Module 3: Accelerated Code Generation & Debugging

Capstone: Build, Diagram, Debug — Ship a Full-Stack TaskFlow App

Overview

This capstone closes Module 3 by tying every lecture in this module — Zero-to-Draft Generation, API Endpoint Creation, Stack-Trace Debugging, and Edge-Case Isolation — into one continuous demo. You will build a full-stack TaskFlow application live in front of the class. Four claims will be proven on the way:

- AI eliminates the blank-canvas problem at every layer of the stack — data, API, UI, tests — when you prompt with R-I-C-E.
- Mermaid.js plus AI gives you architecture, ER, and sequence diagrams in seconds — and they stay in sync with the code because the same model produced both.
- The five-element debugging prompt converts a wall of red text into a one-line root cause and a working fix in under five minutes.
- Edge-case discovery is systematic, not creative, when you run the five-category taxonomy: boundary, null, concurrent, state, environmental.

By the end of the demo, the class will have watched a working full-stack app appear from a blank folder, watched a real bug get caught and fixed, and watched the test suite grow from happy-path-only to edge-case hardened — all of it AI-assisted, all of it under 20 minutes.

Before You Start

- Cursor (or VS Code with the Continue + Claude Code extensions) installed and signed in. We will use Cmd/Ctrl+I (Composer) for multi-file generation.
- Python 3.11+ and Node.js 20+ installed. Verify with python3 --version and node --version.
- Anthropic API key exported in your shell: export ANTHROPIC_API_KEY=sk-ant-…
- A new empty folder named taskflow/ open in Cursor. Inside it, create two empty subfolders: backend/ and frontend/.
- Open https://mermaid.live in a browser tab. Optionally install the Markdown Preview Mermaid Support extension in VS Code.
- A terminal open in the taskflow/ folder, ready to run uvicorn and npm commands. Windows users: use cmd or PowerShell — every command in this capstone has a Windows variant where it differs from macOS/Linux, so watch for the dual code blocks. Backslashes, .bat activation scripts, and quoted version pins are the three things that bite live.

During Hands-On — Five Segments at a Glance

- SEGMENT 1 — Architect with Mermaid: 3 diagrams from 3 prompts — ~4 min
- SEGMENT 2 — Generate the Backend: data model, schemas, routes, tests — ~7 min
- SEGMENT 3 — Generate the Frontend: React + Vite single-page UI — ~5 min
- SEGMENT 4 — Debug a Real Bug: the five-element trace prompt in action — ~4 min
- SEGMENT 5 — Edge-Case Sweep: the five-category taxonomy on one endpoint — ~5 min

Remember

Module 3 is where everything starts to compound. Module 1 gave you the cockpit. Module 2 gave you the architecture. Today you build something real, and you debug it, and you harden it — in the same session, with the same AI, in the same context window. That is the loop you will run for the rest of your career.

Segment 1 — Architect with Mermaid (~4 min)

What to Learn

Before you generate a single line of code, you commit to a contract. In Module 2 we drew systems on whiteboards. Today we draw them in mermaid because diagrams that live as code travel with the codebase, render in GitHub and Notion automatically, and — most importantly — can be generated by the same AI that will generate the code that implements them.

We will produce three diagrams in three prompts: a C4-style architecture diagram, an entity-relationship diagram, and a sequence diagram for the create-task flow. Each is a one-shot prompt. Each renders in mermaid.live or in any markdown viewer.

Step 1. Open Cursor's Composer (Cmd/Ctrl+I). Paste this RICE-shaped prompt for the architecture diagram:

```
Role: senior software architect.
Intent: produce a mermaid flowchart for a TaskFlow full-stack app.
Constraints:
  - top-down (graph TD)
  - three layers: Browser (React+Vite), Backend (FastAPI),
    Storage (SQLite for the demo)
  - show the JSON over HTTP edge between browser and backend
  - show SQLAlchemy as the edge between backend and storage
Output: ONLY the mermaid code block, no prose.
```

Step 2. Save the output to docs/architecture.mmd. Paste it into mermaid.live. The class should see boxes for Browser, FastAPI, and SQLite with labeled arrows between them. That is the system you are about to build, in 12 lines of mermaid.

Step 3. Now the data model. Same Composer, new prompt:

```
Role: data modeler.
Intent: produce a mermaid erDiagram for TaskFlow.
Entities:
  - Task: id (PK), title, description, status (open|in_progress|done),
          priority (low|med|high), due_date, created_at, updated_at
  - Tag:  id (PK), name (unique)
  - TaskTag: task_id (FK), tag_id (FK) — composite PK
Relationships:
  - Task many-to-many Tag via TaskTag
Output: ONLY the mermaid erDiagram block.
```

Step 4. Save as docs/er.mmd, render in mermaid.live. Three entities, one many-to-many. This is the contract that drives the SQLAlchemy models in the next segment — Lecture 1's context-loading principle made literal.

Step 5. Last diagram — the request lifecycle. New Composer prompt:

```
Role: backend engineer.
Intent: mermaid sequenceDiagram for POST /tasks creating a new task.
Participants: User, React UI, FastAPI Router, TaskService, SQLite.
Show: validation, DB insert, 201 response with the created task,
       and the React UI optimistically updating its task list.
Output: ONLY the mermaid sequenceDiagram block.
```

What you should get

Three .mmd files saved in docs/. All three render cleanly in mermaid.live. The architecture diagram has 3 nodes and 2 labeled edges. The ER diagram has 3 entities with the correct cardinality. The sequence diagram shows 5 lifelines and the round-trip from user click to DB write to UI update.

Every diagram took one prompt. Every diagram uses the same domain language we will use in the code. That alignment is not a coincidence — it is what contract-first AI development looks like.

Segment 2 — Generate the Backend (~7 min)

What to Learn

This is Lecture 1 (Zero-to-Draft) and Lecture 2 (API Endpoint Creation) running back to back. We follow the proven pattern: generate in dependency order — data model first, then schemas (three flavors: Create, Update, Response), then router, then integration tests in the same session.

We use the five-part API prompt from Lecture 2: framework, domain, operations, validation, behavior. Tests are not a separate step. Tests are generated from the same context window as the routes.

Step 1. Open a terminal in the backend/ folder and create the virtual environment, then install dependencies. Pick the line for your OS — the differences trip people up live, so call them out explicitly.

macOS / Linux:

```
python3 -m venv .venv
source .venv/bin/activate
pip install fastapi uvicorn "sqlalchemy>=2" pydantic httpx pytest pytest-asyncio
```

Windows (cmd):

```
python -m venv .venv
.venv\Scripts\activate.bat
pip install fastapi uvicorn "sqlalchemy>=2" pydantic httpx pytest pytest-asyncio
```

Note: on Windows the > in 'sqlalchemy>=2' must be inside double quotes — otherwise cmd treats it as a redirection operator. While the install runs, open Composer.

Step 2. Generate the database wiring first — this file is the foundation everything else imports from, so we make it explicit instead of leaving it as a fallback. In Composer:

```
Role: senior Python backend engineer.
Intent: backend/app/database.py — SQLAlchemy 2.x sync setup.
Output exactly:
  - engine bound to "sqlite:///./taskflow.db"
    (with connect_args={"check_same_thread": False} for SQLite)
  - SessionLocal: sessionmaker(autocommit=False, autoflush=False, bind=engine)
  - Base: declarative_base()
  - get_db() generator dependency that yields a Session and closes it
File path: backend/app/database.py
```

Step 3. Generate the data model. Paste the ER diagram from Segment 1 into Composer as context, then prompt:

```
Role: senior Python backend engineer.
Intent: SQLAlchemy 2.x ORM models for TaskFlow.
Constraints:
  - import Base from app.database (do NOT redefine Base here)
  - mapped_column, type-annotated
  - match the entities and relationships in the mermaid erDiagram above
  - Task <-> Tag many-to-many: use ONE pattern only — the explicit
    TaskTag association object — and use back_populates on every
    relationship. Do not also add secondary= shortcuts.
  - status and priority are enums
  - created_at / updated_at default to UTC now
File path: backend/app/models.py
```

Step 4. Generate the three schemas — Create, Update, Response — exactly as Lecture 2 prescribes. Note the explicit required-vs-optional breakdown for TaskCreate. Without this, models tend to copy every model field as required, which produces a brittle API that 422s on any minimal payload — a real-world bug pattern worth calling out to the class:

```
Role: senior Python backend engineer.
Intent: Pydantic v2 schemas for the Task model.
Output 3 separate classes:

  - TaskCreate — required vs optional matters here:
      * title: str  (REQUIRED — the only field a client must send)
      * description: Optional[str] = None
      * status: TaskStatus = TaskStatus.open
      * priority: TaskPriority = TaskPriority.med
      * due_date: Optional[datetime] = None
      (no id, no timestamps — those are server-generated)

  - TaskUpdate — every field Optional[...] = None for partial updates

  - TaskResponse — id, title, description, status, priority,
                   due_date, created_at, updated_at
                   (everything; no internal-only fields exist)

Constraints:
  - reuse the TaskStatus and TaskPriority enums from app.models
    (do NOT redefine them here)
  - ConfigDict(from_attributes=True) on TaskResponse only
File path: backend/app/schemas.py
```

Step 5. Generate the router using the five-part API prompt — note every element is explicit, including where the DB session comes from. The 'do not generate a placeholder' line is a hard-won lesson — without it, models sometimes emit a stub that raises NotImplementedError at runtime:

```
Role: senior FastAPI engineer.
Framework: FastAPI with synchronous route handlers (not async).
Domain: Task resource (see models.py and schemas.py — paste both).
Operations:
  - GET    /tasks          list
  - GET    /tasks/{id}     fetch one
  - POST   /tasks          create
  - PATCH  /tasks/{id}     partial update
  - DELETE /tasks/{id}     delete
Validation: Pydantic v2 strict mode.
Behavior:
  - 201 on POST, 204 on DELETE, 404 if not found, 422 on validation
  - all handlers are def (sync) — we are using sync SQLAlchemy
  - DB session: import get_db from app.database and use it as the
    Depends() dependency. DO NOT generate a placeholder, stub,
    NotImplementedError, or local get_db_session — use the real one
    from app.database directly.
Documentation: comprehensive docstrings (summary + description) on every route.
File path: backend/app/routers/tasks.py
```

Step 6. Same session — Lecture 2's iron rule — generate the integration tests now while the AI still has every schema, status code, and error condition in context:

```
Now generate pytest integration tests for all five routes using
FastAPI's TestClient (sync, since our router is sync) and an
in-memory SQLite test fixture that overrides the get_db dependency.

Cover:
  - happy path for each route
  - 404 on missing id (GET, PATCH, DELETE)
  - 422 on bad payload
  - IMPORTANT: a "minimal payload" test for POST /tasks that sends
    ONLY {"title": "x"} and asserts 201 plus that defaults applied
    (status=open, priority=med, description=None, due_date=None).
    This catches the over-strict-schema bug where TaskCreate
    accidentally marks every field as required.

One file: backend/tests/test_tasks.py
```

Step 7. Wire it up. This step has one easy-to-miss detail that will cost you a 500 error live if you skip it: SQLAlchemy only creates tables for models it knows about, and it only knows about models that have been imported. So main.py must import the models module before calling create_all — even if it doesn't use it directly. Composer prompt:

```
Role: senior FastAPI engineer.
Intent: backend/app/main.py — wire the FastAPI app together.
Constraints:
  - import Base and engine from app.database
  - IMPORTANT: import the models module (from app import models)
    BEFORE calling create_all, so SQLAlchemy registers every table
  - use the modern lifespan context manager (not @app.on_event)
  - on startup, call Base.metadata.create_all(bind=engine)
  - create FastAPI app with title "TaskFlow API"
  - include the tasks router from app.routers.tasks
  - keep it under 25 lines
File path: backend/app/main.py
```

Step 8. Run it. From backend/ with the venv active: uvicorn app.main:app --reload --port 8080. We use 8080 because port 8000 is in a Windows reserved range on many machines and will fail with WinError 10013 — picking 8080 up front avoids a live-demo derail. Open http://localhost:8080/docs. The Swagger UI shows all five endpoints with full descriptions — documentation as a byproduct, exactly as Lecture 2 promised. A taskflow.db file should appear in backend/, confirming create_all ran. Then run pytest -q. All happy-path and basic-error tests pass.

What you should get

A working FastAPI backend with Task CRUD, three Pydantic schemas, in-memory SQLite, autogenerated Swagger UI, and a green pytest suite — generated in five prompts, in roughly seven minutes.

Every route has a docstring. Every status code is explicit. Internal fields never appear in the response model. The test suite was generated from the same context as the implementation, so they match by construction.

Segment 3 — Generate the Frontend (~5 min)

What to Learn

Same RICE pattern, different language. Frontend code is one of the highest-yield places for AI generation because it is dense with boilerplate (state hooks, fetch wrappers, form handlers, conditional rendering) that follows identical shapes from app to app.

We will scaffold a Vite + React app, then generate a single-page TaskFlow UI in one Composer call by passing the OpenAPI schema from Segment 2 as context. The UI will consume the same contract the backend exposes — that is the value of contract-first thinking carried forward.

Step 1. From the taskflow/ root: cd frontend && npm create vite@latest . -- --template react && npm install. Skim the generated files so the class sees what Vite gives you for free.

Step 2. Open backend Swagger (http://localhost:8080/openapi.json) in a browser tab. Copy the JSON. We will pass it to the AI as the contract.

Step 3. In Composer, paste the OpenAPI JSON above the prompt, then:

```
Role: senior React engineer.
Intent: a single-file TaskFlow UI in src/App.jsx.
Constraints:
  - functional component, hooks only (useState, useEffect)
  - fetch from http://localhost:8080 (CORS will be enabled in Step 5)
  - render: header "TaskFlow", a "New Task" form (title, priority,
    status), and a list of tasks below
  - each task: title, status badge, priority badge, Edit + Delete
  - inline edit on click, optimistic update, rollback on error
  - plain CSS in src/App.css — no UI library
  - error states displayed inline, not via alert()
Examples: use the OpenAPI schema above as the source of truth for
          request/response shapes — do not invent fields.
File path: frontend/src/App.jsx (and App.css)
```

Step 4. Read the output. Apply Lecture 1's five-step review: read every line, check every import, trace control flow, ask what's missing (loading state? empty state?), run the linter. If anything looks off, do not regenerate — iterate. Say in chat "add a loading skeleton while the initial fetch is in flight" and watch the AI patch the existing code.

Step 5. CORS. Back in backend/app/main.py, ask the AI: "Add CORSMiddleware allowing http://localhost:5173 for all methods." One prompt, three lines of code.

Step 6. From frontend/: npm run dev. Open http://localhost:5173. Create a task. Watch it appear. Edit it inline. Delete it. The full stack is alive.

What you should get

A working React UI talking to a working FastAPI backend talking to SQLite. Create / read / update / delete all functional. The UI uses the exact field names from the backend schema because the schema was the prompt's source of truth.

If you tally what just happened: in twelve minutes you generated a backend, a frontend, three diagrams, and a test suite. That is the productivity claim from Lecture 1 — 80–85% reduction on scaffolding work — playing out at the system level.

Segment 4 — Debug a Real Bug (~4 min)

What to Learn

Lecture 3 in action. We will inject a realistic logic bug — a missing commit() that silently drops data — and walk through the exact seven-step debugging protocol: reproduce, diagnose, isolate, build the debug prompt, evaluate, fix, document.

The point is not the bug. The point is the muscle memory of the protocol. Logic errors with no stack trace are some of the hardest bugs to debug — Lecture 3, Slide 11, called this out as the place AI debugging shines brightest, because the AI can rubber-duck the code while you describe the discrepancy.

Step 1. Inject the bug into patch_task (the PATCH handler), not create_task. POST is a bad target for this bug class because its response model requires database-assigned fields — id, created_at, updated_at — that only exist after commit, so a missing-commit bug there fails loudly with a ResponseValidationError. PATCH operates on a row that already exists, so the in-memory object already has all the required fields populated. Find the patch_task handler and comment out session.commit() — for example, change session.commit() to # session.commit(). Save. The handler still returns 200 with the updated values (because the in-memory object reflects the setattr changes), but the transaction never commits, so the change is rolled back when the session closes. This is a classic silent-failure bug: no exception, no red text, just data quietly reverting — exactly the kind of thing Lecture 4's edge-case taxonomy calls out.

Step 2. Reproduce. From the UI, click an existing task to edit it inline. Change the title. Save. Watch the title flip back to the original value almost immediately. What is happening: the frontend optimistically shows the new title, the PATCH returns 200, then the frontend re-fetches the task list to stay in sync — and that GET reads from the database, where the change was never committed. The revert is real-time, which makes this an even cleaner demo than waiting for a refresh. PATCH returned 200 with the new values, but the database never received them, and the next read exposes the truth.

Step 3. Frame the situation. Unlike a runtime error, there is no stack trace here — this is a logic error, which Lecture 3 explicitly distinguishes. Point to the class: when there is no trace, your prompt has to describe the discrepancy precisely — what you expected versus what you observed. That is the rubber-duck-upgrade pattern from Lecture 3, Slide 11.

Step 4. Build the debug prompt. Logic errors use a slightly different shape than runtime errors — no trace, but a precise expected-vs-actual. Paste this into Composer:

```
(1) ERROR DESCRIPTION
No exception raised. No 500. PATCH /tasks/{id} returns 200 with
the updated task in the response body. In the UI, the new value
flashes briefly and then reverts to the original — because the
frontend re-fetches GET /tasks after save, and that GET reads
from the database where the change was never persisted. The
taskflow.db file inspected with the sqlite3 CLI confirms the row
still has the old values.

(2) RELEVANT CODE
[paste backend/app/routers/tasks.py — the PATCH handler]
[paste backend/app/database.py — get_db and SessionLocal]

(3) EXPECTED VS ACTUAL
Expected: PATCH /tasks/{id} updates the row in the database;
          a follow-up GET returns the new values.
Actual:   PATCH returns 200 with the new values in the response,
          but a follow-up GET returns the original values.
          The change is silently lost.

(4) WHAT I HAVE TRIED
Confirmed the request body validates (PATCH returns 200, not 422).
Confirmed get_db yields a real Session.
Confirmed the Task object returned by the handler has the new
values — so setattr ran correctly. The mutation just isn't
reaching the file.

(5) FIX REQUEST
Walk through the PATCH handler step by step. Identify where
the in-memory mutation is being lost between the setattr loop
and the response.
```

Step 5. Read the AI's response out loud. It should walk through the patch_task handler step by step, identify that setattr applied the changes to the in-memory Task object, but session.commit() was never called — which is why the response shows the new values (the in-memory object is what FastAPI serializes) but the database transaction was rolled back when the session closed. It will propose uncommenting the commit. Then ask the follow-up that Lecture 3 told you to ask: "What other handlers in this codebase could have the same class of bug?" The AI will scan create_task and delete_task and flag any that are missing a commit too.

Step 6. Apply the fix in isolation. Re-run the failing test (pytest -q tests/test_tasks.py::test_update_task) and confirm it passes. This is the validation step from Lecture 3 — never accept a fix you have not run.

Step 7. Pattern card. End the Composer session with: "Summarize this debugging session as a reusable pattern card: error type, root cause, fix, prevention tip — four lines." Paste the result into docs/debug-patterns.md. That file is now an asset that compounds across every future debugging session.

What you should get

Bug found in under two minutes from prompt to root cause. Fix verified in isolation. Pattern card saved. The full pytest suite passes again. The class has just watched a five-minute debug session that, without the AI and the protocol, would have been a thirty-minute slog through SQLAlchemy session-management documentation trying to figure out where the data went.

Remember — the bug Lecture 4 was designed to catch

Before we generate edge-case tests, point the class at the schema prompt from Segment 2, Step 4. Without the explicit required-vs-optional breakdown, models will routinely mark every field of TaskCreate as required — producing an API that returns 422 on any minimal payload like {"title": "buy milk"}. The happy-path tests would not have caught it because they always send full payloads. This is exactly the null / missing-input category from Lecture 4's taxonomy, and it is the most common production bug in CRUD APIs. The systematic edge-case sweep we are about to run is your defense against this whole class of failure.

Segment 5 — Edge-Case Sweep (~5 min)

What to Learn

Lecture 4 closes the module. Happy-path tests pass — that is necessary, not sufficient. We now run the five-category taxonomy — boundary, null, concurrent, state, environmental — against POST /tasks specifically, generate edge-case tests in their own isolated file, and watch which ones fail.

The discipline rule is non-negotiable: edge-case tests live in their own file with their own pytest mark. When they fail, we fix the source code, never the test. The original green suite must stay green.

Step 1. In Composer, paste the create_task handler and the TaskCreate schema as context, then prompt the cascade exactly as Lecture 4 specifies:

```
(a) Enumerate edge cases for POST /tasks across the five categories:
    BOUNDARY values (length limits, date extremes),
    NULL / missing inputs (every field, alone and combined),
    CONCURRENT requests (same title, simultaneous creates),
    STATE order (create after delete-all, create with stale tag id),
    ENVIRONMENTAL (UTF-8 emoji in title, locale-formatted date).

(b) For each, mark whether it would cause a SILENT failure
    or an EXPLICIT exception.

(c) Pick the three most likely to reach production undetected.

(d) Generate a parametrized pytest file:
       backend/tests/test_tasks_edge_cases.py
    using @pytest.mark.edge_cases for every test.
```

Step 2. Save the generated file. Run only the new tests: pytest -m edge_cases -q. Some pass, some fail. The failures are the bugs your happy-path suite never caught.

Step 3. Pick one failure — typically the boundary on title length, or the missing-field case if your schema didn't enforce it strictly. Show the class the workflow: do not modify the failing test. Modify the schema, the handler, or the model. Re-run.

Step 4. Verify nothing else broke: pytest -q (the full suite, not just edge cases). Both the original tests and the edge-case tests pass. That dual green is the proof that you fixed the bug without breaking the happy path.

Step 5. Generate the team-level asset. Final Composer prompt:

```
Based on the edge cases we identified for POST /tasks,
generate a reusable one-page Edge Case Checklist for any
REST endpoint in this codebase. Group by the five categories.
Output: docs/edge-case-checklist.md
```

Step 6. Commit everything. The repo now contains: backend with happy-path and edge-case tests, frontend, three mermaid diagrams in docs/, a debug pattern card, and an edge-case checklist. That is the deliverable.

What you should get

Two pytest test files, both passing: tests/test_tasks.py (happy path, generated in Segment 2) and tests/test_tasks_edge_cases.py (hardened, generated in Segment 5). A docs/ folder containing architecture.mmd, er.mmd, sequence.mmd, debug-patterns.md, and edge-case-checklist.md. A Git history that tells the whole story.

Wrap-Up — The Full Stack, Hardened, in 20 Minutes

Remember

What just happened in under 20 minutes: three architecture diagrams, a typed Pydantic-and-FastAPI backend with passing tests, a React frontend that talks to it, an injected real-world bug caught and fixed with a five-element prompt, and an edge-case test suite that is hardening the API against the kind of failures that reach production. Every artifact was AI-generated. Every artifact was reviewed. Every fix was verified.

Module 3 is the seam where prompting becomes engineering. You are not a developer who uses AI to autocomplete. You are an engineer who orchestrates AI through a contract, a generation pass, a debug pass, and a hardening pass — the four lectures of this module, executed as one workflow.

Key takeaways

- Contract-first thinking — mermaid diagrams, OpenAPI schemas, Pydantic models — is what keeps AI-generated code from drifting away from your intent. The contract is the prompt's source of truth.
- Generate in dependency order. Models first, then schemas in three flavors (Create, Update, Response), then routes, then tests — and tests live in the same context window as the code they test.
- The five-element debug prompt — error, trace, code, expected vs actual, what you tried — turns a forty-line stack trace into a four-line answer almost every time.
- The five-category edge-case taxonomy — boundary, null, concurrent, state, environmental — is your structural antidote to imagination-limited testing. Run it on every endpoint that touches data.

- Pattern cards and checklists turn one debugging session into team-wide knowledge. Capture every win in 30 seconds. After ten sessions, you have a playbook.

Student Practice — Module 3 Lab

Take the TaskFlow demo built today and extend it. Add Tags as a real resource: TagCreate / TagUpdate / TagResponse, full CRUD, and the many-to-many join through TaskTag. Update the mermaid diagrams. Generate the integration tests in the same session. Run the five-category edge-case sweep on POST /tags and PATCH /tasks/{id}/tags. Deliverables: a pull request containing the new endpoints, both green test suites, an updated er.mmd, and a one-page edge-case checklist for tag operations. Time-box: 90 minutes. If you finish under 60, that is your new baseline for full-stack feature work.