Build-along — two agents, two previews, one duck

~60 minutes, in a new codespace. Today you run two Claude Code agents at once, each in its own git worktree, on a small app called Rubber Duck. Each agent opens a pull request (PR), and Railway deploys each PR to its own live copy of the app, so you can try both before either reaches `main`. Then you teach the agent something new, with a skill or a Model Context Protocol (MCP) server.

Everything you make today is for practice: a new repo and a new Railway project, which you can delete at the end. You need the Claude account you used for Lab 7 (Pro or higher), and the Railway account you deployed Lab 6 with.

---

0. A repo for the duck (10 min)

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: `duck`. Visibility: Private.**
2. In `duck`, click **Code → Codespaces → Create codespace on main**, and wait for `E190BO environment ready.`
3. In the codespace terminal, at the repo root, download the app, and tell git to ignore the folder Claude Code keeps its worktrees in:

   ```
   curl -LO https://course.aiprototypes.org/resources/files/eng190bo-fa26-lecture-16-activity.zip
   python -m zipfile -e eng190bo-fa26-lecture-16-activity.zip .
   rm eng190bo-fa26-lecture-16-activity.zip
   echo ".claude/worktrees/" >> .gitignore
   ```

   You now have:

   - **`src/main.py`**, the application programming interface (API): `GET /api/question` picks a debugging question, and `GET /api/about` says where the app is running.
   - **`src/web/`**, the page, served by the same app.
   - **`src/test_app.py`**, five tests.
   - **`Dockerfile`** and **`.dockerignore`**, at the top of the repo, where Railway looks for them.
   - **`CLAUDE.md`**, the stack, the commands and the rules, already written. Read it.
4. Run it the way you always have:

   ```
   cd src
   uvicorn main:app --reload
   ```

   Open port 8000 from the **PORTS** tab, tell the duck about a bug, and press **Ask the duck**. The bottom of the page says where it's running. Stop the server with **Ctrl-C**, and `cd ..` back to the repo root.
5. Now run the same app in a container, the way Railway will:

   ```
   docker build -t duck .
   docker run --rm -p 8000:8000 duck
   ```

   Open port 8000 again. It's the same page, but now it's running from an image. What does the bottom of the page say, and where does that word come from? Read `about()` in `src/main.py` if you're not sure. **Ctrl-C** to stop it.
6. Commit the app, and push it:

```
git add -A
git commit -m "Add the Rubber Duck app"
git push
```

7. Start `claude` once from the repo root, sign in, and `/exit`. This codespace is new, so Claude Code asks whether to trust this folder: say yes. It won't make a worktree in a folder it doesn't trust yet.

Check:

- ☐ The page works with `uvicorn`, and in Docker.
- ☐ Your `duck` repo on github.com shows the commit, and the `.gitignore` line for `.claude/worktrees/`.
- ☐ `claude` starts without asking you to sign in again.

1. Deploy it, and turn on PR environments (8 min)

1. Sign in at https://railway.com, and click **New Project → GitHub repo → `duck` → Deploy Now**. If `duck` isn't listed, let Railway's GitHub app see it first: it's private. The build log says `Using detected Dockerfile!`.
2. On the new service: **Settings → Networking → Public Networking → Generate Domain**. Open the `https://…up.railway.app` address. The bottom of the page should say **production**.
3. Turn on preview deployments: open the project's settings, then **Environments → Enable PR Environments**.

From now on, every PR you open on `duck` gets its own copy of the app, at its own address, until you merge or close it.

Check:

- ☐ Your production address shows the duck, running in **production**.
- ☐ PR Environments are on.

2. Two agents, two worktrees (13 min)

A git worktree is a second working folder for the same repo, on its own branch. Two agents in two worktrees can't edit each other's files, and each one's work becomes its own PR. The two tasks below are independent: one changes only the page, and the other only the API. That's what makes them safe to run at the same time.

1. In the terminal, at the repo root, start the first agent in a worktree called `history`:

   ```
   claude -w history
   ```

   Press `Shift-Tab` until the prompt says **plan mode on**, and ask:

   > Under the duck's answer, show the last five questions it has asked, newest first. Change only the page, in src/web/; don't touch src/main.py.

   Read the plan. When it says what you asked for, approve it, and let it build.

2. Open a second terminal, with the **+** on the terminal panel. In it, from the repo root, start the second agent:

   ```
   claude -w topics
   ```

   Plan mode again, and ask:

   > Give every question a topic: git, python or web, with at least three questions in each. GET /api/question takes an optional ?topic= and picks a question from that topic; an unknown topic is a 404 with a message saying which topics there are. With no topic, it works as it does now. Change only src/main.py and src/test_app.py, add tests for the topics, and run all the tests.

While this one plans, go back to the first agent and see where it's got to. That's the rhythm: read one agent's work while the other one builds.

3. Open a third terminal, and look at what you've made:

```
git worktree list
ls -a .claude/worktrees/history
```

- How many working folders does this repo have now, and which branch is each one on?
- Is there a `.env` in the `history` worktree? The codespace made one at the repo root when it was created. A worktree is a fresh checkout, so it only has the files git tracks. A `.worktreeinclude` file at the repo root, listing `.env`, would copy it into every new worktree.

Check:

- ❒ Both agents have finished, and the `topics` agent says all the tests pass.
- ❒ `git worktree list` shows three folders: the repo root on `main`, and `worktree-history` and `worktree-topics`.

3. Try both, side by side (5 min)

Each worktree is a complete copy of the app, so you can run both at once, as long as each has its own port. In the third terminal, at the repo root:

```
(cd .claude/worktrees/history/src && uvicorn main:app --port 8001) &
(cd .claude/worktrees/topics/src && uvicorn main:app --port 8002) &
```

Open ports 8001 and 8002 from the **PORTS** tab.

- On 8001, ask the duck a few times: does the history work?
- On 8002, try a topic, and one that doesn't exist:

```
curl "localhost:8002/api/question?topic=git"
curl -i "localhost:8002/api/question?topic=cooking"
```

Then stop both servers:

```
kill %1 %2
```

What would have happened if both had used port 8000?

Check:

- ❒ You saw both versions running at once, on two ports.

4. A pull request, and a preview, for each (8 min)

1. In each agent's terminal, ask:

   > Commit this, push the branch, and open a pull request against main with gh. In the description, say what changed and how to check it.

2. On github.com, open each PR. Within a minute or two, Railway's bot comments with the address of that PR's own environment. Open both addresses.
   - What does the bottom of each page say now?
   - On the `topics` preview, add `/api/question?topic=git` to the address.
   - Open your production address too. Does it have either change?

3. Read each PR's **Files changed**. Did each agent stay inside the files you gave it?

Don't merge yet: the next part uses the PRs while they're open.

Check:

- ☐ Two PRs, each with a comment from Railway, and two previews that both work.
- ☐ Production still says **production**, and has neither change.

5. Teach the agent something (10 min)

Do **one** of these. A skill teaches the agent a procedure: steps it follows when they're needed. An MCP server gives it an ability it doesn't have: here, a way to see your Railway project.

Option A — write a skill

A skill is a Markdown file of instructions, in `.claude/skills/<name>/SKILL.md`. The agent sees only its one-line `description`, until a request matches it; then it reads the whole file. This one checks the app is ready to ship, the same way every time.

1. A third worktree, from the repo root:

```
claude -w ship-check
```

2. Ask it to write the skill:

> Create .claude/skills/ship-check/SKILL.md with exactly this content, and nothing else:

Then paste:

```
---
name: ship-check
description: Checks this app is ready to ship. Use when asked whether a change is ready to
merge or deploy, or to run every check.
---

Run each check from the repo root, in order, and report each one as PASS or FAIL.
For anything that fails, show the last lines of its output.
Don't fix anything: report.

1. Tests: `cd src && python -m pytest -q`
2. Image: `docker build -t duck-check .`
3. Run it the way Railway does, with PORT set:
   `docker run -d --rm --name duck-check -e PORT=8099 -p 8099:8099 duck-check`
4. Wait up to 10 seconds for `curl -fsS localhost:8099/health` to answer.
   Then check that `curl -fsS localhost:8099/api/question` returns JSON with a "question".
5. Always stop the container, even if a check failed: `docker stop duck-check`

End with one line: READY TO SHIP, or NOT READY and the reason.
```

3. Claude Code reads skills when a session starts, so `/exit`, and open the same worktree again: `claude -w ship-check`. Type `/skills`: is `ship-check` listed?
4. Now ask, without naming the skill:

> Is this ready to ship?

Did it use the skill? How can you tell? Then try it by name: `/ship-check`.

5. Have it commit the skill, push, and open a PR, as in part 4. This PR gets a preview too.

- Why is this a skill, and not a paragraph in `CLAUDE.md`?
- `CLAUDE.md` says "all the tests pass before you commit". What would make that a rule the agent *can't* skip?

Option B — add Railway's MCP server

Railway's MCP server lets the agent read your Railway project: its environments, deployments and logs. It uses the Railway command-line interface (CLI), which is installed in the codespace, so first sign the CLI in.

1. In a terminal at the repo root:

```
railway login
railway mcp install --agent claude-code
```

`railway login` shows a link and a code: open the link, sign in, and enter the code.

2. Start `claude` from the repo root, and type `/mcp`. Is `railway` listed and connected? Choose it to see its tools.
3. Press `Shift-Tab` until the prompt says **manual mode on**, so it asks before each tool call, and ask:

> Using Railway, list the environments in my duck project, with each one's address. Then show me the last 20 lines of the deploy log for one of the PR environments.

Read each tool call before you approve it.

- Which of the server's tools could change or delete something in your Railway project?
- The server signs in as you. What should you do before adding one to a project you share with a team?

Check:

- ☐ Option A: `/skills` lists `ship-check`, it ran every check, and its PR is open.
- ☐ Option B: the agent listed your environments, including the PR environments, through the Railway server.

6. Merge, and start again from `main` (5 min)

1. On github.com, merge each PR, with any of the three merge buttons. Did any of them conflict? Why not?
2. On Railway, watch production deploy again, and the PR environments disappear. Open your production address: are both changes there?
3. In each agent's terminal, `/exit`. Claude Code asks whether to keep the worktree: its work is merged, so remove it.
4. At the repo root, bring `main` up to date:

```
git pull
git worktree list
```

One folder again. The next task starts from here, with a new session: `claude -w <next-task>`.

5. Clean up, so nothing keeps running: on Railway, open the project's settings, then **Danger → Delete Project**. You can delete the `duck` repo too, or keep it.

Check:

- ☐ Production shows the history and the topics.
- ☐ `git worktree list` shows only the repo root.
- ☐ The Railway project is deleted.

If something doesn't work

- **`claude -w` says the folder isn't trusted:** run `claude` once from the repo root, accept, and `/exit`.
- **`claude -w` says it can't resolve a base branch:** the repo needs a commit on github.com first. Push, then try again.
- **Railway won't create a project:** your plan may have run out of projects. Delete one you no longer need, such as your Lab 6 project, and try again.
- **No comment from Railway on the PR:** check that PR Environments are on, and that the PR was opened after you turned them on. Closing and reopening the PR tries again.
- **A preview has no address:** generate a domain for the production service first; PR environments only get one when production has one.
- **`Address already in use`:** something else has that port. `kill %1 %2` stops the servers you started in part 3, and **Ctrl-C** stops one in the foreground.
- **`/skills` doesn't list your skill:** check the path is `.claude/skills/ship-check/SKILL.md` inside the worktree, and that the file starts with `---`. Then `/exit` and `claude -w ship-check` again.
- **`/mcp` doesn't list `railway`:** `/exit` and start `claude` again. If it's listed but not connected, run `railway login` again.

Where you are

Two agents built two features at once, in two worktrees, without touching each other's files. You ran both versions side by side on your own ports, and then Railway ran each PR on the internet before it reached `main`. And you either taught the agent a procedure with a skill, or gave it a new ability with an MCP server. That's the loop for your project: one task, one worktree, one PR with a preview, then merge and start again from `main`.