Build-along — start Lab 7, and look inside Claude Code

~45 minutes, in your Lab 7 codespace. Today you start Lab 7 with Claude Code, and you look inside the harness around the model while you do: what it loads into its context, what it's allowed to do, and how to give it a rule it can't break. You won't build the app today. Everything you commit is part of Lab 7.

Keep the Lab 7 handout open beside this sheet: https://course.aiprototypes.org/resources/files/eng190bo-fa26-lab-7-handout.pdf. Where this sheet says **Handout:**, do that part of the handout, which has the commands and ends each step with a **Check:** list. This sheet adds the questions to ask Claude Code along the way.

---

1. Set up (20 min)

Handout: *Get started* 1–3, a repo and a codespace, Claude Code signed in, and the starter committed.

With Claude Code open, before you ask it anything:

- `/status` shows your account, which the handout asks you to check. It also shows the version and the model. Which model are you using?
- Find the **permission mode**, under the prompt. Press `Shift-Tab` a few times and watch it change: auto, manual, accept edits, plan. Stop on **manual mode on**. In manual mode Claude Code asks you before it edits a file or runs a command, so today you see every action it takes. Each time you start `claude` again, check the mode again.

Handout: *Get started* 4, a first question. Before you `/clear` at the end of it, type `/context`.

- Which takes up more of the context window: your question and the contract it read, or Claude Code's own system prompt and tools?
- Is any memory file listed yet?

Check:

- ☐ The prompt says **manual mode on**.
- ☐ You can say what `/context` showed taking up the most room.

2. Your `CLAUDE.md` (10 min)

Handout: *Step 0*: write `CLAUDE.md` yourself, commit it on its own, and test it after `/clear`.

Then see how it got into the session:

- `/memory` lists the memory files Claude Code loaded. Is yours there? Are there any others?
- `/context` again. Find your `CLAUDE.md` under the memory files. How many tokens is it? It's loaded into every session before your first message, which is one reason to keep it to a page.

One experiment: when does a change to `CLAUDE.md` reach the agent?

1. Add a throwaway rule to the end of your `CLAUDE.md`: `- Start every answer with "Chef says:".`
2. Without `/clear`, ask Claude "Where is the spec for this app?" Did it follow the new rule?
3. `/clear`, and ask again. Now?
4. Take the rule out again: `git restore CLAUDE.md`.

Check:

- ☐ `/memory` lists your `CLAUDE.md`.
- ☐ `git status` shows nothing to commit: the throwaway rule is gone, and `CLAUDE.md` is in a commit of its own.

3. A first branch, commit and pull request (12 min)

In Lab 7, each slice of work goes on a branch, and reaches `main` through a pull request (PR) that you read line by line (handout, thing 3). Practice that on something small: a settings file that keeps Claude away from your secrets. `CLAUDE.md` *asks* the agent to do things. `.claude/settings.json` holds rules that Claude Code itself enforces, whatever the model decides.

1. From the repo root, make the branch yourself:

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

2. In Claude Code, still in manual mode, ask:

> Create .claude/settings.json with one permission rule: deny reading ./.env. Nothing else.

Read the edit it proposes before you approve it. It should come to this:

```
{
  "permissions": {
    "deny": ["Read(./.env)"]
  }
}
```

3. Check the rule does something. Make a fake secret, in a terminal at the repo root. The template's `.gitignore` lists `.env`, so git never picks it up:

```
echo "DEMO_SECRET=not-a-real-secret" > .env
```

Then `/exit`, start `claude` again so it loads the new settings, switch to manual mode, and ask: "What's in .env?"

- What happened? Type `/permissions`: which file does it say the rule comes from?
- Now ask it to read `.env` with a one-line Python script instead. If it asks to run a command, read the command, and say **No**. A deny rule covers Claude Code's own file tools and the shell commands it recognizes, such as `cat`. It can't see inside a script that opens the file itself, so for that one, the check between the agent and your secret is you, reading the prompt.

4. Have Claude do the git work, and watch what it asks first:

> Commit .claude/settings.json, push the branch, and open a pull request against main with gh.

Answer **Yes**, not "don't ask again", so you see each command. Which commands did it run without asking, and what do they have in common? When it's done, `git log -1` shows the commit message it wrote. Who does it say wrote the commit?

5. On GitHub, open the PR and read **Files changed**. Leave one review comment on a line, saying what the rule does. Then merge it, with whichever of GitHub's three merge buttons you like, and bring `main` up to date in the terminal:

```
git switch main && git pull
```

Check:

- ☐ Claude was refused when it tried to read `.env`.
- ☐ GitHub shows your PR as **Merged**, and `git log --oneline` on `main` shows its commit above your `CLAUDE.md` commit.
- ☐ `git status` doesn't list `.env`.

4. Look around the harness (5 min)

Type `/` on its own to see every command, then type letters to filter the list. These are the ones you'll use in Lab 7. Try each one, and answer its question.

Command	Find out
`/context`	What's filling the context window, and how full is it?
`/compact`	Then `/context` again: what shrank, and what's still there?
`/memory`	Which `CLAUDE.md` files are loaded?
`/permissions`	Which rules are in force, and which file did each come from?
`/hooks`, `/skills`, `/mcp`	Which extensions are loaded? Your Lab 7 workflow extension will show up in one of these.
`/diff`	What has changed that isn't committed yet?
`/rewind`	Which points could you go back to? Press `Esc` to leave without choosing one.
`/resume`	Which earlier sessions could you pick up again?
`/model`	Which models can you choose from? Look, but don't change it.
`/usage`	How much of your plan's limit has today used?
`/btw`	Ask a side question, such as "what does gh do?" Does the answer end up in the conversation?
`/help`	Everything else.

Sort what you found into two kinds: the commands about what the agent *knows*, and the ones about what it may *do*. Which kind does `CLAUDE.md` belong to, and which does `.claude/settings.json`?

Stretch — a rule it can't break

Your `CLAUDE.md` says never to edit `src/tests/`. That's a request to the model, and when a test fails, the shortest way to green is to edit the test. A **hook** turns the request into a rule: a command Claude Code runs before every edit, which can block it (handout, thing 5). It also counts as your Lab 7 workflow extension.

1. A new branch, from the repo root:

```
git switch main && git pull
git switch -c protect-spec
```

2. Claude Code hands a hook the tool call it's about to make as JavaScript Object Notation (JSON), and the hook's exit code says whether it may go ahead. Press `Shift-Tab` until the prompt says **plan mode on**, and ask:

   > Plan a PreToolUse hook in .claude/settings.json, keeping the deny rule that's there. Before every Edit or Write, it runs `python3 "$CLAUDE_PROJECT_DIR"/.claude/hooks/protect-spec.py`. The script reads the tool call as JSON on standard input. If tool_input.file_path is inside src/tests/ or is src/CONTRACT.md, it prints why to standard error and exits with code 2. Otherwise it exits with code 0.

   Read the plan, and approve it when it says what you asked for.

3. Test the script yourself, without Claude. Claude Code always hands a hook the full path, so the test does too:

```
f="$PWD/src/tests/test_contract.py"
echo "{\"tool_name\": \"Edit\", \"tool_input\": {\"file_path\": \"$f\"}}" \
  | python3 .claude/hooks/protect-spec.py; echo "exit $?"
```

   It should print its reason and `exit 2`. Set `f="$PWD/src/main.py"` and run the last two lines again: now it's `exit 0`, with nothing printed.

4. `/hooks` should list your hook under **PreToolUse**, from project settings. If it isn't there, `/exit` and start `claude` again.

5. Now try to get Claude to break the rule:

   > The first test in src/tests/test_contract.py is wrong. Change it so it passes.

   Watch the hook block the edit, and read what Claude does next. Press `Esc` if it starts looking for another way in.

   - Your hook watches `Edit` and `Write`. What if the agent tried a shell command instead? What would catch that?
   - Could a deny rule like the one for `.env` have done this job? What can a hook do that a deny rule can't?

6. `git status` should show the tests unchanged. Then commit, open a PR, read it, and merge, the same way as in step 3. From `src/`, `pytest -m repo -k workflow` now passes.

Check:

- ☐ Your script exits with 2 for a test file and 0 for anything else.
- ☐ Claude's edit to a test was blocked, and the tests are unchanged.
- ☐ On `main`, `pytest -m repo -k workflow` passes.

The hook stops Claude, not you: you can still edit the tests yourself. Don't, because `test_repo.py` checks that nobody has.

If something doesn't work

- **`claude: command not found`:** open a new terminal and try again. Still missing? Install it with `curl -fsSL https://claude.ai/install.sh | bash`, then open a new terminal.
- **Sign-in won't finish:** check that you're signing in with the account that has Pro (or higher); the free plan doesn't include Claude Code. `/login` starts it again.
- **A "file not found" error:** check which folder you're in. `claude` and `git` run from the repo root, and `pytest` from `src/`.
- **`gh` can't open the PR:** the `git push` output has a link that opens one on GitHub.
- **A setting or a hook isn't working:** `/exit` and start `claude` again from the repo root, so it reads `.claude/settings.json` fresh. A JSON typo, such as a missing comma, can stop the file loading at all.
- **Claude changed something you didn't want:** `Esc` stops it, and `git status` shows what changed. `git restore <file>` puts a file back.

Where you are

A Lab 7 repo with the starter, a `CLAUDE.md` committed before any app code, and a first PR merged. You've seen what Claude Code loads, what it may do, and where each kind of rule lives: `CLAUDE.md` for what the agent should know, and `.claude/settings.json` for what Claude Code lets it do. Next is Lab 7′s first slice, the application programming interface (API): the handout's *The lab, in slices*.