielsberglund/6385a269768e176f3e00dc5970600cc3>), you see that Claude Code provides clear instructions on loading the seed data. However, as before, I don't want to do it manually. We have the MCP server set up, and Claude Code asked whether I wished for Claude Code to load the data. Let's do that:

I have looked at the seed data and it looks great. Please go ahead and load the seed data into our database.

Code Snippet 17:: User Requesting Test Data Loading via MCP

Claude Code responded:

• Perfect! I'll load the seed data into your Neon database, verifying each step. Let me start with:

- Step 1: Loading 001_sample_data.sql
[snipped for brevity]
- First file verified successfully!
 - Events: 10 ✓
 - Contacts: 60 ✓
 - Speaker Sessions: 90 ✓
- Step 2: Loading 002_tickets_and_ratings.sql
[snipped for brevity]
- Second file verified successfully!
 - Tickets: 118 ✓
 - Speaker Ratings: 44 ✓
 - Contact Roles: 102 ✓
 - Email Log: 10 ✓
- Step 3: Running Sample Queries to Demonstrate the Data
Let me run some interesting queries to show what you can do with this data:
[snipped for brevity]
- Perfect! Walk-ins are tracked correctly. Let me complete the summary:
- Seed Data Successfully Loaded!

All realistic test data has been loaded into your Neon PostgreSQL database and verified!
[snipped for brevity]

 What You Can Do Now

Your database is fully populated with realistic data. Here are some things to try:

Natural Language Queries (via Claude)

"Is Thabo Mthembu registered for the December 2025 event?"
 "How many people attended the March 2024 event?"
 "Who are the top 5 rated speakers?"
 "Show me all speakers from Microsoft"
 "How many first-time attendees at the July 2025 event?"
 "Which companies have the most attendees?"
 "Find all organisers"

SQL Queries You Can Run
[snipped for brevity]

 Data Highlights

- ✓ Realistic South African names - Mix of Zulu, Xhosa, Afrikaans, English, Indian
- ✓ Real companies - Vodacom, Discovery, Takealot, Standard Bank, MTN, etc.
- ✓ Attendance patterns - Core community, regulars, first-timers
- ✓ No-shows - Realistic 5-10% no-show rate
- ✓ Walk-ins - 2 on-site registrations tracked
- ✓ Rating distribution - Mostly 4-5 stars, some 3s, varied comments
- ✓ Event progression - ML basics → GenAI → Ethics (realistic topic evolution)
- ✓ Dual roles - Speakers who are also attendees
- ✓ Organisers - 3 core team members
- ✓ Sponsors - Microsoft, AWS, Google, Databricks, etc.

 Database is Production-Ready!

Your Event Contact Management System now has:

- Complete schema (7 tables, 35 indexes, 11 FKs, 16 constraints)
- Realistic test data (434 records across all tables)
- Migration scripts (version-controlled)
- Comprehensive documentation (ER diagram, queries, guides)

Ready for: MCP server integration, natural language queries, email campaigns, analytics, and reporting.

Code Snippet 18:: Claude Code Loading Test Data via MCP

From what we see in *Code Snippet 18*, everything appears to have loaded successfully.

Verifying the Test Data

Let's verify that we have data. For this, I used VSCode:

```

3 SELECT COUNT(*) as contacts FROM contacts;
4 SELECT COUNT(*) as events FROM events;
5 SELECT COUNT(*) as tickets FROM tickets;

```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS POLYGLOT NOTEBOOK POSTGRES

	Results	Messages
1	contacts ↑▼	
1	60	
1	events ↑▼	
1	10	
1	tickets ↑▼	
1	118	

Figure 2: Verifying Loaded Test Data in VSCode

As you see in *Figure 2*, I did a `COUNT(*)` on a couple of tables. All looks good.

NOTE: Why didn't I ask Claude Code/Desktop to verify the data? Good question. I wanted to demonstrate that you can use any PostgreSQL client to connect to the Neon database and verify the data. You are not locked into using Claude Code/Desktop for everything. You can mix and match tools as needed.

Our database should now have realistic, queryable data that demonstrates real-world patterns.

The Dual Interface Advantage: Development + Querying

Now that Claude Code has created and populated our database, let's see the full AI-native workflow in action. We have:

1. **Claude Code** - which just created our entire database schema and test data
2. **Claude Desktop** - which we configured in Part 4 for querying

Both connect to the same Neon database via Postgres MCP Pro, but they serve different purposes:

Claude Code: The Development Interface

Used for:

- Designing and creating database schemas
- Generating and inserting test data
- Creating migration scripts
- Database refactoring and changes
- Generating documentation

Example conversation:

```

You: "Create the database tables we just designed"
Claude Code: [uses postgres:execute_sql via MCP to create all tables, verifies each one]

You: "Generate realistic test data with South African names"
Claude Code: [generates 40 contacts, 145 participation records, 42 sessions, 156 ratings via MCP]

```

Claude Desktop: The Query Interface

Used for:

- Importing Quicket and Sessionize data
- Querying data in natural language
- Running analytics and reports
- Exploring data without writing SQL
- Ad-hoc questions and insights
- Data verification

Example conversation:

```

You: "How many people attended each year?"
Claude Desktop: [uses postgres:execute_sql via MCP to query, presents formatted results]

You: "Who are our most engaged community members?"
Claude Desktop: [crafts complex JOIN query, calculates engagement scores, explains methodology]

```

Let's see both in action with real queries.

Natural Language Queries with Claude Desktop

Now that Claude Code has created and populated our database, let's switch to **Claude Desktop** to query it. This demonstrates the complete AI-native workflow: develop with Claude Code, query with Claude Desktop.

Simple Queries

I opened Claude Desktop (the GUI application we configured in Part 4) and started asking questions about the data Claude Code just created. The queries and Claude Desktop's responses are shown in the following Figures. You see the natural language query outlined in yellow, Claude's thinking in blue, and the final response in red:



Figure 3: Basic Count

What just happened:

1. I asked a question in plain English (no SQL)
2. Claude Desktop understood I wanted a count from the contacts table
3. It used the `postgres:execute_sql` tool via MCP to query the database
4. It initially got an error but recovered.
5. It presented the result in a clear, conversational format

This is the paradigm shift: Domain experts, event organizers, business analysts—anyone can query the database without knowing SQL.

How many of those contacts are speakers

1 step

P execute_sql

- Thought process

I got an error - the column name isn't "role". Let me check the structure of the contact_roles table to see what columns it actually has.

Let me check the structure of the contact_roles table first:

P get_object_details

- Thought process

Good! So the column is called "role_type", not "role". Now I can query for contacts where role_type = 'Speaker' (or maybe it's 'speaker' - I should check what values exist).

Now I can query correctly using the `role_type` column:

P execute_sql

- Thought process

Hmm, got 0 results. Maybe the value is lowercase 'speaker' or maybe there's a different way speakers are tracked. Let me check what role_type values actually exist in the table.

Let me check what role types actually exist:

2 steps

P execute_sql

Request

```
{
  `sql`: 'SELECT COUNT(DISTINCT contact_id) as speaker_count
FROM contact_roles
WHERE role_type = \'speaker\''
}'
```

Response

```
[{'speaker_count': 30}]
```

- Perfect! So there are 30 contacts who have the speaker role.

Out of the 60 total contacts, **30 contacts are speakers.**

⬇️

Figure 4: Filtered Count

What we see in *Figure 4* is interesting because it shows Claude's reasoning process. It first received an error, corrected it, which didn't return any rows; checked the column names and filter values; corrected the query; and finally got the correct answer. Pretty amazing!

Who attended at least one event in 2024. Please show name and event

Let me check the tickets table structure:

- P get_object_details
 - Identified correct database column for query refinement.

Now I can query correctly using `attendee_contact_id`:

- P execute_sql
 - Verified database schema for accurate query construction.

Let me check the events table structure:

- 2 steps
 - P execute_sql
 - Synthesized attendance data across three 2024 events.

Here are the attendees for the 2024 events:

Data & AI Community Day Durban - March 2024 (March 16, 2024)

1. Chloe Williams
2. Johan Botha
3. Kevin O'Brien

Data & AI Community Day Durban - June 2024 (June 22, 2024)

1. David Chen
2. Hendrik Viljoen
3. Johan Botha
4. V

Data & AI Community Day Durban - November 2024 (November 9, 2024)

1. Andile Shabalala
2. Emma Johnson
3. Johan Botha

Notable observations:

- Some attendees were very engaged and attended multiple events, including Johan Botha, Kevin O'Brien, Nomsa Khumalo, Priya Naidoo, Sarah van der Merwe, Sipho Ndlovu, and Thabo Mthembu who attended all three 2024 events!

Figure 5: Recent Event Attendance

The last basic query in *Figure 5* shows how Claude Desktop retrieves the correct data and provides data insights, such as who is engaged and who attended more than one event (highlighted in green).

Complex Analytical Queries

Let us see if we can make things more interesting with some complex queries that require JOINs and aggregations:

```
Who are our most engaged community members? Consider both attendance across years and speaking engagements.
```

Code Snippet 19:: User Requesting Complex Engagement Analysis

What we ask for in *Code Snippet 19* is a complex query requiring Claude Code to:

1. Understand “engagement” holistically
2. Create a scoring methodology
3. JOIN multiple tables
4. Aggregate across years
5. Present meaningful insights

The initial thought process of Claude Desktop:

The user wants to identify the most engaged community members by considering:

- * Attendance across years (not just 2024)
- * Speaking engagements

I need to create a scoring system that rewards both. Let me think about this:

- * Attendance: Could give points per event attended
- * Speaking: Could give points per speaking engagement (probably should be worth more since speaking is higher engagement)

Let me first understand what data I have:

- * Tickets table - shows who attended which events
- * Contact_roles table - shows who had what roles (speaker, organiser, sponsor)
- * Events table - has the events

A good scoring system might be:

- * 1 point per event attended
- * 3-5 points per speaking engagement (since speaking is higher engagement)
- * Maybe bonus points for multi-year attendance

Let me query:

- * Get all attendees with count of events attended
- * Get all speakers with count of speaking engagements
- * Combine them with a scoring formula

I'll create a query that does this.

Code Snippet 20:: Claude Code's Thought Process for Engagement Scoring

Claude Desktop then created and executed the following SQL query:

```

1  WITH attendance_score AS (
2    SELECT
3      c.id, c.first_name || ' ' || c.last_name as name,
4      COUNT(DISTINCT t.event_id) as events_attended,
5      COUNT(DISTINCT e.year) as years_active,
6      COUNT(DISTINCT t.event_id) as attendance_points
7    FROM contacts c
8    JOIN tickets t ON c.id = t.attendee_contact_id
9    JOIN events e ON t.event_id = e.id
10   GROUP BY c.id, c.first_name, c.last_name
11 ),
12 speaking_score AS (
13   SELECT
14     c.id, c.first_name || ' ' || c.last_name as name,
15     COUNT(DISTINCT cr.event_id) as speaking_engagements,
16     COUNT(DISTINCT cr.event_id) * 5 as speaking_points
17   FROM contacts c
18   JOIN contact_roles cr ON c.id = cr.contact_id
19   WHERE cr.role_type = 'speaker'
20   GROUP BY c.id, c.first_name, c.last_name
21 ),
22 combined_score AS (
23   SELECT
24     COALESCE(a.id, s.id) as contact_id,
25     COALESCE(a.name, s.name) as name,
26     COALESCE(a.events_attended, 0) as events_attended,
27     COALESCE(a.years_active, 0) as years_active,
28     COALESCE(s.speaking_engagements, 0) as speaking_engagements,
29     COALESCE(a.attendance_points, 0) as attendance_points,
30     COALESCE(s.speaking_points, 0) as speaking_points,
31     CASE WHEN COALESCE(a.years_active, 0) > 1
32       THEN (COALESCE(a.years_active, 0) - 1) * 2
33     ELSE 0
34     END as multi_year_bonus,
35     COALESCE(a.attendance_points, 0) +
36     COALESCE(s.speaking_points, 0) +
37     CASE WHEN COALESCE(a.years_active, 0) > 1
38       THEN (COALESCE(a.years_active, 0) - 1) * 2
39     ELSE 0
40     END as total_score
41   FROM attendance_score a
42   FULL OUTER JOIN speaking_score s ON a.id = s.id
43 )
44 SELECT
45   name,
46   events_attended,
47   years_active,
48   speaking_engagements,
49   attendance_points,
50   speaking_points,
51   multi_year_bonus,
52   total_score
53   FROM combined_score
54   ORDER BY total_score DESC, events_attended DESC
55   LIMIT 20

```

Code Snippet 21:: Claude Code's Complex Engagement SQL Query

The query in *Code Snippet 21* eventually returned the following results:

Most Engaged Community Members

Scoring System:

- **Attendance Points:** 1 point per event attended
- **Speaking Points:** 5 points per speaking engagement (higher value reflects greater commitment)
- **Multi-Year Bonus:** 2 points for each additional year of engagement beyond the first year
- **Total Score:** Sum of all points

🥇 Top Tier - Super Engaged Members (Score: 34)

These seven community members are your **absolute champions** - they've attended ALL 10 events across 3 years AND spoken at 4 different events:

1. **Thabo Mthembu** - 10 events, 4 talks, 3 years active
2. **Kevin O'Brien** - 10 events, 4 talks, 3 years active
3. **Name removed** - 10 events, 4 talks, 3 years active
4. **John Doe** - 10 events, 4 talks, 3 years active
5. **Jane Smith** - 10 events, 4 talks, 3 years active
6. **Mike Johnson** - 10 events, 4 talks, 3 years active
7. **Sipho Ndlovu** - 10 events, 4 talks, 3 years active

🥈 Second Tier - Highly Engaged (Score: 30)

These members have spoken at 4 events and attended 6 events across 3 years:

8. **Pieter Kruger** - 6 events, 4 talks, 3 years active

10. **Thandiwe Mokoena** - 6 events, 4 talks, 3 years active

🥉 Third Tier - Dedicated Speakers (Score: 20)

Expert speakers who contributed significantly but didn't attend as general attendees:

11. **Prof. Kobus van Heerden** - 4 speaking engagements
12. **Dr Thandi Mabaso** - 4 speaking engagements

Notable Insights:

- Your **top 7 members** are exceptionally committed - 100% attendance AND regular speakers
- These champions represent your community's core and would be excellent candidates for:
 - Advisory roles
 - Mentorship programs
 - Ambassador opportunities
 - Early access to new initiatives
- The multi-year engagement bonus identifies loyalty - all top performers have been with you since the beginning

Would you like me to create a more detailed profile of any of these members or analyze specific engagement patterns?

Figure 6: Complex Engagement Analysis Results

I have made some cuts in the results in *Figure 6* to fit the image, but you get the idea.

This is transformative. I asked a vague question ("most engaged"), and Claude Desktop:

1. Invented a scoring methodology (explained in thinking)
2. Wrote a complex CTE query with multiple JOINs
3. Extracted business insights from the data

No SQL knowledge required. An event organizer could ask this exact question and get these insights, something impossible with traditional database tools.

NOTE: The scoring system used here is arbitrary and can be adjusted based on your definition of engagement. The key takeaway is that Claude Desktop/Code can understand complex requirements and translate them into SQL.

Cohort Retention Analysis

We'll wrap up this part with some analytics that offer deeper insights. Let us look at retention:

```
Analyze attendee retention. For people who attended in 2023, how many came back in subsequent years?
```

Code Snippet 22:: User Requesting Retention Cohort Analysis

The thought process of Claude Desktop:

```
I'll create a cohort retention analysis starting with your 2023 attendees.## 2023 Cohort Retention

### 📈 Executive Summary

**Cohort Size:** 18 attendees who first joined in 2023

**Retention Rates:**
- **2024 Retention:** 55.6% (10 out of 18)
- **2025 Retention:** 55.6% (10 out of 18)
- **Complete Retention (all 3 years):** 55.6% (10 out of 18)

### 🌟 Key Findings

[snipped for brevity]

**The retention is binary** - people either stayed fully engaged or dropped off completely. There
- ✓ **No partial retention** (no one who attended 2024 but skipped 2025)
- ✓ **No "win-back"** cases (no one who skipped 2024 then returned in 2025)

### 📈 Engagement Intensity Over Time

The 10 retained members didn't just stay - they **dramatically increased** their engagement:
[snipped for brevity]

**Key Insight:** While you lost half your cohort, those who stayed became **significantly more engaged** over time.

[snipped for brevity]

### 🎯 The Loyal 10 (Retained All Years)

[snipped for brevity]

###💡 Strategic Recommendations

**Strengths:**
- 55.6% retention is actually quite strong for community events
- Those who stay become super-engaged (4.2 events per person in 2025!)
- Zero churn after year 1 - if they make it past 2023, they stay

**Opportunities:**
- **Focus on Year 1 experience:** The 44.4% who didn't return all left after 2023. Improving the onboarding process could help retain them.
- **Early engagement matters:** Consider more touchpoints in the first 6 months.
- **Exit interviews:** Contact the 8 who left to understand why.
- **New member onboarding:** Create a buddy system pairing new attendees with your loyal 10.

**Questions to explore:**
- What made 2023 different for those who left vs. those who stayed?
- Did the 8 who left attend only 1 event in 2023, or both?
- Were there any common characteristics among those who churned?

Would you like me to dig deeper into any of these areas?
```

Code Snippet 23:: Claude Desktop's Retention Analysis Thought Process

Again, remarkable. I asked for “cohort analysis” and Claude Desktop:

- ✓ Understood “cohort analysis” conceptually
- ✓ Built appropriate SQL with CTEs
- ✓ Calculated retention percentages
- ✓ Contextualized results with insights
- ✓ Provided strategic recommendations
- ✓ Proactively suggested follow-up analyses

This is the power of the dual interface:

- **Claude Code** created the schema and data (development)
- **Claude Desktop** analyzes that data conversationally (usage)

Together, they form a complete AI-native data workflow from development to insights.

What We've Accomplished

Let's reflect on what we just did:

Infrastructure Built

- ✓ Claude Code configured for Postgres MCP Pro
- ✓ Database schema designed through conversation
- ✓ 7 tables created directly in Neon via MCP
- ✓ Migration scripts generated for version control
- ✓ Comprehensive documentation created automatically

Data Generated

- ✓ 10 realistic events (2023-2025)
- ✓ 60 diverse contacts with South African profiles
- ✓ 118 participation records showing real patterns
- ✓ 90 speaker sessions across topics
- ✓ 44 session ratings with realistic distributions

Capabilities Demonstrated

- ✓ Simple queries (counts, filters)
- ✓ Complex multi-table JOINs
- ✓ Analytical queries (cohort analysis, engagement scoring)

- Natural language conversation throughout

Time Investment

- **Traditional approach:** 10-11 hours
 - Schema design: 3 hours
 - SQL writing: 2 hours
 - Test data generation: 4-5 hours
 - Testing and verification: 1 hour
- **AI-Native approach:** ~30 minutes
 - Configuration: 5 minutes
 - Conversational design: 10 minutes
 - Data generation: 10 minutes
 - Verification queries: 5 minutes

A lot, lot faster, and better documented.

The Paradigm Shift Realised

Remember in Part 3 when I said we were building an **AI-native system**? Today, you saw what that actually means:

In **traditional development**:

```

1  -- I write this
2  CREATE TABLE contacts (
3    id SERIAL PRIMARY KEY,
4    ...
5  );
6
7  -- I run this manually
8  psql -h neon.tech -d event_contact_db -f schema.sql
9
10 -- I write more SQL
11 INSERT INTO contacts VALUES (...);
12
13 -- I write queries
14 SELECT COUNT(*) FROM contacts WHERE ...;
```

Code Snippet 24:: Typical Traditional Development Workflow

Compare this with **AI-Native Development**:

```

Me: "Design a database schema for our event management system"
Claude Code: [designs, creates, documents]

Me: "Add realistic test data"
Claude Code: [generates, inserts, verifies]

Me: "Who are our most engaged members?"
Claude Desktop/Code: [analyzes, queries, presents insights]
```

Code Snippet 25:: AI-Native Development Workflow

The difference is **transformative**.

What's Next: Part 6 Preview

With our database populated with test data, Part 6 will focus on building a **custom MCP server** to import real data from Quicket and Sessionize.

What We'll Build

1. **Custom Import MCP Server** (using FastMCP framework)
 - import_quicket_registrations() - Parse Quicket CSV exports
 - import_quicket_checkins() - Update attendance from check-in data
 - import_sessionize_speakers() - Import speaker and session data
 - import Speaker Ratings() - Import session ratings and feedback
 - preview_import() - Dry-run validation before inserting
 - validate_import_data() - Format-specific validation
2. **Real-World Data Processing**
 - Handle Quicket's CSV format quirks
 - Parse Sessionize's JSON exports
 - Detect duplicates (email-based matching)
 - Resolve conflicts (update vs skip vs merge)
3. **Integration with Existing Infrastructure**
 - Custom MCP server calls Postgres MCP Pro for inserts
 - Uses csv-mcp-server for validation
 - Provides conversational import workflow

Example of What's Coming

Having built the custom MCP server, the import workflow will look like this:

```

Me: "Import this Quicket registration file"
[attaches CSV]

Claude Code:
[uses import_quicket_registrations via custom MCP]
"Found 142 registrations. 3 people are already in the database.
How should I handle duplicates?"

Me: "Update existing records with new registration info"

Claude Code:
[processes data]
"Import complete:
- 139 new contacts created
- 3 existing contacts updated
- 142 participation records created
- All linked to 'Data & AI Day 2025' event"

Me: "Show me the new registrations from the past week"

Claude Code:
[queries database]
"12 new registrations in the past week:
[shows list with names, companies, registration dates]

Notable: 5 are first-time attendees, 7 are returning from previous years"

```

Code Snippet 26:: Preview of Custom MCP Import Workflow

This continues the AI-native theme, **import through conversation**, not manual scripts.

Summary: From SQL to Conversation

Today, we crossed a major threshold. We didn't just use AI to **write code faster**, we fundamentally changed **how we interact with databases**, in both development and usage.

What We Built:

- Production database schema created conversationally by Claude Code
- Realistic test data generated and inserted through dialogue
- Natural language querying working end-to-end in Claude Desktop
- Complete documentation and migration scripts generated automatically
- Foundation for AI-native data management with dual interfaces

What We Learned:

- **AI-native development is about conversation, not coding**
 - Claude Code designs, creates, and populates databases via MCP
 - No copy-pasting SQL or manual command execution
 - Iteration happens through dialogue, not file editing
- **MCP servers enable true conversational data work**
 - Same infrastructure serves both development (Claude Code) and usage (Claude Desktop)
 - Natural language becomes the interface for both creating and querying
 - Domain experts can participate without technical knowledge
- **Complex analytics can be expressed as simple questions**
 - "Who are our most engaged members?" → Complex JOIN with scoring algorithm
 - "Analyze retention" → Cohort analysis with percentages
 - No SQL knowledge required from the person asking
- **Documentation happens automatically**
 - Conversation history documents all design decisions
 - Migration scripts generated alongside table creation
 - Schema documentation created as part of the process

Why This Matters:

This isn't just a faster way to build databases. It's a **new way of thinking** about data systems.

When the interface is conversational:

- **Development:** Domain experts can participate in schema design (Claude Code)
- **Usage:** Business users can query directly without SQL (Claude Desktop)
- **Analysis:** Insights emerge through dialogue, not predetermined reports
- **Iteration:** Changes happen through conversation, not code rewrites
- **Documentation:** Built-in through conversation history

The paradigm shift is complete: we've moved from **SQL scripts** to **conversational data work** for both developers (Claude Code) and users (Claude Desktop).

In Part 6, we'll extend this to real-world data import, showing how the entire data pipeline can be conversational.

~ Finally

That's all for Part 5! We've transformed our empty Neon database into a fully populated, queryable system, all through conversation with Claude Code (for development) and Claude Desktop (for querying).

Key Takeaways:

1. **Dual interface power:** Claude Code creates, Claude Desktop queries—both conversational
2. **Schema design through dialogue** is faster, better documented, and more accessible
3. **Test data generation via AI** creates realistic, varied datasets automatically
4. **Natural language querying** makes databases accessible to everyone
5. **MCP architecture** enables truly conversational data work from development to insights

Next time: We build a custom MCP server for importing real event data, completing our AI-native event management system.

Your Turn

I encourage you to try this yourself:

1. Configure Claude Code with Postgres MCP Pro (for development)
2. Design a simple schema through conversation
3. Let Claude Code create and populate it via MCP
4. Use Claude Desktop to query with natural language
5. Experience the dual-interface AI-native workflow

Experience the shift from SQL to conversation—in both development and usage. It's genuinely transformative.

Have questions or thoughts?

Ping me: [\(mailto:niels.it.berglund@gmail.com\)](mailto:niels.it.berglund@gmail.com)

Follow on LinkedIn: [linkedin.com/in/nielsberglund](https://www.linkedin.com/in/nielsberglund) (<https://www.linkedin.com/in/nielsberglund/>)

Found this helpful? Share it with your network! The AI-native development revolution is happening now, and you've just seen how it works in practice.

Series Navigation:

- Part 1: Installation & Setup (</post/2025-07-29-building-an-event-management-system-with-claude-code-claude-code-installation-and-initialisation/>)
- Part 2: IDE Integration (</post/2025-08-13-building-an-event-management-system-with-claude-code-part-2---ide-integration-and-advanced-features/>)
- Part 3: Architecture Planning (</post/2025-12-28-building-an-event-management-system-with-claude-code-part-3---architecting-an-ai-native-system/>)
- Part 4: Database Infrastructure (</post/2026-01-01-building-an-event-management-system-with-claude-code-part-4---database-setup-and-first-conversations/>)
- **Part 5: Schema & Natural Language Queries** (you are here)
- Part 6: Custom MCP Server for Data Import (coming soon)
- Part 7: Email Integration with Brevo MCP
- Part 8: Production Usage and Real Events

← **PREVIOUS POST** (/DRAFTS/BUILDING-AN-EVENT-MANAGEMENT-SYSTEM-WITH-CLAUDE-CODE-PART-4---DATABASE-SETUP-AND-FIRST-CONVERSATIONS/)

Disqus comments not available by default when the website is previewed locally.
comments powered by Disqus (<https://disqus.com>)



Copyright © Niels Berglund 2026

CleanWhite Hugo Theme (<https://themes.gohugo.io/hugo-theme-cleanwhite>) by Huabing
 (<https://zhaohuabing.com>) | 802