
# Module 2: Architectural Planning and System Design, Lesson 3: AI-Assisted Database Schema Design

# Hands-On: Schema-First Design — From Domain to DDL in Three Prompts

## Overview

This demo proves three claims from the lecture in real time:

- Entity identification before DDL prevents expensive missing-table migrations later
- A structured Role + Instruction + Context + Example prompt produces a complete schema artifact in one pass
- Schema evolution via natural language is faster and safer than writing ALTER TABLE by hand

You will design the core TaskFlow database — entities, DDL with indexes and constraints, and a reversible soft-delete migration — using exactly three prompts, all in the same AI chat window. The student lab extends this to a rollback rehearsal and an EXPLAIN ANALYZE review of the three core query patterns.

---

## Before You Start

- Open your AI tool of choice: Claude.ai, Cursor, or GitHub Copilot Chat
- Have a PostgreSQL 14+ instance ready (local, Docker, or a free hosted tier) and an empty database named taskflow
- Keep the four-part prompt template visible on screen or a second monitor throughout
- Do NOT start a new chat between prompts — session continuity is the point

---

> **Remember**
> Every schema prompt follows the same four parts: Role, Instruction, Context, Example. We will prove it by designing TaskFlow from scratch — entities first, DDL second, refinement third — all in one session. Watch the pattern repeat.

## During Hands-On — Keep Visible if Possible

1. ROLE — expertise level + platform specifics (Postgres, multi-tenant SaaS)
2. INSTRUCTION — what deliverables (entities, DDL, indexes, constraints, migration)
3. CONTEXT — domain description, existing tables, query patterns, ownership rules

4. EXAMPLE — output format (entity list → CREATE TABLE in dependency order → CREATE INDEX)

---

## Prompt 1 — Entity Identification (Before Any SQL)

> **What to Learn**
> If you jump straight to DDL, AI will often miss entities — usually a junction table for a many-to-many, or a lookup table for a reference type. Doing entity identification as a separate step surfaces the full domain model before you commit to SQL. One extra prompt step prevents a missing-table migration that is genuinely painful to do later.

**Paste this into the AI chat:**

```
List every entity in this system description, identify its key
attributes, and propose relationships between entities with cardinality
(1:1, 1:M, or M:M).

SYSTEM: TaskFlow is a multi-tenant SaaS project management platform.
Teams belong to tenants. Users belong to teams. Projects belong to
teams. Projects contain tasks. Tasks can depend on other tasks — a
task may have multiple prerequisites and be a prerequisite for
multiple other tasks. Each task receives an AI-generated priority
score, computed and updated by a separate ML service on its own
cadence. We also need an append-only audit log of every change
made to tasks.

Return the output as:
  1. A numbered entity list with key attributes
  2. A relationship table with cardinality and one-line reasoning
  3. A note on any bridging/junction tables required
```

> **What you should get**
> Seven entities: tenants, teams, users, projects, tasks, task_dependencies (junction), ai_priority_scores (1:1 with tasks), and audit_log. Notice how AI surfaces task_dependencies because you described the M:M explicitly — and the 1:1 on scores because you described separate service ownership. These are exactly the two entities that get missed most often when you skip this step.

**You should ask:**

“Why should ai_priority_scores be its own table rather than columns on tasks?”

Answer: Separate access patterns and separate ownership. The ML service writes scores on its own schedule; the task service never writes there. Keeping scores in their own table means the tasks table stays lean, the hot write path (task edits) doesn’t contend with the cold write path (scoring), and either service can be scaled independently. The 1:1 relationship is the structural expression of a service boundary. This is the kind of decision AI surfaces when you describe services, not just data.

---

## Prompt 2 — Full Schema with the Four-Part Template

> **Remember**
> Same chat window. The entities are already in context — I don’t repeat them. I give the AI the role, the instruction, the remaining context, and the output format. Watch the indexes and constraints appear exactly as specified because I wrote a BEHAVIOR block, not a wish list.

**Continue in the same chat session — do NOT start a new one:**

```
ROLE: You are a senior database architect specializing in PostgreSQL
for multi-tenant SaaS platforms. You think in terms of row-level
security, tenant isolation, connection pooling, and write amplification.

INSTRUCTION: Using the entity list above, produce a complete schema
artifact with:
  - All tables: column names, PostgreSQL data types, nullability
  - Primary keys (UUID with gen_random_uuid()) and foreign keys with
    explicit ON DELETE behavior
  - Unique and CHECK constraints
  - Recommended indexes, each with a one-line -- justification comment
  - The complete DDL script, ready to run

CONTEXT:
  - Every tenant-owned table carries tenant_id and has an index
    leading with tenant_id
  - tasks.status is one of: draft, active, blocked, done
  - Top query patterns (index for these specifically):
      1. List active tasks for a project, sorted by priority_score DESC
      2. Fetch a task with its dependencies (both directions)
```

```
      3. Fetch the audit trail for a single task, newest first
  - Ownership: project -> team -> tenant. Deleting a team cascades
    tasks but NOT users.
  - Audit log stores old_values and new_values as JSONB.

EXAMPLE output format (follow this exactly):
  1. Entity list (one line each)
  2. CREATE TABLE statements in dependency order (parents before children)
  3. CREATE INDEX statements grouped by table, each with -- justification
  4. A short "Design Notes" section explaining non-obvious choices
```

**What you should get**

A complete DDL script in dependency order. The tasks table gets a composite index on (tenant_id, project_id, status, priority_score DESC). A partial index appears on tasks WHERE status IN ('active','blocked') — AI inferred it from the "list active tasks" query pattern. The audit log uses JSONB for old_values/new_values. Every foreign key has explicit ON DELETE behavior. task_dependencies has a composite primary key on (task_id, depends_on_task_id) that prevents duplicate edges at the database level.

Save the output as taskflow_v1.sql and apply it to an empty database:

```
psql -d taskflow -f taskflow_v1.sql
```

*Tip: The partial index appears because you described the query pattern in CONTEXT. Generic "index every foreign key" advice would have missed it. This is contract-first thinking applied to storage — the indexes reflect the workload because the workload was in the prompt.*

---

## Prompt 3 — Iterative Refinement + Self-Review

**What to Learn**

Stop writing ALTER TABLE. Describe the business need; review the SQL. Then run the self-review prompt to catch gaps before a human reviewer ever sees the schema. This is where AI schema design stops feeling like code generation and starts feeling like pair programming with a senior database engineer.

**Third prompt — still the same session:**

```
Two changes, plus a self-review.
```

```
CHANGE 1: Add soft-delete support to all user-facing tables
(tenants, teams, users, projects, tasks). Add a deleted_at TIMESTAMPTZ
column (nullable). Add partial indexes WHERE deleted_at IS NULL for
every query that should exclude deleted rows. Note which existing
queries need WHERE clauses updated.

CHANGE 2: Audit-log every change to tasks. Generate the AFTER
INSERT/UPDATE/DELETE trigger function that populates audit_log with
old_values and new_values as JSONB. Include the CREATE TRIGGER statement.

Output both changes as a single reversible migration with an up()
section and a down() section, with one-line comments on each step.
Use plain SQL (no framework-specific syntax).

Then — in the same response, after the migration — run this
self-review on the combined schema (v1 + this migration):
  "Review this schema for normalization violations, missing tenant
   isolation, anti-patterns (EAV, comma-separated lists, TEXT IDs,
   missing updated_at), and index gaps for the stated query patterns.
   List each finding with a one-line fix."
```

> **What you should get**
>
> A single migration script with symmetric up/down sections. The down() drops the trigger first, then the function, then the partial indexes, then the deleted_at columns — reverse dependency order. The self-review flags anything it missed on Prompt 2: commonly a missing updated_at on one table, or a (tenant_id, deleted_at) composite index that would make the soft-delete filter sargable.

**Apply the migration:**

```
psql -d taskflow -f taskflow_v2_migration.sql
```

At least one self-review finding is likely worth addressing. This is intentional:

- The AI produced the schema in 30 seconds. You verify and own the judgment calls.
- If a finding surprises you, read the justification. The fix is usually one column or one index.
- A flagged gap the AI caught is a production incident you won't have.

---

## Wrap-Up — The Pattern, The Assignment Handoff

> **Remember**
>
> Three prompts. One session. Entities, a full DDL artifact with tailored indexes and constraints, and a reversible migration with self-review. No whiteboard. No manual CREATE TABLE. No ALTER TABLE. The schema documents itself because the justifications came from the same prompt as the SQL. That is AI-assisted schema design.

**Key takeaways:**

- Entity identification before DDL — always. The extra prompt is cheap; the missing-table migration is not.
- Four-part prompt: role, instruction, context, example. Every part earns its place.
- Query patterns in CONTEXT produce targeted indexes — not generic foreign-key coverage.
- Iterate with natural language. Describe the business need, review the SQL. Your time is worth more than DDL.
- Self-review before human review. It catches the obvious gaps in seconds so the human review can focus on judgment.