Lab 7 — Build an App with Agentic AI

Coding with Claude Code

You built the idea board one layer at a time, by hand: a database, an application programming interface (API) in front of it, a page in front of that, and a container on Railway around all three. This lab is about building with **Claude Code**, an AI coding agent that works in your terminal. It reads your repo, writes the code, runs the commands, and checks its own work against tests. You direct it through a whole app, from an empty repo to a live web address (URL), and it writes most of the code.

The app is **5C Dish Ratings**: anyone can add a dining hall dish, rate it from 1 to 5 stars, and see which dishes are worth the walk. It has the same skeleton as the idea board. A dish is an idea, a rating is a vote, and `avg_rating` is `vote_count` with one more aggregate. So you already know what every part of it should look like, and that's the point. **You can only check what you understand.** Claude can produce a whole app in an afternoon, but it can't tell you whether that app is right. Setting it up to succeed, steering it, and checking what it built are still yours, and they're what this lab grades.

What to build

1. **A `CLAUDE.md`, committed before any app code.** It pins the stack, says how to run things, and sets the rules. See step 0.
2. **5C Dish Ratings, meeting `CONTRACT.md`.** The API, and a page in plain HTML (HyperText Markup Language), CSS (Cascading Style Sheets) and JavaScript, served by the same app. Lab 6's standard for the page still holds: semantic, accessible, honest about the network, and usable at phone width.
3. **Deployed to Railway, with Postgres.** Lab 6's container, plus a Postgres database beside it, reached through `DATABASE_URL`, with a health check that asks the database. A live `https://` URL that works from a phone on cell data, and ratings that survive a redeploy, which Lab 6's didn't.
4. **Built in slices, each one a reviewed pull request (PR).** At least two merged PRs, and three or four is more natural.
5. **One workflow extension, committed to the repo:** a skill, a hook, a subagent, or a Model Context Protocol (MCP) server that makes Claude better at this project.
6. **One feature beyond the contract**, which you choose and can defend.
7. **A concept map in your `README.md`**: every piece of your app next to the piece of the idea board it corresponds to, and where it lives in your code.

Stretch: an AI summary of each dish's reviews, from a large language model (LLM), using your Glassroom API key (`CONTRACT.md` §7).

Get started

1. A repo and a codespace

Lab 7 gets a repo of its own, made the same way as your Lab 6 one.

1. Open the course template, https://github.com/joshbrake/e190bo-devcontainer-template, and click **Use this template → Create a new repository**. **Owner: your own account. Name: `e190bo-lab-07`. Visibility: Private.** Don't click Fork.
2. In `e190bo-lab-07`, click **Code → Codespaces → Create codespace on main**, and wait for `E190BO environment ready.`

Check:

- ☐ Your `e190bo-lab-07` codespace is open, with a terminal at the repo root.

2. Claude Code

Claude Code is the tool this lab is built around, and every example in this handout uses it. It needs a paid Claude account (Pro or higher), signed in as you. If you can't get one, talk to us before you start.

It comes installed in the course codespace, so in the codespace's terminal:

```
claude --version    # check it's there
claude              # start it, from the repo root
```

If `claude` isn't found, install it with `curl -fsSL https://claude.ai/install.sh | bash`, then open a new terminal.

The first time, it asks you to sign in. In a codespace it shows a link: open it, sign in, and paste the code it gives you back into the terminal. If sign-in won't finish, check that you're signing in with the account that has Pro (or higher), since the free plan doesn't include Claude Code; `/login` starts it again. Type `/status` to see which account it's using, and `/exit` (or `Ctrl-D` twice) to leave.

Check:

- ☐ `/status` shows your own account.

Start `claude` from the **repo root**, so it can see your `CLAUDE.md`, your `src/`, and your `Dockerfile` all at once.

3. The starter: the contract and the tests
From the repo root:

```
mkdir src && cd src
curl -fLO https://course.aiprototypes.org/resources/files/eng190bo-fa26-lab-7-starter.zip
python -m zipfile -e eng190bo-fa26-lab-7-starter.zip .
rm eng190bo-fa26-lab-7-starter.zip
git add . && git commit -m "Add Lab 7 starter" && git push
cd ..    # back to the repo root
```

`claude` and `git` run from the repo root, and `pytest` from `src/`. A "file not found" error usually means you're in the other one.

Check:

- ☐ `ls src` shows `CONTRACT.md` and `tests`.
- ☐ Your repo on github.com shows the commit.

That's all the starter is: no app code, only the spec.

- **`CONTRACT.md`**: what the API promises, in the same shape as the idea board's contract. Read it all, especially the three ⚠ boxes: each one is a place where the idea board's answer, copied over, is wrong.
- **`tests/`**: the contract as tests, which talk to your app over HTTP (Hypertext Transfer Protocol) the way a browser would. They're the same kind of checks as Lab 6's `check.py`, written for `pytest`, the test runner your agent already knows: `pytest` runs them all, and *Running the tests* has the rest. `test_repo.py` reads your git history instead.

The tests are your agent's definition of done. Leave them exactly as they came. `test_repo.py` checks that nobody, you or your agent, has edited them.

4. A first question
Before Claude writes anything, watch it read. Start `claude` and ask it a question, not a task:

> Read src/CONTRACT.md. In two sentences: what does GET /dishes return for a dish nobody has rated, and why? Don't change anything.

The tool calls that scroll past are the agent's loop at work: it reads the contract before it answers.

Check:

- ☐ You saw it read `src/CONTRACT.md` before it answered.
- ☐ Its answer says `avg_rating` is `null`, not `0`.

Then type `/clear` to start a fresh session.

Step 0 — Write your `CLAUDE.md` first

An agent starts every session knowing nothing about your project. `CLAUDE.md` is the file Claude Code reads at the start of every session, before your first message, so it's where the things that are always true go. Write it **before** the agent writes any code, at the **repo root**, next to `README.md`. Write it **yourself**, too: if Claude writes it, they're Claude's rules, and the check-in asks you to explain yours.

Pin the stack. Ask an agent for "a web app that rates dishes" and it may hand you React, Node, Prisma, and a build step, all of which work, and none of which you can read. So say exactly what to use: the stack you already know.

Layer	Use
API	FastAPI and SQLModel, in `src/main.py`
Database	SQLite in the codespace; Postgres on Railway, from `DATABASE_URL`; tables from `create_all`
Page	plain HTML, CSS and JavaScript in `src/web/`, served by the same app; no framework, no build step
Deploy	a `Dockerfile` at the repo root, on Railway, with Railway's Postgres

`test_repo.py` checks that your `CLAUDE.md` names FastAPI, SQLModel, Postgres and Docker.

A shape that works, a page at most:

```
# CLAUDE.md

One or two sentences: what this app is, and where the spec and the tests are.

## Stack — don't change it
(the table above, in your words)

## Commands
How to run the app, the tests, and the tests against production. Which folder each runs from.

## Rules
What must always be true. Start with: never edit src/tests/ or src/CONTRACT.md.
Add one of your own, from the contract's three ⚠ boxes.

## Ask me before
Things the agent must not do on its own: add a dependency, delete data, push, merge.
```

Commit it in a commit of its own, from the repo root:

```
git add CLAUDE.md && git commit -m "Add CLAUDE.md" && git push
```

Test it. In Claude Code, type `/clear`, then ask:

> Without reading any files: what stack must you use here, and what must you never edit?

Check:

- ☐ It answered without reading any files: `CLAUDE.md` was already in its context.
- ☐ Both answers match what you wrote.
- ☐ From `src/`, `pytest -m repo -k claude_md` shows 1 passed and 1 skipped. The pass is the check that your `CLAUDE.md` names the stack. The skip is the check that it came before any app code, which waits until there's app code to compare it with.

If an answer is wrong, the fix is in `CLAUDE.md`, not in the chat. Edit it, commit again, `/clear`, and ask again.

Then keep it current. When the agent does something you don't want, fix the code **and** ask whether the fix belongs in `CLAUDE.md`, so it doesn't happen again next session. The best rules in it are the ones a mistake taught you, and the check-in asks for one.

Seven things you need to know

You know the app. These are about working with Claude Code while it builds it.

1. The loop: explore, plan, build, check, commit

The agent is fast at writing code and bad at knowing whether it's right. Each slice of work goes round the same loop, and the slow steps are the ones that matter:

1. **Explore.** Have it read the contract, the tests, and the code that exists, before it changes anything.
2. **Plan.** Press `Shift-Tab` until the prompt says **plan mode**. In plan mode the agent can read and run commands but can't edit files, and it ends by showing you a plan. Read it. If it says React, or a new table you didn't ask for, say no now, while it's one line of plan and not three hundred lines of code.
3. **Build.** Approve the plan and let it work.
4. **Check.** Run the app, run the tests, and look at the page yourself. "The agent said it works" isn't checking.
5. **Commit.** A small commit, on a branch, of something that works.

The failure mode is skipping 2 and 4.

2. Give the agent a way to check its own work

An agent that can run a check will keep going until the check passes. An agent that can't will tell you it's done. Your tests are that check, so say so:

> Read src/CONTRACT.md and src/tests/test_contract.py. Build the API in src/main.py until `pytest -m contract` passes with the server running, except the test that `GET /` serves the page: that one waits for the page. Don't touch the tests.

The tests can't see the page, though. For the page, you are the check: open it, use it, and break it (the Lab 6 way: stop the server, throttle the network). Or give the agent eyes: the Playwright MCP server (thing 6) lets it drive a real browser and look at what it built.

3. Slices, branches, and reading the diff

Don't ask for the whole app at once. Ask for one slice that you can check on its own, on its own branch:

```
git switch main && git pull
git switch -c api
```

When the slice works, push the branch, open a PR on GitHub (base `main`), and **read every line in Files changed**. This is where you catch what the agent got wrong, before it reaches `main`, and from `main` reaches Railway. Leave at least one review comment on a line: what it does, or why you're keeping or changing it. Then merge it on GitHub,

with any of the three buttons (**Create a merge commit**, **Squash and merge**, or **Rebase and merge**), and `git switch main && git pull`.

While you read, ask:

- Does this match the contract, and the stack in `CLAUDE.md`?
- Is there anything here I couldn't explain? Ask the agent to explain it, or take it out.
- What did it change that I didn't ask for?
- Does the request body have its own class? Is the rule checked in the API, or only on the page?
- Any `innerHTML`, a hardcoded host, a secret, or a test that changed?

The agent can make commits and open PRs for you (it uses the `gh` command), and that's fine. The reading is the part you can't hand over.

4. Context is the scarce resource

Everything in a session (your messages, every file the agent read, every command's output) sits in its **context window**, and it gets worse at its job as that fills. So:

- **One task per session.** `/clear` when you move to the next slice. `CLAUDE.md` is reloaded; the clutter isn't.
- **Correct early.** Press `Esc` the moment it heads the wrong way; your work so far is kept. `Esc Esc` (or `/rewind`) goes back to an earlier point.
- **After two failed corrections, start over.** `/clear`, and write a better first prompt with what you learned. Arguing with a confused session makes it more confused.
- `/context` shows how full the window is.

Rewinding undoes the agent's file edits, but not what its shell commands did, and not your database. Git is your real undo, which is one more reason to commit small.

5. Protect the spec

When a test fails, the shortest path to green is to change the test. Agents take that path more often than you'd think, and it hides the bug instead of fixing it. A rule in `CLAUDE.md` *asks* the agent not to. A **hook** makes sure: a command Claude Code runs before or after a tool call, which can block it. This one, in `.claude/settings.json`, runs a script before every edit:

```
{
  "hooks": {
    "PreToolUse": [
      {
        "matcher": "Edit|Write",
        "hooks": [
          { "type": "command", "command": "python3 \"$CLAUDE_PROJECT_DIR\"/.claude/hooks/
protect-spec.py" }
        ]
      }
    ]
  }
}
```

The script reads the tool call as JavaScript Object Notation (JSON) on its standard input, and if the file is under `src/tests/`, prints why and exits with code `2`, which blocks the edit and tells the agent the reason. Writing that script is a good first workflow extension (thing 6).

6. Teach your agent about this project

Four ways to make the agent better at *this* project, all as files in your repo. You need at least one, committed. Pick the one that fixes something that actually went wrong.

Extension	What it is	Lives in	A good one for this lab
Skill	Instructions for a task the agent loads when it's relevant, or when you type `/name`	`.claude/skills/<name>/SKILL.md`	`/check`: start the server, run every test, report
Hook	A command that runs on an event, every time	`.claude/settings.json`	block edits to `src/tests/` (thing 5)
Subagent	A helper with its own context and instructions, for a job the main agent hands off	`.claude/agents/<name>.md`	a reviewer that reads a diff against `CONTRACT.md` and your checklist
MCP server	A program that gives the agent new tools	`.mcp.json`	Playwright, so it can open the page in a browser

A skill is a Markdown file with a short header, and the `description` is how the agent decides when to use it:

```
---
name: check
description: Run the app and the whole test suite, then report. Use after any change to src/
and before committing.
---

1. From src/, start the server in the background ...
```

An MCP server is added with one command. Add `--scope project` so it's written to `.mcp.json` and committed with your repo:

```
claude mcp add --scope project playwright -- npx -y @playwright/mcp@latest
```

`/skills`, `/hooks` and `/mcp` inside Claude Code show what's loaded.

7. Secrets, with an agent in the room

The agent can read every file in your repo, and run any command you let it run. So:

- **Never paste a secret into the chat.** Whatever you type is sent to the model.
- Your Postgres password is inside `DATABASE_URL`, and it stays in Railway's variables, as a reference to the database's: never in a file, a commit, or the chat.
- For the stretch, put your Glassroom key in the environment of the terminal that runs the server, without writing it to a file:

  ```
  read -s GLASSROOM_API_KEY && export GLASSROOM_API_KEY    # paste, then Enter
  uvicorn main:app --reload
  ```

  On Railway it's one more variable on your app service.
- You can forbid the agent from reading a file at all, with a rule in `.claude/settings.json`: `{"permissions": {"deny": ["Read(./.env)"]}}`.

`test_repo.py` searches your whole history for database passwords and API keys. If one ever lands in a commit, deleting it doesn't help. Get a new one (a new password from Railway, or a new key from us) so the old one is useless.

The lab, in slices

A suggested order: each slice is a branch and a PR, and each ends with something that works. Run everything about the app from `src/`, and `claude` and `docker` from the repo root.

Slice 1 — The API (§1–§5)

Plan mode first. Point the agent at `CONTRACT.md` and the tests, and ask for a plan for the API only. Check the plan names your stack and doesn't touch the page yet. Then build until this passes, with the server running in another terminal:

```
pytest tests/test_contract.py tests/test_quality.py
```

One contract test is about the page (`GET /` serves it), and it fails until slice 2; the page tests skip until then.

Before you open the PR, run each ⚠ from the contract yourself, with `curl` or `/docs`: an unrated dish (`null`, not `0`), `stars` of 6, a dish that doesn't exist. Then read the list query. Is it a `LEFT JOIN`? What does it count? You'll be asked.

Slice 2 — The page

Everything Lab 6 asked of a page still holds: one `<h1>`, labels, real buttons, `textContent`, `response.ok`, loading and error states, phone width. Put those in the prompt, or better, in `CLAUDE.md`. Then check it the Lab 6 way: Network tab, **Offline**, a stopped server, a 375-pixel-wide window.

Decide what an unrated dish looks like. `null` isn't a number, so `avg_rating.toFixed(1)` throws, and `${avg_rating}` prints "null".

Slice 3 — Your feature

One thing the contract doesn't ask for, chosen by you: filtering by hall, a "best this week", one rating per person, a photo URL, a search box. Small is fine. What matters is that you can say why you chose it, how you checked it, and what the agent got wrong along the way. It doesn't change anything the contract promises.

Settle your tables here, before you deploy. `create_all` creates a table that's missing, but never changes one that exists, so a column added once Postgres has your tables never reaches them, and the app breaks on the first query that uses it. A feature that adds a whole new table is fine at any point.

Slice 4 — Deploy, with Postgres

Two parts: what Lab 6 already did, and the database it didn't have. The video *From Compose to Railway* walks through every click.

The container. A `Dockerfile` and `.dockerignore` at the repo root, as in Lab 6: `python:3.13-slim`, your requirements, `COPY src/`, and `uvicorn` on `0.0.0.0` and `$PORT`. Have your agent write them, then read them against your Lab 6 ones, line by line. Build and run it in your codespace before anything goes to Railway.

The database. Four changes to the app, and each is a line or two:

- It reads `DATABASE_URL` from the environment, and falls back to `sqlite:///app.db` when that isn't set, so your codespace still works without Postgres.
- It rewrites a URL that starts `postgresql://` to `postgresql+psycopg://`, because psycopg (version 3) is the driver that's installed. `psycopg[binary]` goes in the requirements file your image installs from.
- `GET /health` runs `SELECT 1`, so it only says 200 when the database answers (§5).
- Its first log line says which database it's using, with the password starred out, as in the video: SQLAlchemy's `engine.url` stars it out for you.

Then on Railway, the video's five steps: a project from your repo; a **Postgres** service beside your app; a `DATABASE_URL` variable on your app that's a **reference** to the database's, not a pasted copy; the health check path `/health` and a generated domain; and from then on, merging to `main` deploys. Read the deploy log: if it names SQLite, your app didn't get `DATABASE_URL`.

Then run the whole suite against production:

```
API_URL=https://your-app.up.railway.app pytest
```

At least one test in this suite can only fail in production. If everything was green in your codespace and something is red now, read the failure message: that's the lab working as intended.

Finally, rate something on your phone, then push a small change so Railway deploys again. The rating should still be there. That's the difference between this app and Lab 6′s.

Stretch — the AI summary (§7)

`GET /dishes/{id}/summary`, from the dish's ratings, through Glassroom. Same library and key as Lab 4: `anthropic`, with `base_url="https://litellm.glassroomai.com"` and `api_key` from `GLASSROOM_API_KEY`. Add `anthropic` to the requirements file your image installs from (ask your agent to ask you first: it's a dependency). Every call costs real money, billed to the course, so it only calls the model when someone asks for a summary, and never for a dish with no ratings. When the model is down or the key is missing, the API says **503** and the page says so politely. That's Lab 6′s "honest about the network", one server further back.

Throughout — the concept map

Your `README.md`, at the repo root, gets a section headed **Concept map**: one row per concept, with the idea board, your app, and where it lives in your code. Fill it in as you go, from code you've read, not from memory. These rows, at least:

Concept	Idea board	Dish ratings	Where it lives here
The parent table			
The child table, and what joins it to its parent			
What a client may send (the request classes)			
A bad request rejected at the edge (422)			
A missing parent (404)			
The aggregate on the list			
The row a plain `JOIN` would drop			
The page asks for the list			
A form, a `POST`, and a re-render			
Honest about the network			
One origin: the page and the API on one server			
Config from the environment			
Where the data lives in production			
Something with no idea-board equivalent			

The last row has more than one good answer. `test_repo.py` checks that the section exists and has its rows. Whether they're right is the check-in.

Running the tests

All from `src/`.

Where	Command	What runs
Your codespace, server running	`pytest`	Everything. The https check skips, and the stretch skips until you build it.
Anywhere, no server	`pytest -m repo`	Just your git history and repo files
Against production	`API_URL=https://… pytest`	Everything, against the real thing

`pytest -m contract` is the must-pass set. The failure messages are written to say what to fix, which makes them good things to hand straight to your agent. Passing everything is the floor, not the ceiling.

The tests add dishes named `test-dish-…` to whatever database they run against, production included. That's fine: they're real rows.

What to submit on Canvas

1. **A one-page executive summary.** At the top: your **live URL** and your **PR links**. Then:
   - one thing the agent got wrong, and how you found out (a test, the page, the diff, production);
   - one rule in your `CLAUDE.md` that came from a mistake;
   - your workflow extension: what it does, and what it fixed;
   - your feature, and why;
   - if you did the stretch, one summary it wrote, and whether you trust it.
2. **Your repo, and the exact commit you're submitting.** On `main`, after `git pull` has brought your last merge back:

   ```
   git switch main && git pull
   git status                  # nothing left uncommitted? commit it first
   git tag lab-07
   git push origin lab-07
   git rev-parse lab-07        # paste this hash into Canvas, with your repo URL
   ```

3. **Your git history**, as a file:

   ```
   git log --graph --stat lab-07 > lab-07-git-log.txt
   ```

   Download it (right-click in the VS Code explorer → **Download**), upload it, then delete it.

We grade the hash. Made a fix after tagging? Commit it, `git tag -f lab-07`, `git push -f origin lab-07`, and resubmit the new hash.

How this is graded

Your lab grade comes primarily from a **10-minute one-on-one check-in**, not from the submission. You demo your work and answer questions about it, **without AI**, and that includes questions about code your agent wrote.

Use your agent as much as you like while you build. Build in a way that leaves you able to explain everything you shipped.

Come with your live URL open on your phone and your repo open in your codespace.

Check-in topics

- **Demo:** Your live URL on a device that isn't your laptop: add a dish and rate it.

- **Map:** Pick a row of your concept map and show me both sides. Then trace a rating from the star you click to the number on the page, through your code, without your agent.
- **Judgment:** Open one of your PRs. What did the agent get wrong, and how did you find out?
- **Live change:** A small change to your app, made by hand, right now.
- **Workflow:** Walk me through your `CLAUDE.md` and your workflow extension. Which rule came from a mistake?

Getting help

- The Lab 7 tutor in Glassroom AI.
- Office hours
- Zulip, for concepts. Not for each other's code.