Module 2: Architectural Planning and System Design

Capstone: TaskFlow Mini — From Prompt to Running App

Overview

This capstone proves the four claims from Module 2 in real time:

- A focused R-I-C-E prompt produces a usable PRD in minutes — not weeks — with user stories, Gherkin acceptance criteria, and an explicit out-of-scope list.
- AI generates a normalized schema with indexes, constraints, and seed data from a plain-English domain description — not a whiteboard session.
- Mermaid.js plus AI produces architecture, sequence, and ER diagrams in seconds, in a syntax that lives in your repo and renders inside your app.
- All three artifacts plug directly into a running full-stack application — the diagrams are not just docs in a wiki, they are a feature on the page.

You will go from blank directory to a running TaskFlow Mini app — Express backend, SQLite database with the AI-designed schema, and a single-page frontend that renders the AI-generated Mermaid diagrams live in the browser — in one session.

Before You Start

- Have Node.js 20+ installed. Verify with node --version.
- Have an Anthropic API key in your environment, or have Claude.ai open in a browser tab to run prompts side-by-side with your IDE: export ANTHROPIC_API_KEY=sk-ant-...
- Create an empty working directory. We will populate it from scratch: mkdir taskflow-mini && cd taskflow-mini.
- Have a modern browser open (Chrome/Edge/Safari). The app runs at http://localhost:3000 in Segment 4.
- Optional but recommended: open mermaid.live in a tab so you can sanity-check generated diagrams before pasting into the app.

During Hands-On — Four Segments at a Glance

- SEGMENT 1 — Generate the PRD with AI (R-I-C-E applied) — ~6 min
- SEGMENT 2 — Design the schema (entity ID → DDL → indexes) — ~6 min
- SEGMENT 3 — Generate three Mermaid diagrams (Architecture, Sequence, ER) — ~6 min
- SEGMENT 4 — Build and run the full-stack app — diagrams as a feature — ~7 min

> **Remember**
> Module 2 is not about hand-coding a CRUD app. It is about practicing the AI-human handshake at full speed: AI drafts an artifact, you validate it, that validated artifact becomes the context for the next prompt, and you ship. By the end of today, the PRD, the schema, the diagrams, and the running app will all trace back to one structured prompt session — and every downstream artifact will be cleaner because the upstream one was.

Segment 1 — Generate the TaskFlow PRD

> **What to Learn**
> A vague prompt produces a vague PRD. A focused R-I-C-E prompt — Role, Instructions, Context, Examples — produces a complete one in a single shot. We will generate a PRD for TaskFlow Mini (a small project-management feature) with a quantified success metric, two user personas, eight user stories, Gherkin acceptance criteria for each, non-functional requirements, and an explicit out-of-scope section. This PRD is the contract every later segment will negotiate against.

Step 1. Create the project directory and a docs folder.

```
mkdir taskflow-mini && cd taskflow-mini
mkdir docs
```

Step 2. Open Claude.ai in a browser tab. Paste this R-I-C-E PRD prompt verbatim:

```
ROLE: You are a senior product manager and software architect with
10 years of experience shipping B2B SaaS products.

INSTRUCTIONS: Generate a PRD for the feature described below.
Include exactly these six sections, in this order:
  1. Executive Summary (3-5 sentences, ONE quantified success metric)
  2. User Personas (2 personas, each with role, goals, frustrations)
  3. User Stories (8 stories, "As a / I want / So that" format,
     prioritized P0 / P1 / P2)
  4. Acceptance Criteria (Gherkin GIVEN/WHEN/THEN, 3 per story)
```

```
  5. Non-Functional Requirements (performance, security, scale)
  6. Out of Scope (3-5 explicit items we are NOT building)

CONTEXT: We are building TaskFlow Mini — a single-team task tracker.
Stack: Node.js + Express backend, SQLite, single-page HTML/JS frontend.
Users are 5-15 person engineering teams. No auth in MVP.
Target: <100ms task list latency. Compliance: none for MVP.

EXAMPLE story format:
  "As a team lead, I want to assign a task to a teammate
   so that ownership is clear."
EXAMPLE acceptance criterion:
  "GIVEN I am viewing the task list, WHEN I click '+ New Task',
   THEN the task is created and visible within 100ms."

Output as Markdown.
```

Step 3. Save the output as docs/PRD.md. Open it and read the Out of Scope section out loud to the class — that section alone will save the most time over the next three segments.

Step 4. Quick validation. Verify the PRD has all six sections, exactly eight user stories, three acceptance criteria per story (24 total), and at least three out-of-scope items. If any section is thin, follow up with: "Expand section 4 with three more acceptance criteria per story." AI will fill the gap immediately.

> **What you should get**
> A 2-3 page PRD.md file with all six required sections, two distinct personas, 8 user stories with P0/P1/P2 priorities, 24 Gherkin acceptance criteria, a non-functional section that includes the <100ms target, and a concrete out-of-scope list. Total time: about 4 minutes of prompting plus 2 minutes of skimming. Compare that to the 2-4 weeks Lecture 2 said this normally takes. The PRD is now the single source of truth that drives every prompt in the next three segments.

Segment 2 — Design the Schema

> **What to Learn**
> Lecture 3 was emphatic: never go straight from a description to CREATE TABLE. Do entity identification first. We will use the two-step workflow — entities → DDL — to turn the PRD into a complete SQLite schema with constraints, indexes, justifications, and seed data. The output drops directly into our app in Segment 4.

Step 1. In Claude.ai, run the entity-identification prompt. Paste your PRD.md contents at the placeholder. Do not let the model generate any SQL yet — that is the whole point.

```
You are a senior database architect. Read the PRD below and list
every entity, its key attributes, and the cardinality of every
relationship between entities. Use a structured list. Do NOT write
any SQL yet.

[paste the contents of docs/PRD.md here]
```

Step 2. Review the entity list with the class. This is the cheapest moment to catch a missing junction table or a forgotten lookup. If the model missed something — for example, a task_assignees junction for a many-to-many — call it out and ask: "Add task_assignees as a junction table for the many-to-many between users and tasks." A 30-second fix here prevents a migration in Segment 4.

Step 3. Run the DDL prompt with the corrected entity list as context.

```
ROLE: You are a senior database architect specializing in SQLite
for embedded production apps.

INSTRUCTIONS: Generate the complete schema for the entities below.
Include:
  - CREATE TABLE statements with column names, types, NOT NULL,
    DEFAULT values, and ISO timestamp columns where appropriate
  - PRIMARY KEY and FOREIGN KEY constraints (ON DELETE behavior)
  - CHECK constraints for enums (status, priority)
  - 3-5 recommended indexes, each with a one-line justification
    tied to a specific user story from the PRD
  - A seed-data INSERT block with 1 team, 3 users, and 5 tasks

CONTEXT: TaskFlow Mini, single-file SQLite database, no migrations
tooling for the MVP. The app uses better-sqlite3 from Node.js.

EXAMPLE output ordering: CREATE TABLE in dependency order first,
then CREATE INDEX, then INSERT statements.

[paste the entity list from Step 1 here]
```

Step 4. Save to db/schema.sql and verify it loads.

```
mkdir db
# (paste the schema into db/schema.sql)
sqlite3 db/taskflow.db < db/schema.sql
sqlite3 db/taskflow.db "SELECT name FROM sqlite_master WHERE type='table';"
```

What you should get

A schema.sql with 4-5 tables (teams, users, tasks, plus task_assignees if you caught the many-to-many), CHECK constraints on status and priority, 3-5 indexes each with a comment justifying it ("supports the active-tasks-by-priority query from story P0-3"), and seed data that lets the app render something on first load. The sqlite3 verification step prints the table list and proves the schema loads cleanly. If it errors, paste the error back to the model — never hand-edit AI-generated DDL on the first pass.

Segment 3 — Generate Three Mermaid Diagrams

What to Learn

Mermaid.js is text. AI is great at text. We will produce three diagrams — a request-flow flowchart, a sequence diagram for the create-task flow, and an ER diagram of the schema we just wrote — and save each as a .mmd file. In Segment 4 the running app will fetch these files and render them live in the browser. That is the payoff of diagrams-as-code: the same file is documentation, source code, and a UI feature, all at once.

Prompt A — Architecture Flowchart. Paste this into Claude.ai:

```
Generate Mermaid.js using "graph LR" showing the request flow for
TaskFlow Mini: Browser SPA → Express server → SQLite database.

Include a decision diamond labeled "valid task payload?".
On invalid: return 400 Bad Request to the browser.
On valid: INSERT into tasks, return 201 Created with the new task ID.

Use clear node labels. Output ONLY the Mermaid code, no explanation,
no markdown fences.
```

Save as docs/diagrams/architecture.mmd.

Prompt B — Sequence Diagram.

```
Generate a Mermaid.js sequenceDiagram with autonumber for the
"create task" flow in TaskFlow Mini.

Participants (in this order): User, Browser, Express API, SQLite.

Show: form submit, client-side validation, POST /api/tasks,
```

```
server-side validation, INSERT into tasks, returned task ID,
201 response, render in UI list.

Output ONLY the Mermaid code.
```

Save as docs/diagrams/sequence.mmd.

Prompt C — ER Diagram. Paste your schema.sql at the placeholder.

```
Generate a Mermaid.js erDiagram for the schema below.

Show cardinality labels using Mermaid notation (||--o{, }o--||, etc.)
on every relationship. Include 3-5 key attributes per entity with
their types. Do not include every column — pick the ones that matter
for understanding the model.

Output ONLY the Mermaid code.

[paste the contents of db/schema.sql here]
```

Save as docs/diagrams/er.mmd.

Validation step. Open mermaid.live in a browser. Paste each .mmd file in turn. All three should render. If one errors out, copy the error message back into the prompt — "the renderer says: <paste error>. Fix the syntax." — and the model will return a corrected version. This validate-paste-validate loop is the diagram equivalent of a unit test.

> **What you should get**
> Three .mmd files in docs/diagrams/, each rendering cleanly on mermaid.live. The architecture diagram shows the LR request flow with the decision diamond. The sequence diagram has autonumbered steps for every interaction. The ER diagram shows your tables with the right cardinality glyphs and a handful of attributes per entity. Each file is plain text, lives in Git, diffs cleanly, and is about to be a live feature in the app.

Segment 4 — Ship the App

> **What to Learn**
> We have a PRD, a schema, and three diagrams. Time to wire them into a running full-stack app. The backend is Express plus better-sqlite3 reading our AI-designed schema. The frontend is a single HTML file that uses Mermaid.js to render our generated .mmd files alongside a live task list. By the end of this segment, the diagrams stop being

> documentation and become a feature on the page — proving the AI-design output is real, not a slide deck.

Step 1. Initialize the project and install dependencies.

```
npm init -y
npm install express better-sqlite3
mkdir public
```

Step 2. Create server.js at the project root.

```
const express = require('express');
const Database = require('better-sqlite3');
const fs = require('fs');
const path = require('path');

const app = express();
const db = new Database('db/taskflow.db');
db.pragma('journal_mode = WAL');
db.pragma('foreign_keys = ON');         // critical — OFF by default in SQLite

app.use(express.json());
app.use(express.static('public'));

const TEAM_ID = 1;                       // single-team MVP per PRD §6

// GET /api/tasks — list tasks with comma-separated assignee names
app.get('/api/tasks', (req, res) => {
  const rows = db.prepare(`
    SELECT
      t.id,
      t.title,
      t.status,
      t.priority,
      GROUP_CONCAT(u.name, ', ') AS assignee
    FROM tasks t
    LEFT JOIN task_assignees ta ON ta.task_id = t.id
    LEFT JOIN users u           ON u.id       = ta.user_id
    WHERE t.team_id = ?
    GROUP BY t.id
    ORDER BY
      CASE t.priority WHEN 'high' THEN 1 WHEN 'medium' THEN 2 ELSE 3 END,
      t.id DESC
  `).all(TEAM_ID);
  res.json(rows);
});
```

```
// POST /api/tasks — create task and (optionally) assign one user atomically
app.post('/api/tasks', (req, res) => {
  const { title, priority = 'medium', assignee_id = null } = req.body;

  if (!title || title.length < 3) {
    return res.status(400).json({ error: 'title must be 3+ chars' });
  }
  if (!['low', 'medium', 'high'].includes(priority)) {
    return res.status(400).json({ error: "priority must be 'low', 'medium', or 'high'" });
  }

  const insertTask = db.prepare(`
    INSERT INTO tasks (team_id, title, status, priority)
    VALUES (?, ?, 'open', ?)
  `);
  const insertAssignee = db.prepare(`
    INSERT INTO task_assignees (task_id, user_id) VALUES (?, ?)
  `);

  const create = db.transaction(() => {
    const info = insertTask.run(TEAM_ID, title, priority);
    if (assignee_id != null) insertAssignee.run(info.lastInsertRowid, assignee_id);
    return info.lastInsertRowid;
  });

  try {
    res.status(201).json({ id: create() });
  } catch (err) {
    res.status(400).json({ error: err.message });
  }
});

// GET /api/diagrams — unchanged
app.get('/api/diagrams', (req, res) => {
  const dir = path.join(__dirname, 'docs', 'diagrams');
  res.json({
    architecture: fs.readFileSync(path.join(dir, 'architecture.mmd'), 'utf8'),
    sequence:     fs.readFileSync(path.join(dir, 'sequence.mmd'),     'utf8'),
    er:           fs.readFileSync(path.join(dir, 'er.mmd'),          'utf8'),
  });
});

app.listen(3000, () => console.log('TaskFlow Mini → http://localhost:3000'));
```

Step 3. Create public/index.html. This is the whole frontend — task list and create form on the left, live-rendered Mermaid diagrams on the right.

```
<!doctype html>
<html><head><meta charset="utf-8"><title>TaskFlow Mini</title>
<script src="https://cdn.jsdelivr.net/npm/mermaid@10/dist/mermaid.min.js"></script>
<style>
  body{font-family:system-ui,sans-serif;margin:0;display:grid;
       grid-template-columns:1fr 1fr;gap:20px;padding:20px;}
  h1{color:#1F497D;grid-column:1/-1;margin:0 0 10px;}
  .panel{border:1px solid #ddd;border-radius:8px;padding:16px;}
  .task{padding:8px;border-bottom:1px solid #eee;}
  .tabs button{margin-right:8px;}
  input,select,button{padding:6px;margin:2px;}
</style></head><body>
  <h1>TaskFlow Mini</h1>
  <div class="panel">
    <h2>Tasks</h2>
    <div id="tasks"></div>
    <h3>+ New Task</h3>
    <input id="t" placeholder="Task title (3+ chars)" size="30">
    <select id="p"><option>low</option><option selected>medium</option>
      <option>high</option></select>
    <button onclick="create()">Create</button>
  </div>
  <div class="panel">
    <h2>Live Architecture (AI-generated, rendered by Mermaid.js)</h2>
    <div class="tabs">
      <button onclick="show('architecture')">Architecture</button>
      <button onclick="show('sequence')">Sequence</button>
      <button onclick="show('er')">ER Diagram</button>
    </div>
    <div id="diagram" class="mermaid"></div>
  </div>
<script>
  mermaid.initialize({ startOnLoad:false, theme:'default' });
  let diagrams = {};

  async function load(){
    const tasks = await (await fetch('/api/tasks')).json();
    document.getElementById('tasks').innerHTML = tasks.map(t =>
      `<div class="task"><b>[${t.priority}]</b> ${t.title}
       <i>— ${t.assignee || 'unassigned'}</i></div>`).join('');
    diagrams = await (await fetch('/api/diagrams')).json();
    show('architecture');
  }
  async function show(key){
    const el = document.getElementById('diagram');
    el.removeAttribute('data-processed');
    el.innerHTML = diagrams[key];
```

```
    await mermaid.run({ nodes:[el] });
  }
  async function create(){
    const title = document.getElementById('t').value;
    const priority = document.getElementById('p').value;
    const r = await fetch('/api/tasks', {method:'POST',
      headers:{'Content-Type':'application/json'},
      body: JSON.stringify({ title, priority })});
    if (!r.ok) return alert((await r.json()).error);
    document.getElementById('t').value = '';
    load();
  }
  load();
</script></body></html>
```

Step 4. Run it.

```
node server.js
# open http://localhost:3000
```

Step 5. Demo to the class. Create a new task — watch it appear in the list (your AI-designed schema is now holding real rows). Click between the Architecture, Sequence, and ER tabs — three AI-generated Mermaid diagrams render live in the browser. The sequence diagram on the page literally describes the request you just made when you clicked Create. The ER diagram on the page literally describes the table the row was just inserted into. The PRD that started all of this is one Markdown file in docs/. One session, four artifacts, all wired together.

> **What you should get**
> A working app at http://localhost:3000 with two panels. On the left: a live task list reading from your AI-designed SQLite schema, plus a form that can create new tasks via POST /api/tasks. On the right: three tabs that render the AI-generated Mermaid diagrams — architecture, sequence, ER — each pulled from /api/diagrams. Create a task and the new row appears. Switch tabs and watch the diagrams re-render. Everything you see on screen — every column in the table, every node in the diagrams — traces back to one PRD that was generated in Segment 1. That is the AI-native engineering loop, end to end, in less than 20 minutes.

Wrap-Up — From PRD to Production-Shaped App

> **Remember**

> In less than 20 minutes you produced a PRD with 8 user stories and 24 Gherkin criteria, a normalized SQLite schema with constraints and justified indexes, three Mermaid diagrams, and a working full-stack app that renders all of those artifacts as live features. Every artifact came from one structured prompt session that started with a single PRD prompt. That is not a collection of demos — that is the AI-human handshake at full speed, and it is the loop you will use for every project from here on.

Key takeaways:

- The PRD is the prompt scaffolding for everything downstream. Get it right and the schema, the diagrams, and the code all line up. Get it wrong and you spend the next three segments fighting drift.
- Entity identification before DDL is non-negotiable. Skip that step and you will get hallucinated tables or a missing junction. Spend 30 seconds, save a migration.
- Mermaid plus AI means diagrams ship in the app, not in the wiki. They diff in pull requests, render in the browser, and update in the same commit as the code they describe.
- The running app is the proof. AI-design output is not slide deck theater — it is real software that real users can hit. Always end the loop at a working URL.
- Every module from here builds on TaskFlow. Keep this repo. Module 3 will add tests and CI/CD against this exact codebase.

> **Student Practice — Before Module 3**
> Pick an internal tool or feature you have built or shipped in the last year — the smaller the better. Run it through the full Module 2 loop on your own. Generate a PRD using the Segment 1 prompt. Run entity identification and DDL using the Segment 2 prompts. Generate one Mermaid diagram of the architecture using a Segment 3 prompt. Compare each AI-generated artifact to what actually exists in your codebase. Where did AI surface things you missed? Where did it miss things you knew? Submit the three artifacts plus a one-page reflection — what you would have done differently if you had had this loop a year ago — as Module 2 homework.